```swift
////////////////////////////////////////////////////////////////////////
// Swift source code of main.swift Version 8.0.2 at greenchess.com //
////////////////////////////////////////////////////////////////////////

// incorporate needed system files //
import AppKit
import AVFoundation

// the game info list data source implementation //
class GameInfoDataSource:NSObject, NSTableViewDataSource
{
   // a loop index //
   var Index:Int = 0

   // the height of a row //
   var Height:Int = 0

   // a list of game info //
   var GameInfoList:[NSAttributedString] = []

   // the row heights //
   var Heights:[Int] = []

   // refresh the game info list //
   func RefreshGameInfoList(_ SavedGameInfoList:[String], Color:NSColor, Font:NSFont)
   {
      // initialize row height to one //
      Height = 1

      // clear the game info list //
      GameInfoList.removeAll()

      // clear the row heights list //
      Heights.removeAll()

      // loop over the game info list //
      for Index in 0 ..< SavedGameInfoList.count
      {
         // append the attributed move string //
         GameInfoList.append(NSAttributedString(string:SavedGameInfoList[Index], attributes:
                             [NSAttributedStringKey.font:Font, NSAttributedStringKey
                              .foregroundColor:Color]))

         // if the row height is one //
         if Height == 1
         {
            // get the row height //
            Height = Int(GameInfoList[Index].size().height) + 5
         }

         // if less than twenty-five characters in game info list //
         if SavedGameInfoList[Index].count < 25
         {
            // append the row height //
            Heights.append(Height)
         }
         // more than twenty-five characters in game info list string //
         else
         {
            // append the row height //
            Heights.append((Int(GameInfoList[Index].size().width / 200) + 1) * Height)
         }
      }
   }

   // return the row height //
   func RowHeight(_ Row:Int) -> Int
   {
      // return the row height //
      return Heights[Row]
   }

   // the number of rows in the game info moves table view //
   func numberOfRows(in TableView:NSTableView) -> Int
   {
      // return the number of moves in the game info list //
      return GameInfoList.count
   }
}

// the pgn moves list data source implementation //
class PGNMovesDataSource:NSObject, NSTableViewDataSource
{
   // the paragraph style to use //
   var ParagraphStyle:NSMutableParagraphStyle = NSMutableParagraphStyle()

   // the maximum number of rows to display //
   var MaxRows:Int = 0

   // a loop index //
   var Index:Int = 0

   // the height of a row //
   var Height:Int = 0

   // a list of pgn moves //
   var PGNMovesList:[NSAttributedString] = []

   // the row heights //
   var Heights:[Int] = []

   // refresh the pgn move list //
   func RefreshPGNMovesList(_ SavedPGNMovesList:[String], Color:NSColor, Font:NSFont)
   {
      // set the paragraph alignment to be justified //
      ParagraphStyle.alignment = .justified

      // initialize row height to one //
      Height = 1

      // clear the move list //
      PGNMovesList.removeAll()

      // clear the row heights list //
      Heights.removeAll()

      // loop over the move list //
      for Index in 0 ..< SavedPGNMovesList.count
      {
         // append the attributed move string //
         PGNMovesList.append(NSAttributedString(string:SavedPGNMovesList[Index], attributes:
                             [NSAttributedStringKey.font:Font, NSAttributedStringKey
                              .foregroundColor:Color, NSAttributedStringKey.paragraphStyle:
                              ParagraphStyle]))

         // if the row height is one //
         if Height == 1
         {
            // get the row height //
            Height = Int(PGNMovesList[Index].size().height) + 5
         }

         // if less than twenty characters in move list string //
         if SavedPGNMovesList[Index].count < 20
         {
            // append the row height //
            Heights.append(Height)
         }
         // twenty or more characters in move list string //
         else
         {
            // append the row height //
            Heights.append((Int(PGNMovesList[Index].size().width / 200) + 1) * Height)
         }
      }
   }

   // return the row height //
   func RowHeight(_ Row:Int) -> Int
   {
      // return the row height //
      return Heights[Row]
   }

   // the number of rows in the current pgn moves table view //
   func numberOfRows(in TableView:NSTableView) -> Int
   {
      // return the number of rows in the pgn moves list to display //
      return min(PGNMovesList.count, MaxRows)
   }
}

// the implementation for the game list data source //
class GameListDataSource:NSObject, NSTableViewDataSource
{
   // initialize the selected row //
   var SelectedRow:Int = 0

   // the index in a list //
   var Index:Int = 0

   // the height of a row //
   var Height:Int = 0

   // the row heights //
   var Heights:[Int] = []

   // the game list indices //
   var GameListIndices:[Int] = []

   // the game list //
   var GameList:[String] = []

   // the attributed game list //
   var AttributedGameList:[NSAttributedString] = []

   // the original game list //
   var GameListOriginal:[String] = []

   // the original game list indices //
   var GameListIndicesOriginal:[Int] = []

   // the font //
   var Font:NSFont = NSFont()

   // return the row height //
   func RowHeight(_ Row:Int) -> Int
   {
      // return the row height //
      return Heights[Row]
   }

   // the size of the original database //
   func DatabaseSize() -> Int
   {
      // return the size of the original database //
      return GameListOriginal.count
   }

   // store the selected row //
   func StoreSelectedRow(_ Row:Int)
   {
      // store the selected row //
      SelectedRow = Row
   }

   // the number of rows in the current game list table view //
   func numberOfRows(in TableView:NSTableView) -> Int
   {
      // return the number of rows in the game list table view //
      return GameList.count
   }

   // store the game list //
   func StoreGameList(_ GameListToStore:[String])
   {
      // assign the game list //
      GameList = GameListToStore

      // remove the last object in the game list since it is empty //
      GameList.removeLast()

      // clear the game list indices //
      GameListIndices.removeAll()

      // loop over the entire game list //
      for Index in 0 ..< GameListToStore.count
      {
         // add the game index to the game list indices //
         GameListIndices.append(Index)
      }

      // save the original game list //
      GameListOriginal = GameList

      // save the original game list indices //
      GameListIndicesOriginal = GameListIndices
   }

   // condense the game list related to the current search string //
   func CondenseGameList(_ SearchString:String)
   {
      // if the search string is empty //
      if SearchString.isEmpty
      {
         // set the current game list to the original game list //
         GameList = GameListOriginal

         // set the current game list indices to the original game list indices //
         GameListIndices = GameListIndicesOriginal

         // exit function since don't need to condense the game list //
         return
      }

      // clear the game list //
      GameList.removeAll()

      // clear the game list indices //
      GameListIndices.removeAll()

      // loop over the entire original game list //
      for Index in 0 ..< GameListOriginal.count
      {
         // if the search string is found in the game list //
         if GameListOriginal[Index].lowercased().range(of:SearchString.lowercased()) != nil
         {
            // add the game from the original list to the current list //
            GameList.append(GameListOriginal[Index])

            // add the game list index from the original list to the current list //
            GameListIndices.append(GameListIndicesOriginal[Index])
         }
      }
   }

   // get the game index from the current selected row in the game table //
   func GetGameIndex(_ Row:Int) -> Int
   {
      // return the game index from the current selected row in the game list //
      return GameListIndices[Row]
   }

   // refresh the game list //
   func RefreshGameList(_ Color:NSColor, TheFont:NSFont)
   {
      // save the font type //
      Font = TheFont

      // initialize row height to one //
      Height = 1

      // clear the attributed game list //
      AttributedGameList.removeAll()

      // clear the row heights list //
      Heights.removeAll()

      // loop over the game list //
      for Index in 0 ..< GameList.count
      {
         // append the attributed game string //
         AttributedGameList.append(NSAttributedString(string:GameList[Index], attributes:
                                   [NSAttributedStringKey.font:Font, NSAttributedStringKey
                                    .foregroundColor:Color]))

         // if the row height is one //
         if Height == 1
         {
            // get the row height //
            Height = Int(AttributedGameList[Index].size().height) + 10
         }

         // if less than fifty characters in game list //
         if GameList[Index].count < 50
         {
            // append the row height //
            Heights.append(Height)
         }
         // more than fifty characters in game list string //
         else
         {
            // append the row height //
            Heights.append((Int(AttributedGameList[Index].size().width / 600) + 1) * Height)
         }
      }
   }

   // the game description for the specified game //
   func GameDescription(_ GameNumber:Int) -> String
   {
      // return the game description for the specified game //
      return GameListOriginal[GameNumber]
   }
}

// the game clock implementation //
class GameClockDisplay:NSView
{
   // an index in a list //
   var Index:Int = 0

   // the x location in the view //
   var X:Float = 0

   // the y location in the view //
   var Y:Float = 0

   // an angle in a circle in radians //
   var Angle:Float = 0

   // the elapsed time for white //
   var WhiteTime:Float = 0

   // the elapsed time for black //
   var BlackTime:Float = 0

   // the clock color //
   var ClockColor:NSColor = NSColor()

   // the clock background color //
   var ClockBackgroundColor:NSColor = NSColor()

   // the main window color //
   var WindowColor:NSColor = NSColor()

   // the clock background visibility //
   var ClockBackgroundVisibility:Int = 0

   // the face of the game clock //
   var ClockFace:NSBezierPath = NSBezierPath()

   // the border on the game clock //
   var ClockBorder:NSBezierPath = NSBezierPath()

   // the second hands on the game clock //
   var SecondHands:NSBezierPath = NSBezierPath()

   // the minute hands on the game clock //
   var MinuteHands:NSBezierPath = NSBezierPath()

   // the hour hands on the game clock //
   var HourHands:NSBezierPath = NSBezierPath()

   // update the game clock for the given white and black times //
   func UpdateClock(_ ElapsedWhiteTime:Float, ElapsedBlackTime:Float)
   {
      // update the display //
      self.needsDisplay = true

      // display if needed //
      displayIfNeeded()

      // get the elapsed time for white //
      WhiteTime = ElapsedWhiteTime

      // get the elapsed time for black //
      BlackTime = ElapsedBlackTime
   }

   // update the clock background visibility //
   func UpdateClockBackgroundVisibility(_ BackgroundVisibilityForClock:Int)
   {
      // update the clock background visibility //
      ClockBackgroundVisibility = BackgroundVisibilityForClock
   }

   // update the main window color //
   func UpdateWindowColor(_ WindowColorForClock:NSColor)
   {
      // update the main window color //
      WindowColor = WindowColorForClock
   }

   // update the game clock background color //
   func UpdateClockBackgroundColor(_ BackgroundColorForClock:NSColor)
   {
      // update the game clock background color //
      ClockBackgroundColor = BackgroundColorForClock
   }

   // update the game clock color //
   func UpdateClockColor(_ ColorForClock:NSColor)
   {
      // update the game clock color //
      ClockColor = ColorForClock
   }

   // create an analog game clock //
   func CreateGameClock()
   {
      // set the line width for the second hands //
      SecondHands.lineWidth = 1

      // set the line width for the minute hands //
      MinuteHands.lineWidth = 3

      // set the line width for the hour hands //
      HourHands.lineWidth = 6

      // make the ends of the second hands rounded //
      SecondHands.lineCapStyle = NSBezierPath.LineCapStyle.roundLineCapStyle

      // make the ends of the minute hands rounded //
      MinuteHands.lineCapStyle = NSBezierPath.LineCapStyle.roundLineCapStyle

      // make the ends of the hour hands rounded //
      HourHands.lineCapStyle = NSBezierPath.LineCapStyle.roundLineCapStyle

      // set the line width of the analog clock border //
      ClockBorder.lineWidth = 3

      // create the border for the white clock //
      ClockBorder.appendOval(in:NSMakeRect(3, 3, 125, 125))

      // create the border for the black clock //
      ClockBorder.appendOval(in:NSMakeRect(183, 3, 125, 125))

      // set the line width of the clock face //
      ClockFace.lineWidth = 1

      // create the center for the white clock //
      ClockFace.appendOval(in:NSMakeRect(60.5, 60.5, 10, 10))

      // create the center for the black clock //
      ClockFace.appendOval(in:NSMakeRect(240.5, 60.5, 10, 10))

      // loop over sixty tick marks //
      for Index in 0 ..< 60
      {
         // set the angle for the current tick mark //
         Angle = Float(Index) * Float.pi / 30

         // get the x location of the current tick mark //
         X = 65 + 60 * sin(Angle)

         // get the y location of the current tick mark //
         Y = 65 + 60 * cos(Angle)

         // if currently on an hour tick mark //
         if Index % 5 == 0
         {
            // make an hour tick mark for the white clock //
            ClockFace.appendOval(in:NSMakeRect(CGFloat(X - 1), CGFloat(Y - 1), 3, 3))

            // make an hour tick mark for the black clock //
            ClockFace.appendOval(in:NSMakeRect(CGFloat(X + 179), CGFloat(Y - 1), 3, 3))
         }
         // if currently on a minute tick mark //
         else
         {
            // make a minute tick mark for the white clock //
            ClockFace.appendOval(in:NSMakeRect(CGFloat(X), CGFloat(Y), 1, 1))

            // make a minute tick mark for the black clock //
            ClockFace.appendOval(in:NSMakeRect(CGFloat(X + 180), CGFloat(Y), 1, 1))
         }
      }
   }

   // the draw rect function for the game clock //
   override func draw(_ Rect:NSRect)
   {
      // set the clock border color to the chosen value //
      ClockColor.set()

      // draw the clock border //
      ClockBorder.stroke()

      // fill the clock border with the main window color //
      WindowColor.set()

      // fill the inside of the clock border //
      ClockBorder.fill()

      // set the clock background color with the selected alpha value //
      ClockBackgroundColor.withAlphaComponent(CGFloat(CGFloat(ClockBackgroundVisibility) / 100)).set()

      // fill the inside of the clock border //
      ClockBorder.fill()

      // set the clock border color //
      ClockColor.set()

      // draw the clock face //
      ClockFace.stroke()

      // fill the clock face //
      ClockFace.fill()

      // clear the second hands //
      SecondHands.removeAllPoints()

      // clear the minute hands //
      MinuteHands.removeAllPoints()

      // clear the hour hands //
      HourHands.removeAllPoints()

      // move to the start of the white second hand //
      SecondHands.move(to:NSMakePoint(65.5, 65.5))

      // get the x location of the end of the white second hand //
      X = 65.5 + 60 * sin(Float.pi * WhiteTime / 30)

      // get the y location of the end of the white second hand //
      Y = 65.5 + 60 * cos(Float.pi * WhiteTime / 30)

      // make line to end of white second hand //
      SecondHands.line(to:NSMakePoint(CGFloat(X), CGFloat(Y)))

      // move to the start of the black second hand //
      SecondHands.move(to:NSMakePoint(245.5, 65.5))

      // get the x location of the end of the black second hand //
      X = 245.5 + 60 * sin(Float.pi * BlackTime / 30)

      // get the y location of the end of the black second hand //
      Y = 65.5 + 60 * cos(Float.pi * BlackTime / 30)

      // make line to end of black second hand //
      SecondHands.line(to:NSMakePoint(CGFloat(X), CGFloat(Y)))

      // draw the second hands //
      SecondHands.stroke()

      // move to the start of the white minute hand //
      MinuteHands.move(to:NSMakePoint(65.5, 65.5))

      // get the x location of the end of the white minute hand //
      X = 65.5 + 50 * sin(Float.pi * WhiteTime / 1800)

      // get the y location of the end of the white minute hand //
      Y = 65.5 + 50 * cos(Float.pi * WhiteTime / 1800)

      // make line to the end of the white minute hand //
      MinuteHands.line(to:NSMakePoint(CGFloat(X), CGFloat(Y)))

      // move to the start of the black minute hand //
      MinuteHands.move(to:NSMakePoint(245.5, 65.5))

      // get the x location of the end of the black minute hand //
      X = 245.5 + 50 * sin(Float.pi * BlackTime / 1800)

      // get the y location of the end of the black minute hand //
      Y = 65.5 + 50 * cos(Float.pi * BlackTime / 1800)

      // make line to the end of the black minute hand //
      MinuteHands.line(to:NSMakePoint(CGFloat(X), CGFloat(Y)))

      // draw the minute hands //
      MinuteHands.stroke()

      // move to the start of the white hour hand //
      HourHands.move(to:NSMakePoint(65.5, 65.5))

      // get the x location of the white hour hand //
      X = 65.5 + 30 * sin(Float.pi * WhiteTime / 21600)

      // get the y location of the white hour hand //
      Y = 65.5 + 30 * cos(Float.pi * WhiteTime / 21600)

      // make line to the end of the white hour hand //
      HourHands.line(to:NSMakePoint(CGFloat(X), CGFloat(Y)))

      // move to the start of the black hour hand //
      HourHands.move(to:NSMakePoint(245.5, 65.5))

      // get the x location of the black hour hand //
      X = 245.5 + 30 * sin(Float.pi * BlackTime / 21600)

      // get the y location of the black hour hand //
      Y = 65.5 + 30 * cos(Float.pi * BlackTime / 21600)

      // make line to the end of the black hour hand //
      HourHands.line(to:NSMakePoint(CGFloat(X), CGFloat(Y)))

      // draw the hour hands //
      HourHands.stroke()
   }
}

// the captured pieces display interface //
class CapturedPiecesDisplay:NSView
{
   // the type of chess piece set to use //
   var ChessPieceSetType:Int = 0

   // the chess piece set type image index //
   var ChessPieceImageIndex:Int = 0

   // an index in a list //
   var Index:Int = 0

   // the number of captured pieces displayed to the view so far //
   var CapturedPieceCount:Int = 0

   // the list of captured pieces to display //
   var CapturedPiecesToDisplay:[Int] = [0, 0, 0, 0, 0]

   // whether the piece is white or not //
   var WhitePiece:Bool = false

   // the image of a piece //
   var PieceImage:NSImage = NSImage()

   // an array of chess piece images //
   var ChessPieceImages:[NSImage] = []

   // the piece width //
   var Width:Int = 0

   // set the chess piece set images //
   func SetChessPieceSetImages(_ ChessPieceSetImages:[NSImage])
   {
      // set the chess piece images //
      ChessPieceImages = ChessPieceSetImages
   }

   // update the captured pieces display //
   func UpdateCapturedPiecesDisplay(_ CapturedPieces:[Int], White:Bool, Set:Int)
   {
      // set the chess piece set type //
      ChessPieceSetType = Set

      // set whether captured pieces are white or not //
      WhitePiece = White

      // copy over the captured pieces //
      CapturedPiecesToDisplay = CapturedPieces

      // set that view needs display //
      self.needsDisplay = true

      // set to display view if needed //
      displayIfNeeded()
   }

   // set the piece image for the given piece type //
   func SetPieceImage(_ PieceType:Int)
   {
      // if the piece is white //
      if WhitePiece
      {
         // set the chess piece image index corresponding to color and piece set type //
         ChessPieceImageIndex = abs(PieceType - 4) + ChessPieceSetType * 12
      }
      // piece is black //
      else
      {
         // set the chess piece image index corresponding to color and piece set type //
         ChessPieceImageIndex = abs(PieceType - 4) + (ChessPieceSetType * 12 + 6)
      }

      // set the chess piece image //
      PieceImage = ChessPieceImages[ChessPieceImageIndex]
   }

   // redefine the draw rectangle function //
   override func draw(_ Rect:NSRect)
   {
      // initialize the captured pieces count //
      CapturedPieceCount = 0

      // loop over the list of captured pieces to display //
      for Index in 0 ..< 5
      {
         // set the piece image for the captured piece //
         SetPieceImage(Index)

         // loop over the number of captured pieces for this particular piece type //
         for _ in 0 ..< CapturedPiecesToDisplay[Index]
         {
            // draw the piece in the view //
            PieceImage.draw(in:NSMakeRect(CGFloat(CapturedPieceCount * Width), 0, CGFloat(Width),
                            CGFloat(Width)), from:NSZeroRect, operation:NSCompositingOperation
                            .sourceOver, fraction:1)

            // increment the captured piece count //
            CapturedPieceCount += 1
         }
      }
   }
}

// a window class //
class AWindow:NSWindow
{
   // the chess board //
   @IBOutlet var Board:GreenChess!

   // override the key down method //
   override func keyDown(with Event:NSEvent)
   {
      // call key down on the board //
      Board.InterceptKeyDown(Event)
   }

   // override the animation resize time //
   override func animationResizeTime(_ Frame:NSRect) -> TimeInterval
   {
      // return animation resize time of 0 seconds //
      return 0
   }
}

// the game gallery grid view //
class GalleryGrid:NSView, CAAnimationDelegate
{
   // the board composite //
   var BoardComposite:NSImage = NSImage()

   // the tracking rectangle tag //
   var TrackingRectangle:NSView.TrackingRectTag = NSView.TrackingRectTag()

   // set that not initially a valid shape //
   var ValidShape:Bool = false

   // set that not initially diagramming //
   var Diagramming:Bool = false

   // set that haven't initially entered tracking rectangle //
   var Entered:Bool = false

   // the location in the window //
   var Location:NSPoint = NSPoint()

   // the shape layers //
   var Shapes:[CAShapeLayer] = []

   // a shape layer //
   var Shape:CAShapeLayer = CAShapeLayer()

   // a shape path //
   var ShapePath:CGMutablePath = CGMutablePath()

   // the animation duration //
   var Duration:Double = 1.0

   // the snapshot bitmap image representation //
   var SnapshotBitmap:NSBitmapImageRep!

   // the snapshot image size //
   var SnapshotImageSize:NSSize = NSSize()

   // the starting x value //
   var StartX:Double = 0

   // whether going right or not //
   var GoingRight:Bool = false

   // the animation start time //
   var StartTime:Date = Date()

   // the elapsed time of the animation //
   var ElapsedTime:Double = 0

   // whether animating or not //
   var Animating:Bool = false

   // whether reviewing or not //
   var Reviewing:Bool = false

   // the height of the view //
   var Height:Double = 0

   // the width of the view //
   var Width:Double = 0

   // create an animation layer //
   var AnimationLayer:CALayer = CALayer()

   // create an animation //
   var Animation:CABasicAnimation = CABasicAnimation(keyPath:"position")

   // a loop index //
   var Index:Int = 0

   // the fractional offset from the game gallery or grid slider //
   var FractionalOffset:Double = 0

   // the x location value //
   var X:Double = 0

   // the y location value //
   var Y:Double = 0

   // the x location center of the animation layer //
   var CenterX:Double = 0

   // the maximum animation layer offset //
   var MaxOffset:Double = 0

   // the gallery width //
   var GalleryWidth:Double = 0

   // the grid width //
   var GridWidth:Double = 0

   // the width of one frame in the animation //
   var FrameWidth:Double = 0

   // the array of board frames //
   var Frames:[NSImage] = []

   // the gallery image //
   var GalleryImage:NSImage = NSImage()

   // the grid image //
   var GridImage:NSImage = NSImage()

   // the current image //
   var Image:NSImage = NSImage()

   // the snapshot image //
   var SnapshotImage:NSImage = NSImage()

   // whether is a grid or not //
   var Grid:Bool = false

   // the diagram cursor //
   var DiagramCursor:NSCursor = NSCursor()

   // the chess board //
   @IBOutlet var Board:GreenChess!

   // set the diagram cursor //
   func SetDiagramCursor(_ MarkerColor:String)
   {
      // set that currently diagramming //
      Diagramming = true

      // set the current diagram cursor //
      DiagramCursor = NSCursor.init(image:NSImage(named:NSImage.Name(rawValue:
                                    "marker_\(MarkerColor)_icon"))!, hotSpot:NSPoint(x:2, y:30))
   }

   // set the current animation rate //
   func SetAnimationRate(_ AnimationRate:Int)
   {
      // compute the duration for each frame to animate for //
      Duration = Double(101 - AnimationRate) / 10.0
   }

   // take a snapshot of the view //
   func TakeSnapshot()
   {
      // set the snapshot image size //
      SnapshotImageSize = NSMakeSize(CGFloat(Width), CGFloat(Height))

      // set the snapshot bitmap to gallery or grid size //
      SnapshotBitmap = self.bitmapImageRepForCachingDisplay(in:NSMakeRect(0, 0, CGFloat(Width),
                                                            CGFloat(Height)))!

      // set the board snapshot size //
      SnapshotBitmap.size = SnapshotImageSize

      // draw the specified image into the bitmap representation //
      self.cacheDisplay(in:self.bounds, to:SnapshotBitmap)

      // initialize a board snapshot image //
      SnapshotImage = NSImage(size:SnapshotImageSize)

      // add the board snapshot bitmap to the board snapshot image //
      SnapshotImage.addRepresentation(SnapshotBitmap)
   }

   // animation did start delegate //
   func animationDidStart(_ TheAnimation:CAAnimation)
   {
      // set the arrow cursor //
      NSCursor.arrow.set()

      // set that not currently a valid shape //
      ValidShape = false

      // get the animation start time //
      StartTime = Date()
   }

   // animation did stop delegate //
   func animationDidStop(_ TheAnimation:CAAnimation, finished Flag:Bool)
   {
      // if in tracking rectangle and diagramming //
      if Entered && Diagramming
      {
         // set the diagram cursor //
         DiagramCursor.set()
      }

      // set that no longer animating //
      Animating = false

      // update the gallery grid slider value //
      Board.GalleryGridSlider.doubleValue = SliderValue()

      // if x went past the minimum value or not reviewing //
      if (X <= -MaxOffset + CenterX) || !Reviewing
      {
         // set that no longer reviewing //
         Reviewing = false

         // set the gallery grid buttons to true //
         Board.SetGalleryGridButtonsToState(true)
      }
      // if reviewing //
      else
      {
         // go right in the game gallery grid //
```

```
      Board.GameGalleryGridRightAction(self)
   }
}

// stop the animation //
func StopAnimation()
{
   // if not animating //
   if !Animating
   {
      // exit function since not animating //
      return
   }

   // get the elapsed time of the animation //
   ElapsedTime = min(Duration, abs(StartTime.timeIntervalSinceNow))

   // if going right and the animation duration is greater than zero //
   if GoingRight && Animation.duration > 0
   {
      // set the new x location value //
      X += (Animation.duration - ElapsedTime) / Animation.duration * min(FrameWidth,
           abs(StartX - X))
   }
   // if going left and animation duration is greater than zero //
   else if !GoingRight && Animation.duration > 0
   {
      // set the new x location value //
      X -= (Animation.duration - ElapsedTime) / Animation.duration * min(FrameWidth,
           abs(StartX - X))
   }

   // update the animation layer position //
   AnimationLayer.position = CGPoint(x:CGFloat(X), y:CGFloat(Y))

   // set that no longer reviewing //
   Reviewing = false

   // remove all the gallery grid view animations //
   AnimationLayer.removeAllAnimations()
}

// the draw rect function //
override func draw(_ Rect:NSRect)
{
   // set the border color to gray //
   NSColor.gray.set()

   // draw the border //
   NSBezierPath(rect:self.bounds).stroke()
}

// if the mouse entered the tracking rectangle //
override func mouseEntered(with Event:NSEvent)
{
   // set that entered the tracking rectangle //
   Entered = true

   // if diagramming and not animating //
   if Diagramming && !Animating
   {
      // set the cursor to the diagram cursor //
      DiagramCursor.set()
   }
}

// if the mouse exited the tracking rectangle //
override func mouseExited(with Event:NSEvent)
{
   // set that no longer in the tracking rectangle //
   Entered = false

   // set the cursor back to the arrow cursor //
   NSCursor.arrow.set()
}

// get the current location //
func GetLocation(_ Event:NSEvent)
{
   // get the location in the window //
   Location = Event.locationInWindow

   // adjust the location x value //
   Location.x += CGFloat(CenterX - X) - self.frame.origin.x

   // adjust the location y value //
   Location.y -= self.frame.origin.y

   // if the location x value is too low //
   if Location.x <= CGFloat(CenterX - X)
   {
      // reset the location x value //
      Location.x = CGFloat(CenterX - X + 3.0)
   }

   // if location y value is too low //
   if Location.y <= 0
   {
      // reset the location y value //
      Location.y = 3.0
   }

   // if the location x value is too high //
   if Location.x >= CGFloat(Width + CenterX - X)
   {
      // reset the location x value //
      Location.x = CGFloat(Width + CenterX - X - 3.0)
   }

   // if the location y value is too high //
   if Location.y >= CGFloat(Height)
   {
      // reset the location y value //
      Location.y = CGFloat(Height - 3.0)
   }

   // if location is inside the tracking rectangle //
   if Location.x > CGFloat(CenterX - X + 3.0) && Location.x < CGFloat(Width + CenterX - X - 3.0)
      && Location.y > 3.0 && Location.y < CGFloat(Height - 3.0)
   {
      // set the diagram cursor //
      DiagramCursor.set()
   }
   // location is outside the tracking rectangle //
   else
   {
      // set the arrow cursor //
      NSCursor.arrow.set()
   }
}
```

```
// undo a shape //
func UndoShape()
{
   // if there is a shape to undo //
   if Shapes.count > 0
   {
      // remove the shape layer from the super layer //
      Shapes[Shapes.count - 1].removeFromSuperlayer()

      // remove most current shape from the shape list //
      Shapes.removeLast()
   }
}

// erase all the shapes //
func EraseShapes()
{
   // loop over all the shapes //
   for Index in 0 ..< Shapes.count
   {
      // remove the shape from the super layer //
      Shapes[Index].removeFromSuperlayer()
   }

   // remove all the shapes from the shape list //
   Shapes.removeAll()
}

// override mouse down //
override func mouseDown(with Event:NSEvent)
{
   // if animating or not diagramming //
   if Animating || !Diagramming
   {
      // exit function since animating or not diagramming //
      return
   }

   // set that the current shape is valid //
   ValidShape = true

   // play the sound of a marker pen starting to write //
   Board.PlayGameSoundThread("marker")

   // get the location for the shape //
   GetLocation(Event)

   // create a shape layer //
   Shape = CAShapeLayer()

   // create a new mutable shape path //
   ShapePath = CGMutablePath()

   // set the shape line width //
   Shape.lineWidth = 4

   // set the shape line cap style to round //
   Shape.lineCap = "kCALineCapRound"

   // set the shape line join style to round //
   Shape.lineJoin = "kCALineJoinRound"

   // set the shape stroke color //
   Shape.strokeColor = Board.GalleryGridColor.cgColor

   // set the shape fill color //
   Shape.fillColor = NSColor.clear.cgColor

   // set the shape path starting point //
   ShapePath.move(to:CGPoint(x:Location.x, y:Location.y))

   // assign the shape path to the shape //
   Shape.path = ShapePath

   // append a new shape layer //
   Shapes.append(Shape)

   // add the new shape layer to the animation layer //
   AnimationLayer.addSublayer(Shape)
}

// override mouse dragged //
override func mouseDragged(with Event:NSEvent)
{
   // if animating or not diagramming or not a valid shape //
   if Animating || !Diagramming || !ValidShape
   {
      // return since animating or not diagramming or not a valid shape //
      return
   }

   // get the location for the shape //
   GetLocation(Event)

   // add the line to the shape path //
   ShapePath.addLine(to:CGPoint(x:Location.x, y:Location.y))

   // assign the shape path to the shape //
   Shapes[Shapes.count - 1].path = ShapePath
}

// override mouse up //
override func mouseUp(with Event:NSEvent)
{
   // if animating or not diagramming or not a valid shape //
   if Animating || !Diagramming || !ValidShape
   {
      // set that no longer a valid shape //
      ValidShape = false

      // return since not animating or not diagramming or not a valid shape //
      return
   }

   // get the location for the shape //
   GetLocation(Event)

   // add the line to the shape path //
   ShapePath.addLine(to:CGPoint(x:Location.x, y:Location.y))

   // assign the shape path to the shape //
   Shapes[Shapes.count - 1].path = ShapePath

   // set that no longer a valid shape //
   ValidShape = false
}

// create the gallery and grid images //
func CreateGalleryGridImage(_ IsGrid:Bool)
{
   // get whether is a grid view or not //
   Grid = IsGrid
```

```
    // set that not currently animating //
    Animating = false

    // set that not currently reviewing //
    Reviewing = false

    // set that not currently diagramming //
    Diagramming = false

    // set that currently not a valid shape //
    ValidShape = false

    // set the gallery grid buttons to true //
    Board.SetGalleryGridButtonsToState(true)

    // set that want to rasterize the animation layer //
    AnimationLayer.shouldRasterize = true

    // set the animation layer rasterization to 2 //
    AnimationLayer.rasterizationScale = 2

    // get the view height //
    Height = Double(self.frame.size.height)

    // get the view width //
    Width = Double(self.frame.size.width)

    // set the gallery width //
    GalleryWidth = max(FrameWidth * Double(Frames.count), Width)

    // set the grid width //
    GridWidth = max(GalleryWidth - Width, Width)

    // set the x initial location //
    X = CenterX

    // set the y location //
    Y = Height / 2.0

    // create the gallery image //
    GalleryImage = NSImage(size:NSSize(width:GalleryWidth, height:Double(FrameWidth)))

    // lock the gallery focus //
    GalleryImage.lockFocus()

    // loop over all the frames //
    for Index in 0 ..< Frames.count
    {
       // draw the board composite into the gallery image //
       BoardComposite.draw(in:NSMakeRect(CGFloat(FrameWidth * Double(Index)), 0,
                           CGFloat(FrameWidth), CGFloat(FrameWidth)))

       // draw the frame into the gallery image //
       Frames[Index].draw(in:NSMakeRect(CGFloat(FrameWidth * Double(Index)), 0,
                          CGFloat(FrameWidth), CGFloat(FrameWidth)))
    }

    // unlock the gallery focus //
    GalleryImage.unlockFocus()

    // if creating a grid //
    if Grid
    {
       // create the grid image //
       GridImage = NSImage(size:NSSize(width:GridWidth, height:Height))

       // lock the grid focus //
       GridImage.lockFocus()

       // draw the top grid row //
       GalleryImage.draw(in:NSMakeRect(0, CGFloat(Height - FrameWidth), CGFloat(GalleryWidth),
                         CGFloat(FrameWidth)))

       // draw the bottom grid row //
       GalleryImage.draw(in:NSMakeRect(CGFloat(-Width), 0, CGFloat(GalleryWidth),
                         CGFloat(FrameWidth)))

       // unlock the grid focus //
       GridImage.unlockFocus()

       // set the current image //
       Image = GridImage

       // get the x location center of the animation layer //
       CenterX = GridWidth / 2.0

       // set the maximum animation layer offset value //
       MaxOffset = GridWidth - Width

       // set the animation layer bounds //
       AnimationLayer.bounds = CGRect(x:0, y:0, width:Int(GridWidth), height:Int(Height))

       // set the animation layer contents //
       AnimationLayer.contents = Image
    }
    // is a gallery //
    else
    {
       // set the current image //
       Image = GalleryImage

       // get the x location center of the animation layer //
       CenterX = GalleryWidth / 2.0

       // set the maximum animation layer offset value //
       MaxOffset = GalleryWidth - Width

       // set the animation layer bounds //
       AnimationLayer.bounds = CGRect(x:0, y:0, width:Int(GalleryWidth), height:Int(Height))

       // set the animation layer contents //
       AnimationLayer.contents = Image
    }

    // set the x initial location //
    X = CenterX

    // set the y location //
    Y = Height / 2.0

    // set the animation delegate //
    Animation.delegate = self

    // add a tracking rectangle for the cursor //
    TrackingRectangle = self.addTrackingRect(self.bounds, owner:self, userData:nil, assumeInside:
                                             false)

    // disable the cursor rectangles //
    self.window!.disableCursorRects()
}
```

```
   // get the current slider value //
   func SliderValue() -> Double
   {
      // return the current slider value //
      return (CenterX - X) / MaxOffset
   }

   // set the animation layer position //
   func SetAnimationLayerPosition()
   {
      // remove all the animations from the animation layer //
      AnimationLayer.removeAllAnimations()

      // remove the animation effects //
      CATransaction.setDisableActions(true)

      // set the animation ending point //
      Animation.toValue = NSValue(point:NSPoint(x:X, y:Y))

      // update the animation layer position //
      AnimationLayer.position = CGPoint(x:CGFloat(X), y:CGFloat(Y))

      // set the animation duration //
      Animation.duration = min(abs(StartX - X) / FrameWidth * Duration, Duration)
   }

   // update the game gallery grid view //
   func UpdateGalleryGridView(_ TheFractionalOffset:Double)
   {
      // get the fractional offset from the game gallery grid slider //
      FractionalOffset = TheFractionalOffset

      // compute the x location value of the animation layer from the fractional offset //
      X = CenterX - FractionalOffset * MaxOffset

      // set the animation layer position //
      SetAnimationLayerPosition()
   }

   // animate moving a frame to the right or the left //
   func Animate(_ Right:Bool)
   {
      // set if going right or not //
      GoingRight = Right

      // if have reached the maximum or the minimum value of x and not reviewing //
      if (((X <= -MaxOffset + CenterX && GoingRight) || (X >= CenterX && !GoingRight)) &&
         !Reviewing) || (Grid && GridWidth == Width) || (!Grid && GalleryWidth == Width)
      {
         // exit function //
         return
      }

      // set that currently animating //
      Animating = true

      // update the gallery grid buttons //
      Board.SetGalleryGridButtonsToState(!Reviewing)

      // if have reached the minimum value of x and reviewing //
      if X <= -MaxOffset + CenterX && GoingRight && Reviewing
      {
         // set x to the initial value //
         X = CenterX

         // update the gallery grid slider value //
         Board.GalleryGridSlider.doubleValue = SliderValue()
      }

      // save the starting x value //
      StartX = X

      // set the animation starting point //
      Animation.fromValue = NSValue(point:NSPoint(x:X, y:Y))

      // if going right //
      if GoingRight
      {
         // decrement the x location value by one frame width //
         X -= Double(FrameWidth)

         // if went past the minimum x location value //
         if X <= CenterX - MaxOffset
         {
            // set x to the mimimum x location value //
            X = CenterX - MaxOffset
         }
      }
      // going left //
      else
      {
         // increment the x location value by one frame width //
         X += Double(FrameWidth)

         // if went past the maximum x location value //
         if X >= CenterX
         {
            // set x to the maximum x location value //
            X = CenterX
         }
      }

      // set the animation layer position //
      SetAnimationLayerPosition()

      // add the animation to the animation layer //
      AnimationLayer.add(Animation, forKey:"position")
   }
}

// the colors view //
class ColorsView:NSView
{
   // whether view has a border or not //
   var Bordered:Bool = false

   // whether in label colors view or not //
   var InLabelColors:Bool = false

   // whether in info colors view or not //
   var InInfoColors:Bool = false

   // whether in move list colors view or not //
   var InMoveListColors:Bool = false

   // whether in game list colors view or not //
   var InGameListColors:Bool = false

   // whether in the game clock border colors or not //
   var InGameClockColors:Bool = false
```

```
// whether in the game clock background colors or not //
var InGameClockBackgroundColors:Bool = false

// the current background color set //
var ColorSet:Int = 0

// the current color rectangle //
var CurrentColorRect:NSRect = NSRect()

// the colors //
var Colors:[NSColor] = []

// the current color //
var CurrentColor:Int = 0

// the width of a color bar //
var Width:Float = 0

// the current mouse location //
var Mouse:NSPoint = NSPoint()

// the chess board //
@IBOutlet var Board:GreenChess!

// update the current color //
func UpdateCurrentColor(_ Event:NSEvent)
{
   // get the current mouse location //
   Mouse = Event.locationInWindow

   // get the current color //
   GetCurrentColor()

   // if for label colors //
   if InLabelColors
   {
      // set board labels font color //
      Board.SetBoardLabelsFontColor()

      // recomposite the board with the updated labels layer //
      Board.RecompositeBoard(3, Span:1, Width:660)
   }
   // if for game info colors //
   else if InInfoColors
   {
      // set game info font color //
      Board.SetInfoFontColor()
   }
   // if for move list colors //
   else if InMoveListColors
   {
      // set the move list font color //
      Board.SetMoveListFontColor()
   }
   // if for game list colors //
   else if InGameListColors
   {
      // set the game list font color //
      Board.SetGameListFontColor()
   }
   // if for game clock border colors //
   else if InGameClockColors
   {
      // set the game clock border color //
      Board.SetGameClockColor()
   }
   // if for the game clock background colors //
   else if InGameClockBackgroundColors
   {
      // set the game clock background color //
      Board.SetGameClockBackgroundColor()
   }
   // in window colors //
   else
   {
      // set the current background color //
      Board.SetBackgroundColor(ColorSet)

      // recomposite the board with the updated color layer //
      Board.RecompositeBoard(0, Span:1, Width:660)
   }
}

// override mouse down //
override func mouseDown(with Event:NSEvent)
{
   // update the current color from the mouse location //
   UpdateCurrentColor(Event)
}

// override mouse dragged //
override func mouseDragged(with Event:NSEvent)
{
   // update the current color from the mouse location //
   UpdateCurrentColor(Event)
}

// get the current color //
func GetCurrentColor()
{
   // if not a color set that was not previously active //
   if !(ColorSet > 0 && !Bordered)
   {
      // get the current color from the mouse location //
      CurrentColor = Int(Float(Mouse.x - 30) / Width)
   }

   // if the current color is less than zero //
   if CurrentColor < 0
   {
      // set the current color to zero //
      CurrentColor = 0
   }
   // if the current color is greater than the maximum possible value //
   else if CurrentColor > Colors.count - 1
   {
      // set the current color to the maximum possible value //
      CurrentColor = Colors.count - 1
   }
}

// the draw rect function //
override func draw(_ Rect:NSRect)
{
   // loop over all the colors //
   for Index in 0 ..< Colors.count
   {
      // set the current fill color //
      Colors[Index].setFill()
```

```
         // draw the current color rectangle //
         NSBezierPath(rect:(NSMakeRect(CGFloat(Float(Index) * Width), 10, CGFloat(Width), 20)))
                     .fill()
      }

      // set the border color //
      NSColor(red:0, green:0, blue:0, alpha:0.4).set()

      // draw the colors spectrum border //
      NSBezierPath(rect:NSMakeRect(1, 10, self.frame.size.width - 2, 20)).stroke()

      // if the current color is zero //
      if CurrentColor == 0
      {
         // set the first current color rectangle //
         CurrentColorRect = NSMakeRect(1, 1, CGFloat(Width), 38)
      }
      // if the current color is the last color //
      else if CurrentColor == Colors.count - 1
      {
         // set the last current color rectangle //
         CurrentColorRect = NSMakeRect(CGFloat(Float(CurrentColor) * Width - 1), 1, CGFloat(Width),
                                       38)
      }
      // is between the first and last possible color //
      else
      {
         // set the current color rectangle //
         CurrentColorRect = NSMakeRect(CGFloat(Float(CurrentColor) * Width), 1, CGFloat(Width), 38)
      }

      // set the fill color to the current color //
      Colors[CurrentColor].setFill()

      // fill the current color rectangle with the current color //
      NSBezierPath(rect:CurrentColorRect).fill()

      // set the border color //
      NSColor(red:0, green:0, blue:0, alpha:0.4).set()

      // draw the current color rectangle border //
      NSBezierPath(rect:CurrentColorRect).stroke()

      // if have a border //
      if Bordered
      {
         // draw the view border //
         NSBezierPath(rect:self.bounds).stroke()
      }
   }

   // store the chosen color set and color //
   func StoreColorSet(_ TheColors:[NSColor], ChosenColor:Int)
   {
      // store the chosen color set //
      Colors = TheColors

      // set the current color //
      CurrentColor = ChosenColor

      // set the current color bar width //
      Width = Float(self.frame.size.width) / Float(Colors.count)

      // set that view needs display //
      self.needsDisplay = true

      // set to display view if needed //
      displayIfNeeded()
   }
}

// the game attributes class //
class GameAttributes:NSObject
{
   // the default fen string //
   let DefaultFENLayout:String = "rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR"

   // the shared user default preferences //
   var AppPreferences:UserDefaults = UserDefaults.standard

   // image data //
   var ImageData:Data = Data()

   // a position image //
   var PositionImage:NSImage = NSImage()

   // the move list string for the game //
   var MoveListString:String = ""

   // the annotation list string for the game //
   var AnnotationListString:String = ""

   // the fen layout string to use for the position editor //
   var FENLayout:String = ""

   // the starting move number for the game from the fen string //
   var FENStartMoveNumber:String = ""

   // the fen color to move string to use for the position editor //
   var FENColor:String = ""

   // the fen castling rights to use for the position editor //
   var FENCastlingRights:String = ""

   // the fen enpassant target square to use for the position editor //
   var FENEnpassantSquare:String = ""

   // the name of the person playing as white //
   var WhiteName:String = ""

   // the name of the person playing as black //
   var BlackName:String = ""

   // the location where the game was played //
   var SiteLocation:String = ""

   // the title of the tournament event //
   var EventTitle:String = ""

   // the round number of the game in the tournament //
   var RoundNumber:String = ""

   // the end result of the game //
   var EndResult:String = ""

   // the date the game was played on //
   var DateGamePlayed:String = ""

   // the time control used for the game //
   var TimeControl:String = ""
```

```
// the opening for the game //
var Opening:String = ""

// the section for the game //
var Section:String = ""

// the board number for the game //
var BoardNumber:String = ""

// the white elo for the game //
var WhiteElo:String = ""

// the black elo for the game //
var BlackElo:String = ""

// the white uscf for the game //
var WhiteUSCF:String = ""

// the black uscf for the game //
var BlackUSCF:String = ""

// the annotator for the game //
var Annotator:String = ""

// the eco for the game //
var ECO:String = ""

// initialize to empty attributes //
func Initialize()
{
   // initialize the move list string //
   MoveListString = ""

   // initialize the annotation list string //
   AnnotationListString = ""

   // initialize the fen layout string //
   FENLayout = DefaultFENLayout

   // initialize the fen color string //
   FENColor = "w"

   // initialize the fen castling rights string //
   FENCastlingRights = "KQkq"

   // initialize the fen enpassant square string //
   FENEnpassantSquare = "-"

   // initialize the fen start move number string //
   FENStartMoveNumber = "1"

   // initialize name of white player string //
   WhiteName = ""

   // initialize name of black player string //
   BlackName = ""

   // initialize site location string //
   SiteLocation = ""

   // initialize event title string //
   EventTitle = ""

   // initialize round number string //
   RoundNumber = ""

   // initialize end result string //
   EndResult = ""

   // initialize date game played string //
   DateGamePlayed = ""

   // initialize time control string //
   TimeControl = ""

   // initialize the opening string //
   Opening = ""

   // initialize the section string //
   Section = ""

   // initialize the board number string //
   BoardNumber = ""

   // initialize white elo string //
   WhiteElo = ""

   // initialize black elo string //
   BlackElo = ""

   // initialize white uscf string //
   WhiteUSCF = ""

   // initialize black uscf string //
   BlackUSCF = ""

   // initialize annotator string //
   Annotator = ""

   // initialize eco string //
   ECO = ""

   // initialize the position image //
   PositionImage = NSImage()
}

// initialize the game attributes for the given tag //
func Initialize(_ Tag:String)
{
   // retrieve the move list string if was stored //
   MoveListString = PreferenceString("MoveListString" + Tag, Default:"")

   // retrieve the annotation list string if was stored //
   AnnotationListString = PreferenceString("AnnotationListString" + Tag, Default:"")

   // retrieve the fen layout if was stored //
   FENLayout = PreferenceString("FENLayout" + Tag, Default:DefaultFENLayout)

   // retrieve the fen color if was stored //
   FENColor = PreferenceString("FENColor" + Tag, Default:"w")

   // retrieve the fen castling rights if was stored //
   FENCastlingRights = PreferenceString("FENCastlingRights" + Tag, Default:"KQkq")

   // retrieve the fen enpassant square if was stored //
   FENEnpassantSquare = PreferenceString("FENEnpassantSquare" + Tag, Default:"-")

   // retrieve the fen start move number if was saved //
   FENStartMoveNumber = PreferenceString("FENStartMoveNumber" + Tag, Default:"1")

   // retrieve name of white player if was saved //
   WhiteName = PreferenceString("WhiteName" + Tag, Default:"")
```

```
// retrieve name of black player if was saved //
BlackName = PreferenceString("BlackName" + Tag, Default:"")

// retrieve site location if was saved //
SiteLocation = PreferenceString("SiteLocation" + Tag, Default:"")

// retrieve event title if was saved //
EventTitle = PreferenceString("EventTitle" + Tag, Default:"")

// retrieve round number if was saved //
RoundNumber = PreferenceString("RoundNumber" + Tag, Default:"")

// retrieve end result if was saved //
EndResult = PreferenceString("EndResult" + Tag, Default:"")

// retrieve date game played if was saved //
DateGamePlayed = PreferenceString("DateGamePlayed" + Tag, Default:"")

// retrieve time control if was saved //
TimeControl = PreferenceString("TimeControl" + Tag, Default:"")

// retrieve the opening if was saved //
Opening = PreferenceString("Opening" + Tag, Default:"")

// retrieve the section if was saved //
Section = PreferenceString("Section" + Tag, Default:"")

// retrieve the board number if was saved //
BoardNumber = PreferenceString("BoardNumber" + Tag, Default:"")

// retrieve white elo if was saved //
WhiteElo = PreferenceString("WhiteElo" + Tag, Default:"")

// retrieve black elo if was saved //
BlackElo = PreferenceString("BlackElo" + Tag, Default:"")

// retrieve white uscf if was saved //
WhiteUSCF = PreferenceString("WhiteUSCF" + Tag, Default:"")

// retrieve black uscf if was saved //
BlackUSCF = PreferenceString("BlackUSCF" + Tag, Default:"")
```

```
      BlackUSCF = PreferenceString("BlackUSCF" + Tag, Default:"")

      // retrieve annotator if was saved //
      Annotator = PreferenceString("Annotator" + Tag, Default:"")

      // retrieve eco if was saved //
      ECO = PreferenceString("ECO" + Tag, Default:"")

      // retrieve the position image //
      PositionImage = PreferenceImage("PositionImage" + Tag, Default:NSImage())
   }

   // store the autosave data //
   func StoreAutosaveData(_ Tag:String)
   {
      // store the move list to preferences //
      AppPreferences.set(MoveListString, forKey:"MoveListString" + Tag)

      // store the annotation list to preferences //
      AppPreferences.set(AnnotationListString, forKey:"AnnotationListString" + Tag)

      // store the fen layout to preferences //
      AppPreferences.set(FENLayout, forKey:"FENLayout" + Tag)

      // store the fen color to preferences //
      AppPreferences.set(FENColor, forKey:"FENColor" + Tag)

      // store the fen castling rights to preferences //
      AppPreferences.set(FENCastlingRights, forKey:"FENCastlingRights" + Tag)

      // store the fen enpassant square to preferences //
      AppPreferences.set(FENEnpassantSquare, forKey:"FENEnpassantSquare" + Tag)

      // store the fen start move number to preferences //
      AppPreferences.set(FENStartMoveNumber, forKey:"FENStartMoveNumber" + Tag)

      // store the white player name to preferences //
      AppPreferences.set(WhiteName, forKey:"WhiteName" + Tag)

      // store the black player name to preferences //
      AppPreferences.set(BlackName, forKey:"BlackName" + Tag)

      // store the site location to preferences //
      AppPreferences.set(SiteLocation, forKey:"SiteLocation" + Tag)

      // store the event title to preferences //
      AppPreferences.set(EventTitle, forKey:"EventTitle" + Tag)

      // store the round number to preferences //
      AppPreferences.set(RoundNumber, forKey:"RoundNumber" + Tag)

      // store the end result to preferences //
      AppPreferences.set(EndResult, forKey:"EndResult" + Tag)

      // store the date game played to preferences //
      AppPreferences.set(DateGamePlayed, forKey:"DateGamePlayed" + Tag)

      // store the time control to preferences //
      AppPreferences.set(TimeControl, forKey:"TimeControl" + Tag)

      // store the opening to preferences //
      AppPreferences.set(Opening, forKey:"Opening" + Tag)

      // store the section to preferences //
      AppPreferences.set(Section, forKey:"Section" + Tag)

      // store the board number to preferences //
      AppPreferences.set(BoardNumber, forKey:"BoardNumber" + Tag)

      // store the white elo to preferences //
      AppPreferences.set(WhiteElo, forKey:"WhiteElo" + Tag)

      // store the black elo to preferences //
      AppPreferences.set(BlackElo, forKey:"BlackElo" + Tag)

      // store the white uscf to preferences //
      AppPreferences.set(WhiteUSCF, forKey:"WhiteUSCF" + Tag)

      // store the black uscf to preferences //
      AppPreferences.set(BlackUSCF, forKey:"BlackUSCF" + Tag)

      // store the annotator to preferences //
      AppPreferences.set(Annotator, forKey:"Annotator" + Tag)

      // store the eco to preferences //
      AppPreferences.set(ECO, forKey:"ECO" + Tag)

      // save the position image to preferences //
      AppPreferences.set(NSKeyedArchiver.archivedData(withRootObject:PositionImage), forKey:
                         "PositionImage" + Tag)
   }

   // get the user preference for the specified key //
   func PreferenceImage(_ Key:String, Default:NSImage) -> NSImage
   {
      // if the value for the key already exists //
      if AppPreferences.bool(forKey:Key + "Exists")
      {
         // if the data to unarchive exists //
         if NSKeyedUnarchiver.unarchiveObject(with:AppPreferences.object(forKey:Key) as! Data) !=
                                              nil
         {
            // return the saved image //
            return NSKeyedUnarchiver.unarchiveObject(with:AppPreferences.object(forKey:Key) as!
                                                     Data) as! NSImage
         }
         // the data to unarchive doesn't exist //
         else
         {
            // return the default image value //
            return Default
         }
      }
      // value for the key doesn't exist yet //
      else
      {
         // set that the value for the key currently exists now //
         AppPreferences.set(true, forKey:Key + "Exists")

         // get the image data for the default image //
         ImageData = NSKeyedArchiver.archivedData(withRootObject:Default)

         // save the user preference for the image key //
         AppPreferences.set(ImageData, forKey:Key)

         // return the defalt image //
         return Default
      }
   }

   // get the user saved preference for this key //
   func PreferenceString(_ Key:String, Default:String) -> String
   {
      // if the user saved preference exists //
      if AppPreferences.bool(forKey:Key + "Exists")
      {
         // return the user saved preference for this key //
         return AppPreferences.string(forKey:Key)!
      }
      // the user saved preference does not yet exist //
      else
      {
         // set that the user preference now exists for this key //
         AppPreferences.set(true, forKey:Key + "Exists")

         // set the default value for this key //
         AppPreferences.set(Default, forKey:Key)

         // return the default value for this key //
         return Default
      }
   }
}

// the greenchess implementation //
class GreenChess:NSView, NSTableViewDelegate, CAAnimationDelegate, NSWindowDelegate,
                 NSApplicationDelegate
{
   // the window scale //
   var WindowScale:CGFloat = 0

   // an illustration piece type //
   var IllustrationPieceType:Int = 0

   // an illustration piece image //
   var IllustrationPiece:NSImage = NSImage()

   // the rotation with perspective transform identity matrix //
   var RotationWithPerspective:CATransform3D = CATransform3DIdentity

   // if the game explorer window is open //
   var GameExplorerWindowOpen:Bool = false

   // if found any invalid moves //
   var FoundInvalidMoves:Bool = false

   // whether should close the main window or not //
   var ShouldCloseMainWindow:Bool = false

   // whether have stockfish analysis to show or not //
   var StockfishAnalysisToDisplay:Bool = false

   // the illustration size for the move list //
   var IllustrationSize:CGFloat = 180

   // the illustration square size //
   var IllustrationSquareSize:CGFloat = 0

   // the illustration piece layer offset to account for the border //
   var IllustrationOffset:CGFloat = 0

   // the core graphics context for illustrations //
   var IllustrationContext:CGContext? = nil

   // the table row //
   var TableRow:Int = 0

   // a table cell view //
   var TableCell:NSTableCellView = NSTableCellView()

   // whether already called awake from nib //
   var Awoke:Bool = false

   // the previous board layer that had changed //
   var PreviousLayer:Int = -1

   // the previous span of the board layers that had changed //
   var PreviousSpan:Int = 1

   // the bottom board layer for the board composite //
   var BottomBoardLayer:NSImage = NSImage()

   // the top board layer for the board composite //
   var TopBoardLayer:NSImage = NSImage()

   // the pieces layer square size //
   var PiecesLayerSquareSize:CGFloat = 0

   // the pieces layer offset due to the board border //
   var PiecesLayerOffset:CGFloat = 0

   // number of plies //
   var Plies:Int = 0

   // pgn representation of the current game //
   var PGN:String = ""

   // the pasteboard //
   var Pasteboard:NSPasteboard = NSPasteboard.general

   // whether moving the piece by selecting squares //
   var MoveBySelectingSquares:Bool = false

   // the number of selected squares from the user //
   var SelectedSquares:Int = 0

   // the progress activity circle //
   var ProgressCircle:CALayer = CALayer()

   // the annotation board layer for the left //
   var AnnotationBoardLayerLeft:CALayer = CALayer()

   // the annotation board layer for the middle //
   var AnnotationBoardLayerMiddle:CALayer = CALayer()

   // the annotation board layer for the right //
   var AnnotationBoardLayerRight:CALayer = CALayer()

   // the annotation pieces layer for the left //
   var AnnotationPiecesLayerLeft:CALayer = CALayer()

   // the annotation pieces layer for the middle //
   var AnnotationPiecesLayerMiddle:CALayer = CALayer()

   // the annotation pieces layer for the right //
   var AnnotationPiecesLayerRight:CALayer = CALayer()

   // whether animating progress or not //
   var AnimateProgress:Bool = false

   // the animation progress direction //
   var ProgressDirectionUp:Bool = true

   // the array of board frames //
   var Frames:[NSImage] = []

   // the annotation frames //
   var AnnotationFrames:[NSImage] = []

   // the illustrations for the pgn moves list //
   var Illustrations:[NSImage] = []

   // the current window color //
   var CurrentWindowColor:NSColor = NSColor()

   // the window background pattern //
   var WindowBackgroundPattern:NSImage = NSImage()

   // the window background overlay //
   var WindowBackgroundOverlay:NSImage = NSImage()

   // the board composite image //
   var BoardComposite:NSImage = NSImage()

   // the sized board composite image //
   var SizedBoardComposite:NSImage = NSImage()

   // the illustration sized board composite //
   var IllustrationSizedBoardComposite:CGImage!

   // the pieces layer //
   var PiecesLayer:NSImage = NSImage()

   // the board layer opacities //
   var BoardLayerOpacities:[Float] = []

   // the board layers //
   var BoardLayers:[NSImage] = []

   // the square roughness circles //
   var RoughnessCircles:NSBezierPath = NSBezierPath()

   // set that not initially a valid shape //
   var ValidShape:Bool = false

   // the shape layers //
   var Shapes:[CAShapeLayer] = []

   // a shape layer //
   var Shape:CAShapeLayer = CAShapeLayer()

   // a shape path //
   var ShapePath:CGMutablePath = CGMutablePath()

   // the board roughness x index //
   var RoughnessXIndex:Int = 0

   // the board roughness y index //
   var RoughnessYIndex:Int = 0

   // the light square roughness image //
   var SquareRoughnessLight:NSImage = NSImage()

   // the square border image //
   var SquareBorderImage:NSImage = NSImage()

   // the board tint white image //
   var BoardTintWhiteImage:NSImage = NSImage()

   // the board tint black image //
   var BoardTintBlackImage:NSImage = NSImage()

   // the dark square roughness image //
   var SquareRoughnessDark:NSImage = NSImage()

   // a board layer image //
   var LayerImage:NSImage = NSImage()

   // the saved position slot //
   var SavedPositionSlot:Int = 0

   // the position list index //
   var PositionListIndex:Int = 0

   // the game attributes //
   var Attributes:GameAttributes = GameAttributes()

   // whether to start the game clock or not //
   var StartClock = false

   // whether to hide the game clock in the panel or not //
   var HideGameClockInPanel:Bool = false

   // whether to hide the game clock in the panel or not //
   var HideClock:CGFloat = 0

   // whether to hide the captured pieces in the panel or not //
   var HideCapturedPiecesInPanel:Bool = false

   // whether to hide the captured pieces in the panel or not //
   var HidePieces:CGFloat = 0

   // the minimum window size //
   var WindowMinSize:NSSize = NSSize()

   // the maximum window size //
   var WindowMaxSize:NSSize = NSSize()

   // whether in fullscreen mode or not //
   var FullScreen:Bool = false

   // the current preferences selection //
   var PreferencesSelection:Int = 20

   // the previously attached sheet if there was one //
   var PreviouslyAttachedSheet:NSWindow? = nil

   // whether in a window panel or not //
   var InPanel:Bool = false

   // whether in the second position editor menu //
   var SecondPositionMenu:Bool = false

   // the position editor cursor //
   var PosCursor:NSCursor = NSCursor()

   // the diagram cursor //
   var DiagramCursor:NSCursor = NSCursor()

   // the position editor moving piece //
   var PositionMovingPiece:Int = 0

   // the snapshot file name //
   var SnapshotFilename:String = ""

   // the number of gallery grid movie frames //
   var GalleryGridMovieFrames:Int = 0

   // the gallery grid view title //
   var GalleryGridTitle:String = ""

   // whether in the enter fen window or not //
   var InEnterFENWindow:Bool = false

   // the entered fen string //
   var EnteredFENString:String = ""

   // the default fen layout //
   let DefaultFENLayout:String = "rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR"

   // whether the fen string was valid or not //
   var ValidFENString:Bool = true

   // the entered fen string array //
   var EnteredFENStringArray:[String] = []

   // the raw entered fen string array //
   var RawEnteredFENStringArray:[String] = []

   // the window color scaling value //
   var ColorScaling:CGFloat = 0

   // whether font is bold or not //
   var FontBoldString:String = ""

   // the board label font size //
   var BoardLabelSize:CGFloat = 20

   // the file label offset //
   var FileLabelOffset:CGFloat = 0

   // the rank label offset //
   var RankLabelOffset:CGFloat = 0

   // the board labels attributes //
   var BoardLabelsAttributes:NSDictionary = [:]

   // the board labels font type //
   var BoardLabelsFontType:Int = 0

   // the move list font type //
   var MoveListFontType:Int = 0

   // the game info font type //
   var GameInfoFontType:Int = 0

   // the game list font type //
   var GameListFontType:Int = 0

   // the board labels bold setting //
   var BoardLabelsBold:Bool = false

   // the move list bold setting //
   var MoveListBold:Bool = false

   // the game info bold setting //
   var GameInfoBold:Bool = false

   // the game list bold setting //
   var GameListBold:Bool = false

   // the board labels font sizes setting //
   var BoardLabelsSizes:Int = 0

   // the move list font sizes setting //
   var MoveListSizes:Int = 0

   // the game info font sizes setting //
   var GameInfoSizes:Int = 0

   // the game list sizes setting //
   var GameListSizes:Int = 0

   // the possible fonts //
   var Fonts:[String] = ["American Typewriter", "Arial", "Arial Narrow", "Avenir Next",
                         "Avenir Next Condensed", "Baskerville", "Bodoni 72", "Chalkboard",
                         "Chalkboard SE", "Cochin", "Comic Sans MS", "Copperplate", "Courier",
                         "Courier New", "Didot", "Georgia", "Gill Sans", "Helvetica",
                         "Helvetica Neue", "Iowan Old Style", "Lucida Grande", "Menlo",
                         "Noteworthy", "Optima", "Palatino", "PT Mono", "PT Serif", "Skia",
                         "Snell Roundhand", "Times", "Times New Roman", "Trebuchet MS", "Verdana"]

   // the board label font //
   var BoardLabelFont:NSFont = NSFont()

   // initialize the previously selected row in the game list table view //
   var PreviouslySelectedRow:Int = 0

   // the pixel buffer //
   var PixelBuffer:CVPixelBuffer? = nil

   // the movie frame index //
   var FrameIndex:Int = 0

   // the av asset writer adaptor //
   var Adaptor:AVAssetWriterInputPixelBufferAdaptor? = nil

   // the video input //
   var VidInput:AVAssetWriterInput? = nil

   // the video writer //
   var VideoWriter:AVAssetWriter? = nil

   // the current label color index //
   var LabelColor:Int = 0

   // the current info color index //
   var InfoColor:Int = 0

   // the current move list color index //
   var MoveListColor:Int = 0

   // the current game list color index //
   var GameListColor:Int = 0

   // the game clock border color index //
   var GameClockColor:Int = 0

   // the board tint color //
   var BoardTintColor:Int = 0

   // the game clock background color index //
   var GameClockBackgroundColor:Int = 0

   // the previous main window title //
   var PreviousMainWindowTitle:String = ""

   // the board label index //
   var LabelIndex:Int = 0

   // a file label //
   var FileLabel:NSAttributedString = NSAttributedString()

   // a rank label //
   var RankLabel:NSAttributedString = NSAttributedString()

   // the total number of threads available //
   var TotalThreads:Int = ProcessInfo.processInfo.processorCount

   // the number of threads to use for the uci engine //
   var ThreadsToUse:Int = 0

   // the total number of pieces on the board //
   var TotalPieces:Int = 0

   // the timed game opponent index //
   var TimedGameOpponent:Int = 0

   // the timed game color index //
   var TimedGameColor:Int = 0

   // the timed game time control index //
   var TimedGameTime:Int = 0

   // number of left parentheses in a move annotation //
   var LeftParentheses:Int = 0

   // number of right parentheses in a move annotation //
   var RightParentheses:Int = 0

   // the move offset for printing game info if had black move first in fen notation //
   var MoveOffset:Int = 0

   // the size of the square on the board //
   var SquareSize:Int = 75

   // the size of the square offset on the board //
   var SquareOffset:Int = 0

   // the current move location in the game annotator //
   var GameAnnotatorMoveLocation:Int = 0

   // the current rate of animation for a moving piece //
   var AnimationRate:Int = 0

   // the gallery grid animation rate //
   var GalleryGridAnimationRate:Int = 0

   // the chess board border visibility //
   var BorderVisibility:Int = 0

   // the board tint visibility //
   var BoardTintVisibility:Int = 0

   // the window texture opacity //
   var WindowTextureOpacity:Int = 0

   // the board roughness //
   var BoardRoughness:Int = 0

   // the square border value //
   var SquareBorder:Int = 0

   // the game clock background visibility //
   var ClockBackgroundVisibility:Int = 0

   // the chess board border texture //
   var BorderTexture:Int = 0

   // the current uci engine difficulty setting //
   var UCIEngineDifficulty:Int = 0

   // the current uci engine hash size //
   var UCIEngineHash:Int = 0

   // the current uci engine riskiness //
   var UCIEngineRiskiness:Int = 0

   // the orginal animation rate before a timed game //
   var OriginalAnimationRate:Int = 0

   // an index in a list //
   var Index:Int = 0

   // an index in a list //
   var ListIndex:Int = 0

   // an index in a list //
   var OpeningBookIndex:Int = 0

   // an index in a list //
   var MoveListCountIndex:Int = 0

   // the list index for the autosave game data //
   var AutosaveIndex:Int = 0

   // the chess piece image index //
   var ChessPieceImageIndex:Int = 0

   // number of values counted //
   var Count:Int = 0

   // a rank index in a list //
   var Rank:Int = 0

   // a file index in a list //
   var File:Int = 0

   // a line index in a list //
   var Line:Int = 0

   // the file of the enpassant square //
   var EnpassantSquareFile:Int = 0

   // the rank of the enpassant square //
   var EnpassantSquareRank:Int = 0

   // an index in a move list //
   var MoveIndex:Int = 0

   // the rainbow background colors //
   var RainbowBackgroundColors:[NSColor] = []

   // the blue background colors //
   var BlueBackgroundColors:[NSColor] = []

   // the green background colors //
   var GreenBackgroundColors:[NSColor] = []

   // the yellow background colors //
   var YellowBackgroundColors:[NSColor] = []

   // the gray background colors //
   var GrayBackgroundColors:[NSColor] = []

   // the brown background colors //
   var BrownBackgroundColors:[NSColor] = []

   // the teal background colors //
   var TealBackgroundColors:[NSColor] = []

   // the font colors //
   var FontColors:[NSColor] = []

   // the game clock colors //
   var GameClockColors:[NSColor] = []

   // the board tint colors //
   var BoardTintColors:[NSColor] = []

   // the chess board border colors //
   var BorderColors:[NSColor] = []

   // the list of captured white pieces to display //
   var CapturedWhitePieces:[Int] = [0, 0, 0, 0, 0]

   // the list of captured black pieces to display //
   var CapturedBlackPieces:[Int] = [0, 0, 0, 0, 0]

   // the list of pieces in the fen layout from the position editor //
   var PiecesInFENLayout:[Int] = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

   // the max number of pieces that could have been promoted by white //
   var AllowedWhitePromotedPieces:Int = 0

   // the max number of pieces that could have been promoted by black //
   var AllowedBlackPromotedPieces:Int = 0

   // the number of pieces that were possibly promoted by white //
   var PromotedWhitePieces:Int = 0

   // the number of pieces that were possibly promoted by black //
   var PromotedBlackPieces:Int = 0

   // the number of plies made //
   var PlyCount:Int = 0

   // maximum time for the chess engine to search for a move //
   var SearchTime:Int = 0

   // the chessboard type to display //
   var BoardType:Int = 0

   // the current background color set //
   var BackgroundColorSet:Int = 0

   // the current rainbow background color //
   var RainbowBackgroundColor:Int = 0

   // the current blue background color //
   var BlueBackgroundColor:Int = 0

   // the current brown background color //
   var BrownBackgroundColor:Int = 0

   // the current green background color //
   var GreenBackgroundColor:Int = 0

   // the current teal background color //
   var TealBackgroundColor:Int = 0

   // the current yellow background color //
   var YellowBackgroundColor:Int = 0

   // the current gray background color //
   var GrayBackgroundColor:Int = 0

   // the chess piece set type //
   var ChessPieceSetType:Int = 0

   // the type of piece that is currently moving on the board //
   var MovingPiece:Int = 0

   // the blank layout of the board //
   var BlankLayout:[[Int]] = [[0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0],
                              [0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0],
                              [0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0],
                              [0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0]]

   // the layout of the chess pieces on the chessboard //
   var Layout:[[Int]] = [[0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0],
                         [0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0],
                         [0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0],
                         [0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0]]

   // the saved layout of the chess pieces on the chessboard //
   var SavedLayout:[[Int]] = [[0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0],
                              [0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0],
                              [0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0],
                              [0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0]]

   // the type of piece a pawn is promoting to if had a pawn promotion //
   var PromotionType:Int = 0

   // the length of the list to loop over //
   var ListEnd:Int = 0

   // type of piece //
   var PieceType:Int = 0

   // the game sound selection //
   var GameSoundSelection:Int = 0

   // the current game state which is used in the timed game update clock function //
   var CurrentGameState:Int = 0

   // the game data for the current chess game //
   var GameData:UnsafeMutableRawPointer = DataNew()

   // the saved game data for the current chess game //
   var SavedGameData:UnsafeMutableRawPointer = DataNew()

   // the saved position images //
   var SavedPositionImages:[NSImage] = []

   // a chess board used for engine move calculations //
   var Board:UnsafeMutableRawPointer = BoardNew()

   // a chess board used for extracting pgn notation //
   var GameDataBoard:UnsafeMutableRawPointer = BoardNew()

   // a chess board used for reading in the opening book //
   var OpeningBoard:UnsafeMutableRawPointer = BoardNew()

   // a chess board used to see if have any threatened pieces or not //
   var ThreatenedBoard:UnsafeMutableRawPointer = BoardNew()

   // a chess board to see whether king was illegally captured or not //
   var KingCapturedTestBoard:UnsafeMutableRawPointer = BoardNew()

   // a chess engine //
   var Engine:UnsafeMutableRawPointer = EngineNew()

   // a list of pgn games //
   var GameList:UnsafeMutableRawPointer = DatabaseNew()

   // a list of pgn games for the chess opening book //
   var OpeningBookGameList:UnsafeMutableRawPointer = DatabaseNew()

   // a chess opening book //
   var OpeningBook:UnsafeMutableRawPointer = OpeningBookNew()

   // whether currently recording a movie of a game or not //
   var RecordingMovie:Bool = false

   // whether playing a match of uci engine against itself //
   var PlayingEngineMatch:Bool = false

   // whether core animating or not //
   var CoreAnimating:Bool = false

   // whether to highlight which pieces can be captured //
   var HighlightCapturedPieces:Bool = false

   // whether to highlight which pieces are threatened //
   var HighlightThreatenedPieces:Bool = false

   // whether to multithread the uci engine or not //
   var MultithreadUCIEngine:Bool = false

   // whether to highlight the last move made during a game being played //
   var HighlightLastMoveMade:Bool = false

   // whether originally using an opening book before the timed game //
   var OriginalUseEcoBook:Bool = false

   // whether use the encyclopedia of chess openings or not //
   var UseEcoBook:Bool = false

   // whether want to autosave the game data or not //
   var AutosaveGame:Bool = false

   // whether want to autoload the stockfish engine or not //
   var AutoloadStockfish:Bool = false

   // whether to show the stockfish analysis or not //
   var ShowStockfishAnalysis:Bool = false

   // whether have a valid FEN enpassant target square or not //
   var ValidFENEnpassantSquare:Bool = false

   // whether the Forsyth-Edwards board layout is legal or not //
   var LegalFENLayout:Bool = false

   // whether a move read in from a game file exists or not //
   var MoveExists:Bool = false

   // whether white won the game or not //
   var WhiteWonGame:Bool = false

   // whether black won the game or not //
   var BlackWonGame:Bool = false

   // whether the game was drawn or not //
   var DrawnGame:Bool = false

   // whether editing nonempty board position or not //
   var EditingNonemptyPosition:Bool = false

   // whether currently editing the board position or not //
   var InPositionEditor:Bool = false

   // whether currently in the database explorer or not //
   var InDatabaseExplorer:Bool = false

   // whether currently entering in game info or not //
   var EnteringGameInfo:Bool = false

   // whether in the about window or not //
   var InAboutWindow:Bool = false

   // whether loading a saved position or not //
   var LoadingSavedPosition:Bool = false

   // whether querying to save a position //
   var QuerySavePosition:Bool = false

   // whether querying to load a position that doesn't exist //
   var QueryLoadPosition:Bool = false

   // initialize load position index to zero //
   var LoadPositionIndex:Int = 0

   // whether currently searching the game list or not //
   var SearchingGameList:Bool = false

   // whether have read in the game list yet //
   var ReadInGameList:Bool = false

   // whether currently entering in preferences or not //
   var EnteringPreferences:Bool = false

   // whether in the game gallery or not //
   var InGameGallery:Bool = false

   // whether in the game grid or not //
   var InGameGrid:Bool = false

   // whether creating the game gallery grid or not //
   var CreatingGalleryGrid:Bool = false

   // whether currently querying what type pawn will promote to //
   var QueryPawnPromotion:Bool = false

   // whether or not uci engine has been loaded successfully //
   var UCIEngineLoaded:Bool = false

   // whether currently querying if want to start a new game or not //
   var QueryStartNewGame:Bool = false

   // query whether the uci engine was loaded //
   var QueryUCIEngineLoaded:Bool = false

   // whether currently querying user that timed game is over //
   var QueryTimedGameOver:Bool = false

   // whether currently querying if want to open a game file or not //
   var QueryOpenGameFile:Bool = false

   // query whether want to save the edited position from the position editor //
   var QuerySaveEditedPosition:Bool = false

   // whether currently querying if want to quit the application or not //
   var QueryQuitApplication:Bool = false

   // whether an illegal move is being undone or not //
   var UndoingIllegalMove:Bool = false

   // whether the engine is thinking or not //
   var EngineThinking:Bool = false

   // whether the user wants the board to be rotated or not //
   var RotateBoard:Bool = false

   // whether want to hide the game info panel or not //
   var HideInfoPanel:Bool = false

   // whether want a small info panel or not //
   var SmallInfoPanel:Bool = false

   // whether want a large info panel or not //
   var LargeInfoPanel:Bool = false

   // whether the user has chosen to play against another human or not //
   var HumanOpponent:Bool = false

   // whether playing against the built-in greenchess engine or not //
   var EngineOpponent:Bool = false

   // whether playing against uci chess engine or not //
```

```
var UCIEngineOpponent:Bool = false

// whether the piece is a valid piece to be moved //
var ValidPiece:Bool = false

// whether diagramming a chessboard or not //
var Diagramming:Bool = false

// whether the game sound is turned off or not //
var SoundOff:Bool = false

// whether user is currently reviewing the game //
var ReviewingGame:Bool = false

// whether want to pause a game or not //
var PauseGameReview:Bool = false

// whether currently exploring the game or not //
var ExploringGame:Bool = false

// whether currently annotating the game or not //
var AnnotatingGame:Bool = false

// whether currently playing a timed game or not //
var PlayingTimedGame:Bool = false

// whether currently entering timed game info //
var EnteringTimedGameInfo:Bool = false

// whether entering fen notation to convert //
var EnteringFEN:Bool = false

// whether currently analyzing the game or not //
var AnalyzingGame:Bool = false

// whether was currently analyzing the game or not //
var WasAnalyzingGame:Bool = false

// whether will save the edited board position or not //
var WillSaveEditedPosition:Bool = false

// the white kingside castling rights for the position editor //
var WhiteKingsideCastle:Bool = false

// the white queenside castling rights for the position editor //
var WhiteQueensideCastle:Bool = false

// the black kingside castling rights for the position editor //
var BlackKingsideCastle:Bool = false

// the black queenside castling rights for the position editor //
var BlackQueensideCastle:Bool = false

// the timed game time offset relative to the six o'clock position //
var TimedGameOffset:Int = 0

// the remaining time for white //
var RemainingWhiteTime:Int = 0

// the remaining time for black //
var RemainingBlackTime:Int = 0

// the elapsed game time for white //
var ElapsedWhiteTime:Int = 0

// the elapsed game time for black //
var ElapsedBlackTime:Int = 0

// the total time for the timed game //
var TotalGameTime:Int = 0

// the time left in the timed game //
var TimeLeft:Int = 0

// the volume for the game sound to play //
var GameSoundVolume:Float = 0

// the chess engine task for engine //
var EngineTask:Process = Process()

// the chess engine write pipe for engine //
var EnginePipeWrite:Pipe = Pipe()

// the chess engine read pipe for engine //
var EnginePipeRead:Pipe = Pipe()

// the data to write to engine //
var EngineDataToWrite:Data = Data()

// image data for the saved position images //
var ImageData:Data = Data()

// the animation layer //
var AnimationLayer:CALayer = CALayer()

// the moving piece layer //
var MovingPieceLayer:CALayer = CALayer()

// the position layers //
var PositionLayers:[CALayer] = [CALayer(), CALayer()]

// a gui alert panel //
var Alert:NSAlert = NSAlert()

// messages from chess engine //
var EngineMessages:[String] = []

// the light chess square names //
var ChessLightSquareNames:[String] = ["light_bamboo", "light_white", "light_white",
                                      "light_chalk", "light_cloth", "light_felt", "light_glass",
                                      "light_gold", "light_granite", "light_white",
                                      "light_white", "light_leather", "light_marble",
                                      "light_plastic", "light_rust", "light_stone",
                                      "light_silver", "light_white", "light_wood"]

// the dark chess square names //
var ChessDarkSquareNames:[String] = ["dark_bamboo", "dark_blue", "dark_brown", "dark_chalk",
                                     "dark_cloth", "dark_felt", "dark_glass", "dark_gold",
                                     "dark_granite", "dark_gray", "dark_green", "dark_leather",
                                     "dark_marble", "dark_plastic", "dark_rust", "dark_stone",
                                     "dark_silver", "dark_teal", "dark_wood"]

// array of all the chess piece names //
var ChessPieceNames:[String] = ["whitepawn1", "whiteknight1", "whitebishop1", "whiterook1",
                                "whitequeen1", "whiteking1", "blackpawn1", "blackknight1",
                                "blackbishop1", "blackrook1", "blackqueen1", "blackking1",
                                "whitepawn2", "whiteknight2", "whitebishop2", "whiterook2",
                                "whitequeen2", "whiteking2", "blackpawn2", "blackknight2",
                                "blackbishop2", "blackrook2", "blackqueen2", "blackking2",
                                "whitepawn3", "whiteknight3", "whitebishop3", "whiterook3",
                                "whitequeen3", "whiteking3", "blackpawn3", "blackknight3",
                                "blackbishop3", "blackrook3", "blackqueen3", "blackking3",
                                "whitepawn4", "whiteknight4", "whitebishop4", "whiterook4",
                                "whitequeen4", "whiteking4", "blackpawn4", "blackknight4",
                                "blackbishop4", "blackrook4", "blackqueen4", "blackking4"]

// the chess piece set names //
var ChessPieceSetNames:[String] = ["Standard", "3D", "3D Textured", "Figurine"]

// the chess board names //
var ChessBoardNames:[String] = ["Bamboo", "Blue", "Brown", "Chalk", "Cloth", "Felt", "Glass",
                                "Gold", "Granite", "Gray", "Green", "Leather", "Marble",
                                "Plastic", "Rust", "Sandstone", "Silver", "Teal", "Wood"]

// the game sound names //
var GameSoundNames:[String] = ["sound_thump", "sound_wrongsquare", "sound_pop", "sound_click",
                               "sound_clear", "sound_warning", "sound_switch",
                               "sound_buttonclick", "sound_camera", "sound_marker",
                               "sound_movie"]

// rank symbols from 1 to 8 //
var RankSymbols:[String] = ["1", "2", "3", "4", "5", "6", "7", "8"]

// file symbols from a to h //
var FileSymbols:[String] = ["a", "b", "c", "d", "e", "f", "g", "h"]

// the buttons to be displayed in the alert window //
var AlertButtons:[String] = []

// engine messages converted to tokens //
var EngineMessageTokens:[String] = []

// the components of a filename //
var FilenameComponents:[String] = []

// the current mouse location //
var MouseLocation:NSPoint = NSPoint()

// the rank and file location of a square on the board //
var RankAndFile:NSPoint = NSPoint()

// the starting square for a moving piece to be animated //
var StartSquare:NSPoint = NSPoint()

// the ending square for a moving piece to be animated //
var EndSquare:NSPoint = NSPoint()

// the temporary square to hold the start square for a swap //
var TempSquare:NSPoint = NSPoint()

// animation start point //
var AnimStart:NSPoint = NSPoint()

// animation end point //
var AnimEnd:NSPoint = NSPoint()

// a game sound //
var GameSound:NSSound = NSSound()

// the gallery grid snapshot //
var GalleryGridSnapshot:NSImage = NSImage()

// the gallery grid image //
var GalleryGridImage:NSImage = NSImage()

// the board snapshot image //
var SnapshotImage:NSImage = NSImage()

// the captured piece highlight image //
var CapturedHighlight:NSImage = NSImage()

// the threatened piece highlight image //
var ThreatenedHighlight:NSImage = NSImage()

// the animate progress image //
var ProgressImage:NSImage = NSImage()

// the enpassant square highlight //
var EnpassantHighlight:NSImage = NSImage()

// the last move highlight image //
var LastMoveHighlight:NSImage = NSImage()

// the illustration last move highlight //
var IllustrationLastMoveHighlight:CGImage!

// the selected square highlight image //
var SelectedSquareHighlight:NSImage = NSImage()

// the eraser icon for the position editor //
var Eraser:NSImage = NSImage(named:NSImage.Name(rawValue:"eraser_icon"))!

// the message icon image //
var MessageIcon:NSImage = NSImage(named:NSImage.Name(rawValue:"message_icon"))!

// the validated icon image //
var ValidatedIcon:NSImage = NSImage(named:NSImage.Name(rawValue:"validated_icon"))!

// the warning icon image //
var WarningIcon:NSImage = NSImage(named:NSImage.Name(rawValue:"warning_icon"))!

// the exit icon image //
var ExitIcon:NSImage = NSImage(named:NSImage.Name(rawValue:"exit_icon"))!

// the play icon //
var PlayIcon:NSImage = NSImage(named:NSImage.Name(rawValue:"start_icon"))!

// the pause icon //
var PauseIcon:NSImage = NSImage(named:NSImage.Name(rawValue:"pause_icon"))!

// the human opponent icon //
var HumanOpponentIcon:NSImage = NSImage()

// the greenchess opponent icon //
var GreenChessOpponentIcon = NSImage(named:NSImage.Name(rawValue:"greenchess_opponent"))!

// the uci engine opponent icon //
var UCIEngineOpponentIcon = NSImage(named:NSImage.Name(rawValue:"uciengine_opponent"))!

// the current diagram color //
var DiagramColor:NSColor = NSColor()

// the read the opening book thread //
var ReadOpeningBookThread:Thread = Thread()

// the engine computation thread //
var GCEngineThread:Thread = Thread()

// the game sound thread //
var GameSoundThread:Thread = Thread()

// the save game movie thread //
var SaveGameMovieThread:Thread = Thread()

// the save gallery grid movie thread //
var SaveGalleryGridMovieThread:Thread = Thread()

// the gallery grid thread //
var GalleryGridThread:Thread = Thread()

// the open game thread //
var OpenGameThread:Thread = Thread()

// the name of chess engine //
var EngineName:String = ""

// the engine filename document type extension //
var FilenameExtension:String = ""

// the name of a game file to open //
var GameFilename:String = ""

// a line of text from uci engine messages string //
var EngMsg:String = ""

// the beginning square of the move //
var FromSquare:String = ""

// the ending square of the move //
var ToSquare:String = ""

// the move in UCI notation //
var UCIMove:String = ""

// the data string read in from chess engine //
var EngineRawData:String = ""

// the string symbol that the pawn promoted to //
var PromotionSymbol:String = ""

// a move annotation string //
var AnnotationString:String = ""

// the string to pattern search for if is a substring of another string //
var SearchString:String = ""

// the return code from an alert //
var ReturnCode:Int = 0

// the panel for saving the game to the chosen filename and directory //
var SavePanel:NSSavePanel = NSSavePanel()

// the panel for opening a pgn game file //
var OpenPanel:NSOpenPanel = NSOpenPanel()

// the panel for appending a pgn game file //
var AppendPanel:NSOpenPanel = NSOpenPanel()

// the engine write file handle //
var EngineFileHandleWrite:FileHandle = FileHandle()

// the engine read file handle //
var EngineFileHandleRead:FileHandle = FileHandle()

// a list of pgn moves for the pgn moves list data source //
var PGNMovesList:[String] = []

// the pgn moves list for the annotator //
var PGNMovesListForAnnotator:[String] = []

// the game info list //
var GameInfoList:[String] = []

// the array containing the individual fen string components //
var FENStringArray:[String] = []

// the light chess square images //
var ChessLightSquareImages:[NSImage] = []

// the dark chess square images //
var ChessDarkSquareImages:[NSImage] = []

// the chess board images //
var ChessBoardImages:[NSImage] = []

// array of the chess piece images //
var ChessPieceImages:[NSImage] = []

// array of the chess piece cgimages //
var ChessPieceCGImages:[CGImage] = []

// array of all the game sounds //
var GameSounds:[NSSound] = []

// list of all the moves currently made in the game //
var MoveList:[String] = []

// a temporary move list //
var TempMoveList:[String] = []

// the raw stockfish analysis move list //
var RawStockfishMoveList:[String] = []

// the stockfish analysis move list //
var StockfishMoveList:[String] = []

// the stockfish principal variation line of play //
var StockfishPV:String = ""

// the stockfish analysis string //
var StockfishAnalysisString:String = ""

// the saved position data array //
var SavedPositionData:[UnsafeMutableRawPointer] = []

// list of all the moves for use in reading in the chess opening book //
var OpeningBookMoveList:[String] = []

// list of all the move annotations in the game //
var AnnotationList:[String] = []

// the previous list of all the moves currently made in the game //
var PreviousMoveList:[String] = []

// the previous list of all the move annotations in the game //
var PreviousAnnotationList:[String] = []

// list of squares where a capture can occur //
var ListOfCaptureSquares:[String] = []

// list of squares where a piece can be threatened //
var ListOfThreatenSquares:[String] = []

// the app preferences //
var AppPreferences:UserDefaults = UserDefaults.standard

// the command string to send to the engine //
var EngineCommand:String = ""

// a pgn move string for the pgn moves list data source //
var PGNMove:String = ""

// a pgn label string //
var PGNLabel:String = ""

// the fen string //
var FENString:String = ""

// the saved fen layout string to use for the position editor //
var FENSavedLayout:String = ""

// the saved fen start move number to use for the position editor //
var FENSavedStartMoveNumber:String = ""

// the saved fen color to move to use for the position editor //
var FENSavedColor:String = ""

// the saved fen castling rights to use for the position editor //
var FENSavedCastlingRights:String = ""

// the saved fen enpassant target square to use for the position editor //
var FENSavedEnpassantSquare:String = ""

// the selected fen piece from the fen layout for the board //
var FENPiece:String = ""

// the date formatter //
var TheDateFormatter:DateFormatter = DateFormatter()

// a basic move animation //
var MoveAnimation:CABasicAnimation = CABasicAnimation(keyPath:"position")

// the saved position animation //
var PositionAnimation:CABasicAnimation = CABasicAnimation(keyPath:"opacity")

// the board snapshot bitmap image //
var SnapshotBitmap:NSBitmapImageRep!

// the gallery grid cgimage //
var GalleryGridCGImage:CGImage!

// a zobrist hash value //
var ZobristHash:CUnsignedLongLong = 0

// the game list data //
var GameListData:GameListDataSource = GameListDataSource()

// the pgn moves list data //
var PGNMovesData:PGNMovesDataSource = PGNMovesDataSource()

// the game info data //
var GameInfoData:GameInfoDataSource = GameInfoDataSource()

// the notification center //
var Notify:NotificationCenter = NotificationCenter.default

// a gallery grid view //
var GalleryGridView:GalleryGrid = GalleryGrid()

// the gallery grid review button //
var GalleryGridReview:NSToolbarItem = NSToolbarItem()

// a gallery grid window //
var GalleryGridWindow:AWindow = AWindow()

// the gallery grid toolbar //
var GalleryGridToolbar:NSToolbar!

// the gallery grid color //
var GalleryGridColor:NSColor = NSColor()

// a gallery grid slider //
var GalleryGridSlider:NSSlider = NSSlider()

// the board preview image //
@IBOutlet var BoardPreview:NSImageView!

// the window color preview //
@IBOutlet var WindowColorPreview:NSImageView!

// the game clock border color view //
@IBOutlet var ClockColorView:ColorsView!

// the game clock background color view //
@IBOutlet var ClockBackgroundColorView:ColorsView!

// the rainbow background colors view //
@IBOutlet var RainbowBackgroundView:ColorsView!

// the blue background colors view //
@IBOutlet var BlueBackgroundView:ColorsView!

// the yellow background colors view //
@IBOutlet var YellowBackgroundView:ColorsView!

// the gray background colors view //
@IBOutlet var GrayBackgroundView:ColorsView!

// the teal background colors view //
@IBOutlet var TealBackgroundView:ColorsView!

// the green background colors view //
@IBOutlet var GreenBackgroundView:ColorsView!

// the brown background colors view //
@IBOutlet var BrownBackgroundView:ColorsView!

// the board labels color view //
@IBOutlet var BoardLabelsColorView:ColorsView!

// the game info color view //
@IBOutlet var GameInfoColorView:ColorsView!

// the move list color view //
@IBOutlet var MoveListColorView:ColorsView!

// the game list color view //
@IBOutlet var GameListColorView:ColorsView!

// the game gallery view //
@IBOutlet var GameGalleryView:GalleryGrid!

// the game grid view //
@IBOutlet var GameGridView:GalleryGrid!

// the main toolbar //
@IBOutlet var MainToolbar:NSToolbar!

// the gallery toolbar //
@IBOutlet var GalleryToolbar:NSToolbar!

// the grid toolbar //
@IBOutlet var GridToolbar:NSToolbar!

// the preferences toolbar //
@IBOutlet var PreferencesToolbar:NSToolbar!

// the opponent toolbar item //
@IBOutlet var OpponentToolbarItem:NSToolbarItem!

// the button to go left in the game explorer //
@IBOutlet var GameExplorerLeft:NSToolbarItem!

// the button to go right in the game explorer //
@IBOutlet var GameExplorerRight:NSToolbarItem!

// the button to go right in the game gallery //
@IBOutlet var GameGalleryRight:NSToolbarItem!

// the button to go right in the game grid //
@IBOutlet var GameGridRight:NSToolbarItem!

// the button to go left in the game gallery //
@IBOutlet var GameGalleryLeft:NSToolbarItem!

// the button to go left in the game grid //
@IBOutlet var GameGridLeft:NSToolbarItem!

// the game grid movie button //
@IBOutlet var GameGridMovie:NSToolbarItem!

// the game grid snapshot button //
@IBOutlet var GameGridSnapshot:NSToolbarItem!

// the game grid erase button //
@IBOutlet var GameGridErase:NSToolbarItem!

// the game grid undo shape button //
@IBOutlet var GameGridUndoShape:NSToolbarItem!

// the game grid blue color button //
@IBOutlet var GameGridBlueColor:NSToolbarItem!

// the game grid cyan color button //
@IBOutlet var GameGridCyanColor:NSToolbarItem!

// the game grid magenta color button //
@IBOutlet var GameGridMagentaColor:NSToolbarItem!

// the game grid green color button //
@IBOutlet var GameGridGreenColor:NSToolbarItem!

// the game grid red color button //
@IBOutlet var GameGridRedColor:NSToolbarItem!

// the game grid yellow color button //
@IBOutlet var GameGridYellowColor:NSToolbarItem!

// the game grid black color button //
@IBOutlet var GameGridBlackColor:NSToolbarItem!

// the game grid white color button //
@IBOutlet var GameGridWhiteColor:NSToolbarItem!

// the diagram blue color toolbar item //
@IBOutlet var DiagramBlueColorItem:NSToolbarItem!

// the diagram cyan color toolbar item //
@IBOutlet var DiagramCyanColorItem:NSToolbarItem!

// the diagram magenta color toolbar item //
@IBOutlet var DiagramMagentaColorItem:NSToolbarItem!

// the diagram white color toolbar item //
@IBOutlet var DiagramWhiteColorItem:NSToolbarItem!

// the diagram yellow color toolbar item //
@IBOutlet var DiagramYellowColorItem:NSToolbarItem!

// the diagram black color toolbar item //
@IBOutlet var DiagramBlackColorItem:NSToolbarItem!

// the diagram green color toolbar item //
@IBOutlet var DiagramGreenColorItem:NSToolbarItem!

// the diagram red color toolbar item //
@IBOutlet var DiagramRedColorItem:NSToolbarItem!

// the diagram snapshot toolbar item //
@IBOutlet var DiagramSnapshotItem:NSToolbarItem!

// the undo path toolbar item //
@IBOutlet var UndoPathItem:NSToolbarItem!

// the clear diagram toolbar item //
@IBOutlet var ClearDiagramItem:NSToolbarItem!

// the game gallery movie button //
@IBOutlet var GameGalleryMovie:NSToolbarItem!

// the game gallery snapshot button //
@IBOutlet var GameGallerySnapshot:NSToolbarItem!

// the game gallery erase button //
@IBOutlet var GameGalleryErase:NSToolbarItem!

// the game gallery undo shape button //
@IBOutlet var GameGalleryUndoShape:NSToolbarItem!

// the game gallery blue color button //
@IBOutlet var GameGalleryBlueColor:NSToolbarItem!

// the game gallery magenta color button //
@IBOutlet var GameGalleryMagentaColor:NSToolbarItem!

// the game gallery cyan color button //
@IBOutlet var GameGalleryCyanColor:NSToolbarItem!

// the game gallery yellow color button //
@IBOutlet var GameGalleryYellowColor:NSToolbarItem!

// the game gallery black color button //
@IBOutlet var GameGalleryBlackColor:NSToolbarItem!

// the game gallery white color button //
@IBOutlet var GameGalleryWhiteColor:NSToolbarItem!

// the game gallery green color button //
@IBOutlet var GameGalleryGreenColor:NSToolbarItem!

// the game gallery red color button //
@IBOutlet var GameGalleryRedColor:NSToolbarItem!

// the enter analysis mode button //
@IBOutlet var EnterAnalysisModeButton:NSToolbarItem!

// enter the position editor mode button //
@IBOutlet var EnterPositionEditorModeButton:NSToolbarItem!

// the enter diagram mode button //
@IBOutlet var EnterDiagramModeButton:NSToolbarItem!

// the game explorer snapshot button //
@IBOutlet var GameExplorerSnapshotButton:NSToolbarItem!

// the game explorer game to movie button //
@IBOutlet var GameExplorerMovieButton:NSToolbarItem!

// the exit the game explorer button //
@IBOutlet var ExitGameExplorerButton:NSToolbarItem!

// button to start the game review //
@IBOutlet var GameReviewButton:NSToolbarItem!

// the game gallery review button //
@IBOutlet var GameGalleryReviewButton:NSToolbarItem!

// the game grid review button //
@IBOutlet var GameGridReviewButton:NSToolbarItem!

// position editor query button of whether white is to move next //
@IBOutlet var WhiteMovesToolbarButton:NSToolbarItem!

// position editor button of whether white can still kingside castle //
@IBOutlet var WhiteKingsideCastleToolbarButton:NSToolbarItem!

// position editor button of whether black can still kingside castle //
@IBOutlet var BlackKingsideCastleToolbarButton:NSToolbarItem!

// position editor button of whether white can still queenside castle //
@IBOutlet var WhiteQueensideCastleToolbarButton:NSToolbarItem!

// position editor button of whether black can still queenside castle //
@IBOutlet var BlackQueensideCastleToolbarButton:NSToolbarItem!

// button for if want to save the edited position from the position editor //
@IBOutlet var PositionEditorSaveToolbarButton:NSToolbarItem!

// the position editor enpassant square button //
@IBOutlet var PositionEditorEP:NSToolbarItem!

// the position editor second menu button //
@IBOutlet var PositionEditorSecondMenu:NSToolbarItem!

// the position editor eraser button //
@IBOutlet var PositionEditorEraser:NSToolbarItem!

// the position editor white pawn //
@IBOutlet var PositionEditorWhitePawn:NSToolbarItem!

// the position editor white knight //
@IBOutlet var PositionEditorWhiteKnight:NSToolbarItem!

// the position editor white bishop //
@IBOutlet var PositionEditorWhiteBishop:NSToolbarItem!

// the position editor white rook //
@IBOutlet var PositionEditorWhiteRook:NSToolbarItem!

// the position editor white queen //
@IBOutlet var PositionEditorWhiteQueen:NSToolbarItem!

// the position editor white king //
@IBOutlet var PositionEditorWhiteKing:NSToolbarItem!

// the position editor black pawn //
@IBOutlet var PositionEditorBlackPawn:NSToolbarItem!

// the position editor black knight //
@IBOutlet var PositionEditorBlackKnight:NSToolbarItem!

// the position editor black bishop //
@IBOutlet var PositionEditorBlackBishop:NSToolbarItem!

// the position editor black rook //
@IBOutlet var PositionEditorBlackRook:NSToolbarItem!

// the position editor black queen //
@IBOutlet var PositionEditorBlackQueen:NSToolbarItem!

// the position editor black king //
@IBOutlet var PositionEditorBlackKing:NSToolbarItem!

// the exit diagram toolbar button //
@IBOutlet var ExitDiagramToolbarButton:NSToolbarItem!

// the button to click on if white won the game //
@IBOutlet var WhiteWonButton:NSButton!

// the button to click on if black won the game //
@IBOutlet var BlackWonButton:NSButton!

// the button to click on if the game ended in a draw //
@IBOutlet var DrawButton:NSButton!

// the button to click on if want to use the encyclopedia of chess openings //
@IBOutlet var EcoBookButton:NSButton!

// the autosave game data option button //
@IBOutlet var AutosaveGameButton:NSButton!

// the autoload stockfish option button //
@IBOutlet var AutoloadStockfishButton:NSButton!

// the stockfish analysis button //
@IBOutlet var ShowStockfishAnalysisButton:NSButton!

// the game sounds button //
@IBOutlet var GameSoundsButton:NSButton!

// the move by selecting square button //
@IBOutlet var MoveBySelectingSquaresButton:NSButton!

// the button to click on if want to highlight pieces which can be captured //
@IBOutlet var HighlightCapturedPiecesButton:NSButton!

// the button to click on if want to highlight pieces which are threatened //
@IBOutlet var HighlightThreatenedPiecesButton:NSButton!

// the button to click on if want to multithread the uci engine //
@IBOutlet var MultithreadUCIEngineButton:NSButton!

// the button to click on if want to highlight the last move made during a game //
@IBOutlet var HighlightLastMoveMadeButton:NSButton!

// the load position window //
@IBOutlet var LoadPositionWindow:AWindow!

// the game gallery window //
@IBOutlet var GameGalleryWindow:AWindow!

// the game grid window //
@IBOutlet var GameGridWindow:AWindow!

// the enter fen window //
@IBOutlet var FENWindow:AWindow!

// the about window //
@IBOutlet var AboutWindow:AWindow!

// the enter game info window //
@IBOutlet var EnterGameInfoWindow:AWindow!

// the timed game window //
@IBOutlet var TimedGameWindow:AWindow!

// the enter preferences window //
@IBOutlet var PreferencesWindow:AWindow!

// the main gui window //
@IBOutlet var MainWindow:AWindow!

// the game list window //
@IBOutlet var GameListWindow:AWindow!

// the game explorer window //
@IBOutlet var GameExplorerWindow:AWindow!

// the game annotator window //
@IBOutlet var GameAnnotatorWindow:AWindow!

// the database explorer window //
@IBOutlet var DatabaseExplorerWindow:AWindow!

// the game explorer move slider //
@IBOutlet var GameExplorerSlider:NSSlider!

// the game gallery slider //
@IBOutlet var GameGallerySlider:NSSlider!

// the game grid slider //
@IBOutlet var GameGridSlider:NSSlider!

// the game annotator slider //
@IBOutlet var GameAnnotatorSlider:NSSlider!

// the database explorer slider //
@IBOutlet var DatabaseExplorerSlider:NSSlider!

// the database game explorer slider //
@IBOutlet var DatabaseGameExplorerSlider:NSSlider!

// the animation rate slider //
@IBOutlet var AnimationRateSlider:NSSlider!

// the gallery and grid animation rate slider //
@IBOutlet var GalleryGridAnimationRateSlider:NSSlider!

// the chess board border visibility slider //
@IBOutlet var BorderVisibilitySlider:NSSlider!

// the board tint visibility slider //
@IBOutlet var BoardTintVisibilitySlider:NSSlider!

// the window texture opacity slider //
@IBOutlet var WindowTextureOpacitySlider:NSSlider!

// the board roughness slider //
@IBOutlet var BoardRoughnessSlider:NSSlider!

// the square border slider //
@IBOutlet var SquareBorderSlider:NSSlider!

// the board tint color slider //
@IBOutlet var BoardTintColorSlider:NSSlider!

// the clock background visibility slider //
@IBOutlet var ClockBackgroundVisibilitySlider:NSSlider!

// the chess board border texture slider //
@IBOutlet var BorderTextureSlider:NSSlider!

// the uci engine difficulty slider //
@IBOutlet var UCIEngineDifficultySlider:NSSlider!

// the uci engine hash slider //
@IBOutlet var UCIEngineHashSlider:NSSlider!

// the uci engine riskiness slider //
@IBOutlet var UCIEngineRiskinessSlider:NSSlider!

// the search time slider //
@IBOutlet var SearchTimeSlider:NSSlider!

// the hide game clock in panel menu item //
@IBOutlet var HideGameClockInPanelMenuItem:NSMenuItem!

// the hide captured pieces in panel menu item //
@IBOutlet var HideCapturedPiecesInPanelMenuItem:NSMenuItem!

// the new game menu item //
@IBOutlet var NewGameMenuItem:NSMenuItem!

// the board snapshot menu item //
@IBOutlet var BoardSnapshotMenuItem:NSMenuItem!

// the engine match menu item //
@IBOutlet var EngineMatchMenuItem:NSMenuItem!

// the hide application menu item //
@IBOutlet var HideMenuItem:NSMenuItem!

// the quit application menu item //
@IBOutlet var QuitMenuItem:NSMenuItem!

// the hide info panel menu item //
@IBOutlet var HideInfoMenuItem:NSMenuItem!

// the large info panel menu item //
@IBOutlet var LargeInfoMenuItem:NSMenuItem!

// the small info panel menu item //
@IBOutlet var SmallInfoMenuItem:NSMenuItem!

// the position editor menu item //
@IBOutlet var PositionEditorMenuItem:NSMenuItem!

// the current position editor menu item //
@IBOutlet var CurrentPositionEditorMenuItem:NSMenuItem!

// the chess engine opponent menu item //
@IBOutlet var UCIEngineOpponentMenuItem:NSMenuItem!

// the go stop menu item //
@IBOutlet var GoStopMenuItem:NSMenuItem!

// the enter game info menu item //
@IBOutlet var EnterGameInfoMenuItem:NSMenuItem!

// the about menu item //
@IBOutlet var AboutMenuItem:NSMenuItem!

// the save position menu item //
@IBOutlet var SavePositionMenuItem:NSMenuItem!

// the load position menu item //
@IBOutlet var LoadPositionMenuItem:NSMenuItem!

// the enter preferences menu item //
@IBOutlet var PreferencesMenuItem:NSMenuItem!

// the game gallery menu item //
@IBOutlet var GameGalleryMenuItem:NSMenuItem!

// the game grid memu item //
@IBOutlet var GameGridMenuItem:NSMenuItem!

// the search the game list menu item //
@IBOutlet var SearchGameListMenuItem:NSMenuItem!

// the open game menu item //
```

```
// the open game menu item //
@IBOutlet var OpenGameMenuItem:NSMenuItem!

// the save game menu item //
@IBOutlet var SaveGameMenuItem:NSMenuItem!

// the append game menu item //
@IBOutlet var AppendGameMenuItem:NSMenuItem!

// the game explorer menu item //
@IBOutlet var GameExplorerMenuItem:NSMenuItem!

// the database explorer menu item //
@IBOutlet var DatabaseExplorerMenuItem:NSMenuItem!

// the game annotator menu item //
@IBOutlet var GameAnnotatorMenuItem:NSMenuItem!

// the timed game menu item //
@IBOutlet var TimedGameMenuItem:NSMenuItem!

// the enter fen menu item //
@IBOutlet var EnterFENMenuItem:NSMenuItem!

// the rotate board menu item //
@IBOutlet var RotateBoardMenuItem:NSMenuItem!

// the human opponent menu item //
@IBOutlet var HumanOpponentMenuItem:NSMenuItem!

// the engine opponent menu item //
@IBOutlet var EngineOpponentMenuItem:NSMenuItem!

// the text field for the time the chess engine will search for //
@IBOutlet var SearchTimeView:NSTextField!

// the app name view in the about window //
@IBOutlet var AppNameView:NSTextField!

// the text field for the animation rate of the pieces moving //
@IBOutlet var AnimationRateView:NSTextField!

// the gallery grid animation rate view //
@IBOutlet var GalleryGridAnimationRateView:NSTextField!

// the chess board border visibility view //
@IBOutlet var BorderVisibilityView:NSTextField!

// the board tint visibility view //
@IBOutlet var BoardTintVisibilityView:NSTextField!

// the window texture opacity view //
@IBOutlet var WindowTextureOpacityView:NSTextField!

// the board roughness view //
@IBOutlet var BoardRoughnessView:NSTextField!

// the square border view //
@IBOutlet var SquareBorderView:NSTextField!

// the board tint color view //
@IBOutlet var BoardTintColorView:NSTextField!

// the clock background opacity view //
@IBOutlet var ClockBackgroundOpacityView:NSTextField!

// the chess board border texture view //
@IBOutlet var BorderTextureView:NSTextField!

// the text field for the uci engine difficulty //
@IBOutlet var UCIEngineDifficultyView:NSTextField!

// the text field for the uci engine hash size //
@IBOutlet var UCIEngineHashView:NSTextField!

// the text field for the uci engine riskiness view //
@IBOutlet var UCIEngineRiskinessView:NSTextField!

// the text field for the current move in the annotator //
@IBOutlet var PGNMoveView:NSTextField!

// the load position view //
@IBOutlet var LoadPositionView:NSView!

// the annotation view //
@IBOutlet var AnnotationView:NSView!

// the table view of a list of games from a game list //
@IBOutlet var GameListTableView:NSTableView!

// the table view of the pgn moves list table view //
@IBOutlet var PGNMovesTableView:NSTableView!

// the game info table view //
@IBOutlet var GameInfoTableView:NSTableView!

// the rainbow colors text field //
@IBOutlet var RainbowBackgroundField:NSTextField!

// the game clock color text field //
@IBOutlet var GameClockColorField:NSTextField!

// the game clock background color text field //
@IBOutlet var GameClockBackgroundColorField:NSTextField!

// the teal colors text field //
@IBOutlet var TealBackgroundField:NSTextField!

// the blue colors text field //
@IBOutlet var BlueBackgroundField:NSTextField!

// the green colors text field //
@IBOutlet var GreenBackgroundField:NSTextField!

// the yellow colors text field //
@IBOutlet var YellowBackgroundField:NSTextField!

// the gray colors text field //
@IBOutlet var GrayBackgroundField:NSTextField!

// the brown colors text field //
@IBOutlet var BrownBackgroundField:NSTextField!

// the board labels text field //
@IBOutlet var BoardLabelsField:NSTextField!

// the board textures field //
@IBOutlet var BoardTexturesField:NSTextField!

// the piece set field //
@IBOutlet var PieceSetField:NSTextField!

// the move list text field //
@IBOutlet var MoveListField:NSTextField!

// the game list text field //
@IBOutlet var GameListField:NSTextField!

// the game info text field //
@IBOutlet var GameInfoField:NSTextField!

// the game list search box to search for a specific game //
@IBOutlet var GameListSearchBox:NSSearchField!

// the game annotation text field //
@IBOutlet var GameAnnotationTextField:NSTextView!

// the text field for the name of the person playing as white //
@IBOutlet var WhiteNameTextField:NSTextField!

// the text field for the time control of the game //
@IBOutlet var TimeControlTextField:NSTextField!

// the text field for the opening of the game //
@IBOutlet var OpeningTextField:NSTextField!

// the text field for the section of the game //
@IBOutlet var SectionTextField:NSTextField!

// the text field for the board number of the game //
@IBOutlet var BoardNumberTextField:NSTextField!

// the text field for the white elo of the game //
@IBOutlet var WhiteEloTextField:NSTextField!

// the text field for the black elo of the game //
@IBOutlet var BlackEloTextField:NSTextField!

// the text field for the white uscf of the game //
@IBOutlet var WhiteUSCFTextField:NSTextField!

// the text field for the black uscf of the game //
@IBOutlet var BlackUSCFTextField:NSTextField!

// the text field for the annotator of the game //
@IBOutlet var AnnotatorTextField:NSTextField!

// the enter fen to convert text field //
@IBOutlet var EnterFENTextField:NSTextField!

// the text field for the eco of the game //
@IBOutlet var ECOTextField:NSTextField!

// the text field for the name of the person playing as black //
@IBOutlet var BlackNameTextField:NSTextField!

// the text field for the geographic location of the game site //
@IBOutlet var SiteLocationTextField:NSTextField!

// the text field for the title of the chess event or tournament //
@IBOutlet var EventTitleTextField:NSTextField!

// the text field for the current game round being played //
@IBOutlet var RoundNumberTextField:NSTextField!

// the date the game was played on from the games setting window //
@IBOutlet var GameDate:NSDatePicker!

// the board textures button //
@IBOutlet var BoardTexturesButton:NSPopUpButton!

// the piece set button //
@IBOutlet var PieceSetButton:NSPopUpButton!

// the timed game opponent popup button //
@IBOutlet var TimedGameOpponentButton:NSPopUpButton!

// the timed game color popup button //
@IBOutlet var TimedGameColorButton:NSPopUpButton!

// the timed game time control popup button //
@IBOutlet var TimedGameTimeButton:NSPopUpButton!

// the board labels font button //
@IBOutlet var BoardLabelsFontButton:NSPopUpButton!

// the move list font button //
@IBOutlet var MoveListFontButton:NSPopUpButton!

// the game info font button //
@IBOutlet var GameInfoFontButton:NSPopUpButton!

// the game list font button //
@IBOutlet var GameListFontButton:NSPopUpButton!

// the board labels bold button //
@IBOutlet var BoardLabelsBoldButton:NSPopUpButton!

// the move list bold button //
@IBOutlet var MoveListBoldButton:NSPopUpButton!

// the game info bold button //
@IBOutlet var GameInfoBoldButton:NSPopUpButton!

// the game list bold button //
@IBOutlet var GameListBoldButton:NSPopUpButton!

// the board labels font sizes button //
@IBOutlet var BoardLabelsSizesButton:NSPopUpButton!

// the move list font sizes button //
@IBOutlet var MoveListSizesButton:NSPopUpButton!

// the game info sizes button //
@IBOutlet var GameInfoSizesButton:NSPopUpButton!

// the game list sizes button //
@IBOutlet var GameListSizesButton:NSPopUpButton!

// the game clock //
@IBOutlet var GameClock:GameClockDisplay!

// the game clock preview //
@IBOutlet var GameClockPreview:GameClockDisplay!

// the view to display the captured white pieces on //
@IBOutlet var CapturedPiecesWhiteView:CapturedPiecesDisplay!

// the view to display the captured black pieces on //
@IBOutlet var CapturedPiecesBlackView:CapturedPiecesDisplay!

// the awake from nib function //
override func awakeFromNib()
{
   // if already called the awake from nib function //
   if Awoke
   {
      // exit function since awake from nib was already called //
      return
   }
   // if haven't called awake from nib yet //
   else
   {
      // set that awoke from nib has been called //
      Awoke = true
   }

   // allow the view to draw concurrently //
   self.canDrawConcurrently = true

   // set the main window to allow concurrent drawing //
   MainWindow.allowsConcurrentViewDrawing = true

   // initialize all images and sounds used //
   InitializeMedia()

   // initialize all the app preferences //
   InitializePreferences()

   // initialize all the data sources //
   InitializeDataSources()

   // run the thread to read in the chess opening book //
   RunReadOpeningBookThread()

   // start a new chess game //
   StartNewGame(true)

   // if chose to autosave game data //
   if AutosaveGame
   {
      // extract from the autosave game data //
      ExtractFromGameData()
   }

   // reset the game clock //
   ResetGameClock()

   // create a game timer that gets called ten times per second //
   RunLoop.current.add(Timer(timeInterval:0.1, target:self, selector:#selector(GreenChess
                       .UpdateGameClock), userInfo:nil, repeats:true), forMode:RunLoopMode
                       .commonModes)

   // add a tracking rectangle for the board //
   self.addTrackingRect(NSMakeRect(30, 30, 600, 600), owner:self, userData:nil, assumeInside:
                        false)

   // disable cursor rects //
   self.window!.disableCursorRects()

   // set the double click action command for the game list table view //
   GameListTableView.doubleAction = #selector(GreenChess.LoadGameFromListButtonAction(_:))

   // remove all items in the timed game opponent popup button //
   TimedGameOpponentButton.removeAllItems()

   // add greenchess to the timed game popup button //
   TimedGameOpponentButton.addItem(withTitle:"GreenChess")

   // update the main window //
   UpdateMainWindow()

   // center the main window //
   MainWindow.center()

   // update the sheet sizes //
   UpdateSheetSizes()

   // add a timer to remove extra edit menu items //
   RunLoop.current.add(Timer(timeInterval:0.2, target:self, selector:
                       #selector(GreenChess.RemoveEditItems), userInfo:nil, repeats:false),
                       forMode:RunLoopMode.commonModes)
}

// set the window background color //
func SetWindowBackgroundColor(_ Color:NSColor)
{
   // save the current window color //
   CurrentWindowColor = Color

   // set the window background pattern size //
   WindowBackgroundPattern = NSImage(size:NSSize(width:40, height:40))

   // lock the window background pattern focus //
   WindowBackgroundPattern.lockFocus()

   // set the fill color //
   Color.setFill()

   // fill the rectangle with the given color //
   NSBezierPath(rect:NSMakeRect(0, 0, 40, 40)).fill()

   // draw the window background overlay into the rectangle //
   WindowBackgroundOverlay.draw(in:NSMakeRect(0, 0, 40, 40), from:NSZeroRect, operation:
                                NSCompositingOperation.sourceOver, fraction:0.75 *
                                CGFloat(WindowTextureOpacity) / 100.0)

   // unlock the window background pattern focus //
   WindowBackgroundPattern.unlockFocus()

   // set the main window background color to the window background pattern //
   MainWindow.backgroundColor = NSColor(patternImage:WindowBackgroundPattern)

   // update the window color pattern preview //
   WindowColorPreview.image = WindowBackgroundPattern

   // update the game clock window color //
   GameClock.UpdateWindowColor(MainWindow.backgroundColor)

   // update the game clock window preview color //
   GameClockPreview.UpdateWindowColor(MainWindow.backgroundColor)

   // update the background color layer //
   UpdateBackgroundColorLayer()
}

// create the window background overlay //
func CreateWindowBackgroundOverlay()
{
   // set the window background overlay size //
   WindowBackgroundOverlay = NSImage(size:NSSize(width:40, height:40))

   // lock the window background overlay focus //
   WindowBackgroundOverlay.lockFocus()

   // loop over the roughness x index //
   for RoughnessXIndex in 0 ..< 10
   {
      // loop over the roughness y index //
      for RoughnessYIndex in 0 ..< 10
      {
         // set the fill color to light gray //
         NSColor.lightGray.setFill()

         // make a light gray rectangle //
         NSBezierPath(rect:NSMakeRect(CGFloat(RoughnessXIndex * 4), CGFloat(RoughnessYIndex * 4)
                      + 2.5, 4, 1)).fill()

         // make a light gray rectangle //
         NSBezierPath(rect:NSMakeRect(CGFloat(RoughnessXIndex * 4) + 2.5,
                      CGFloat(RoughnessYIndex * 4), 1, 4)).fill()

         // set the fill color to gray //
         NSColor.gray.setFill()

         // make a gray rectangle //
         NSBezierPath(rect:NSMakeRect(CGFloat(RoughnessXIndex * 4), CGFloat(RoughnessYIndex * 4)
                      + 0.5, 4, 1)).fill()

         // make a gray rectangle //
         NSBezierPath(rect:NSMakeRect(CGFloat(RoughnessXIndex * 4) + 0.5,
                      CGFloat(RoughnessYIndex * 4), 1, 4)).fill()

         // set the fill color to light gray //
         NSColor.lightGray.setFill()

         // if on an even square //
         if (RoughnessXIndex + RoughnessYIndex) % 2 == 0
         {
            // make a light gray rectangle //
            NSBezierPath(rect:NSMakeRect(CGFloat(RoughnessXIndex * 4) + 0.5,
                         CGFloat(RoughnessYIndex * 4) + 2.5, 1, 1)).fill()
         }
         // if is an odd square //
         else
         {
            // make a light gray rectangle //
            NSBezierPath(rect:NSMakeRect(CGFloat(RoughnessXIndex * 4) + 2.5,
                         CGFloat(RoughnessYIndex * 4) + 0.5, 1, 1)).fill()
         }
      }
   }

   // unlock the window background overlay focus //
   WindowBackgroundOverlay.unlockFocus()
}

// get the window sheet attachment point //
func window(_ Window:NSWindow, willPositionSheet Sheet:NSWindow, using Rect:NSRect) -> NSRect
{
   // return the window sheet attachment point //
   return CGRect(x:0, y:705, width:Window.frame.size.width, height:0)
}

// if the window resized //
func windowDidResize(_ Notif:Notification)
{
   // update the sheet sizes //
   UpdateSheetSizes()
}

// the window will exit full screen //
func windowWillExitFullScreen(_ Notif:Notification)
{
   // clear the selected square if needed //
   ClearSelectedSquareIfNeeded()
}

// the window did exit full screen //
func windowDidExitFullScreen(_ Notif:Notification)
{
   // set that no longer in full screen mode //
   FullScreen = false

   // update the main window //
   UpdateMainWindow()

   // if should close the main window //
   if ShouldCloseMainWindow
   {
      // close the main window //
      MainWindow.close()
   }

   // set that no longer need to close the main window //
   ShouldCloseMainWindow = false
}

// clear the tables //
func ClearTables()
{
   // set game info list to empty list //
   GameInfoList = [" ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ",
                   " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " "]

   // update the game info list //
   UpdateGameInfoList()

   // set pgn moves list to empty list //
   PGNMovesList = [" ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ",
                   " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " ", " "]

   // remove all the frames //
   Frames.removeAll()

   // remove all the illustrations //
   Illustrations.removeAll()

   // set stockfish analysis to display to false //
   StockfishAnalysisToDisplay = false

   // update the moves list //
   UpdateMoveList()

   // replay all the moves //
   ReplayMoves(0, End:MoveList.count, Display:false)
}

// the window entered full screen //
func windowDidEnterFullScreen(_ Notif:Notification)
{
   // clear the tables //
   ClearTables()
}

// the window will enter full screen mode //
func windowWillEnterFullScreen(_ Notif:Notification)
{
   // clear the selected square if needed //
   ClearSelectedSquareIfNeeded()

   // set the info panel size to large //
   InfoPanelAction(false, Small:false, Large:true)

   // set that in full screen mode //
   FullScreen = true
}

// copy the current game to the pasteboard //
@IBAction func CopyGameToPasteboardAction(_:AnyObject)
{
   // clear the pasteboard contents //
   Pasteboard.clearContents()

   // set the pasteboard contents to the current game pgn //
   Pasteboard.setString(GamePGN(), forType:NSPasteboard.PasteboardType.string)
}

// open the greenchess webpage //
@IBAction func OpenGreenChessWebpage(_:AnyObject)
{
   // open the greenchess webpage //
   NSWorkspace.shared.open(URL(string:"https://greenchess.com")!)
}

// open the greenchess support webpage //
@IBAction func OpenGreenChessSupportWebpage(_:AnyObject)
{
   // open the greenchess support webpage //
   NSWorkspace.shared.open(URL(string:"https://greenchess.com/contact")!)
}

// open the stockfish support webpage //
@IBAction func OpenStockfishSupportWebpage(_:AnyObject)
{
   // open the stockfish support webpage //
   NSWorkspace.shared.open(URL(string:"https://stockfishchess.org/download")!)
}

// open the gnu license webpage //
@IBAction func OpenGNULicenseWebpage(_:AnyObject)
{
   // open the gnu license webpage //
   NSWorkspace.shared.open(URL(string:"https://gnu.org/licenses/gpl-3.0.en.html")!)
}

// return the view for the specified row in the table view //
func tableView(_ View:NSTableView, viewFor Column:NSTableColumn?, row Row:Int) -> NSView?
{
   // if the current view is the pgn moves table view //
   if View == PGNMovesTableView
   {
      // the modified table row to account for max rows //
      TableRow = PGNMovesData.PGNMovesList.count * 2 - min(PGNMovesData.PGNMovesList.count * 2,
                 PGNMovesData.MaxRows) + Row

      // create a table cell //
      TableCell = View.makeView(withIdentifier:NSUserInterfaceItemIdentifier(rawValue:"PGN"),
                                owner:self) as! NSTableCellView

      // if an even row //
      if TableRow % 2 == 0
      {
         // set the table cell text to empty //
         TableCell.textField!.stringValue = ""

         // if index is less than the illustrations count //
         if TableRow / 2 < Illustrations.count
         {
            // set the table cell image view to the illustration //
            TableCell.imageView!.image = Illustrations[TableRow / 2]
         }
         // if is the stockfish analysis string //
         else if TableRow / 2 == PGNMovesList.count - 1 && StockfishAnalysisToDisplay
         {
            // set the table cell image view to the stockfish icon //
            TableCell.imageView!.image = UCIEngineOpponentIcon
         }
         // index is out of bounds //
         else
         {
            // set the table cell image view to be empty //
            TableCell.imageView!.image = NSImage()
         }
      }
      // if is an odd row //
      else
      {
         // set the table cell text field attributed string value to the pgn move //
         TableCell.textField!.attributedStringValue = PGNMovesData.PGNMovesList[TableRow / 2]

         // set the table cell image view to be an empty image //
         TableCell.imageView!.image = NSImage()
      }

      // return the cell //
      return TableCell
   }
   // if is the game info table view //
   else if View == GameInfoTableView
   {
      // create a table cell //
      TableCell = View.makeView(withIdentifier:NSUserInterfaceItemIdentifier(rawValue:"Info"),
                                owner:self) as! NSTableCellView

      // set the table cell attributed string value //
      TableCell.textField!.attributedStringValue = GameInfoData.GameInfoList[Row]

      // return the table cell //
      return TableCell
   }
   // if is the game list table view //
   else if View == GameListTableView
   {
      // create a table cell //
      TableCell = View.makeView(withIdentifier:NSUserInterfaceItemIdentifier(rawValue:"Games"),
                                owner:self) as! NSTableCellView

      // if the selected row is equal to the row index make font color white //
      if GameListData.SelectedRow == Row
      {
         // return the attributed string at the row index in the game list //
         TableCell.textField!.attributedStringValue = NSAttributedString(string:GameListData
                                                      .GameList[Row], attributes:
                                                      [NSAttributedStringKey.font:GameListData
                                                      .Font, NSAttributedStringKey
                                                      .foregroundColor:NSColor.white])
      }
      // the selected row is not equal to the row index so make chosen font color //
      else
      {
         // return the attributed string at the row index in the game list //
         TableCell.textField!.attributedStringValue =  GameListData.AttributedGameList[Row]
      }

      // return the table cell //
      return TableCell
   }
   // not a valid table view //
   else
   {
      // return a nil value //
      return nil
   }
}

// get the dynamic table view row height //
func tableView(_ TableView:NSTableView, heightOfRow Row:Int) -> CGFloat
{
   // if is the pgn moves table views //
   if TableView == PGNMovesTableView
   {
      // the modified table row to account for max rows //
      TableRow = PGNMovesData.PGNMovesList.count * 2 - min(PGNMovesData.PGNMovesList.count * 2,
                 PGNMovesData.MaxRows) + Row

      // if is an even row //
      if TableRow % 2 == 0
      {
         // if is the stockfish analysis string //
         if TableRow / 2 == PGNMovesList.count - 1 && StockfishAnalysisToDisplay
         {
            // return the row height of 26 //
            return 26
         }
         // if is an illustration //
         else
         {
            // return the row size for a board illustration //
            return IllustrationSize
         }
      }
      // if an odd row //
      else
      {
         // return the row height for the pgn move table to adjust for text annotations //
         return min(CGFloat(PGNMovesData.RowHeight(TableRow / 2)), 660)
      }
   }
   // if is the game info table view //
   else if TableView == GameInfoTableView
   {
      // return the row height for the game info to adjust to string size //
      return CGFloat(GameInfoData.RowHeight(Row))
   }
   // if is the game list table view //
   else if TableView == GameListTableView
   {
      // return the row height for the game list //
      return CGFloat(GameListData.RowHeight(Row))
   }
   // not a valid table view //
   else
   {
      // return a row height of 30 //
      return 30
   }
}

// reload the selected and previously selected data rows //
func ReloadDataRows()
{
   // if not searching the game list //
   if !SearchingGameList
   {
      // return since not searching the game list //
      return
   }

   // store the currently selected row in the game list data //
   GameListData.StoreSelectedRow(GameListTableView.selectedRow)

   // reload the data for the previously selected game list data //
   GameListTableView.reloadData(forRowIndexes:IndexSet(integer:PreviouslySelectedRow),
                                columnIndexes:IndexSet(integer:0))

   // reload the data for the selected row //
   GameListTableView.reloadData(forRowIndexes:IndexSet(integer:GameListTableView.selectedRow),
                                columnIndexes:IndexSet(integer:0))

   // save the previously selected row //
   PreviouslySelectedRow = GameListTableView.selectedRow
}

// if the table view selection is changing //
func tableViewSelectionIsChanging(_ TheNotification:Notification)
{
   // reload the selected and previously selected row data //
   ReloadDataRows()
}

// if the table view selection did change //
func tableViewSelectionDidChange(_ TheNotification:Notification)
{
   // reload the selected and previously selected row data //
   ReloadDataRows()
}

// if the mouse entered the tracking rectangle //
override func mouseEntered(with Event:NSEvent)
{
   // update the cursor //
   UpdateCursor(Event.locationInWindow, Moving:false)
}

// if the mouse exited the tracking rectangle //
override func mouseExited(with Event:NSEvent)
{
   // set the cursor back to the arrow cursor //
   NSCursor.arrow.set()
}

// handle application being clicked on in dock //
func applicationShouldHandleReopen(_ Application:NSApplication, hasVisibleWindows Flag:Bool) ->
                                   Bool
{
   // make the main window visible //
   MainWindow.makeKeyAndOrderFront(nil)

   // return that main window was made visible //
   return true
}

// undo the draw path //
@IBAction func UndoPath(_:AnyObject)
{
   // play the game sound of a click //
   PlayGameSoundThread("click")

   // if there is a shape to undo //
   if Shapes.count > 0
   {
      // remove the shape layer from the super layer //
      Shapes[Shapes.count - 1].removeFromSuperlayer()

      // remove most current shape from the shape list //
      Shapes.removeLast()
   }
}

// clear the diagram the user drew on //
@IBAction func ClearDiagram(_:AnyObject)
{
   // play the sound of the diagram being cleared //
   PlayGameSoundThread("clear")

   // loop over all the shapes //
   for Index in 0 ..< Shapes.count
   {
      // remove the shape from the super layer //
      Shapes[Index].removeFromSuperlayer()
   }

   // remove all the shapes from the shape list //
   Shapes.removeAll()
}

// delegate to handle user clicking on pgn file to open application //
func application(_ Application:NSApplication, openFile Filename:String) -> Bool
{
   // make the main window visible //
   MainWindow.makeKeyAndOrderFront(nil)

   // if don't want to open file under these conditions //
   if HaveAttachedSheet() || InPositionEditor || (MovingPiece != 0) || EngineThinking
      || CoreAnimating || PlayingTimedGame || PlayingEngineMatch || InPanel
   {
      // exit function since don't want to open game file under these conditions //
      return true
   }

   // assign the game filename //
   GameFilename = Filename

   // convert the game filename into its components //
   FilenameComponents = GameFilename.components(separatedBy:".")

   // if the filename has a document file type extension //
   if FilenameComponents.count >= 2
   {
      // get the filename extension type //
      FilenameExtension = FilenameComponents[FilenameComponents.count - 1]

      // if the file is not a pgn file //
      if FilenameExtension.lowercased() != "pgn"
      {
         // return that can't open this type of file format //
         return false
      }
   }
   // filename doesn't have a document extension type //
   else
   {
      // return that can't open this type of file format //
      return false
   }

   // if there are no moves in the current game //
   if MoveList.isEmpty
   {
      // open the user selected game file //
      OpenGame()
   }
   // there are moves in the current game //
   else
   {
      // set that performing query to see if user wants to open game file //
      QueryOpenGameFile = true

      // assign the alert buttons //
      AlertButtons = ["Yes", "No"]

      // make the alert buttons //
      MakeAlert(AlertButtons, Query:"Open Game File?")
   }

   // return that done with function //
   return true
}

// finish up a game review //
func FinishUpGameReview()
{
   // set that no longer reviewing the game //
   ReviewingGame = false

   // set that no longer core animating a move //
   CoreAnimating = false

   // set the moving piece to none //
   MovingPiece = 0

   // turn on the game explorer buttons //
   SetGameExplorerButtons(true)
}

// validate a toolbar item //
override func validateToolbarItem(_ Item:NSToolbarItem) -> Bool
{
   // return the toolbar item enabled state //
   return Item.isEnabled
}

// set the game explorer buttons to the given state //
func SetGameExplorerButtons(_ State:Bool)
{
   // if the state is true //
   if State
   {
      // set the game review button to the play icon //
      GameReviewButton.image = PlayIcon
   }
   // the state is false //
   else
   {
      // set the game review button to the pause icon //
      GameReviewButton.image = PauseIcon
   }

   // set the game review button to the given state //
   GameReviewButton.isEnabled = State

   // set the enter analysis mode button to the given state //
   EnterAnalysisModeButton.isEnabled = State

   // set the enter position editor mode button to the given state //
   EnterPositionEditorModeButton.isEnabled = State

   // set the enter diagram mode button to the given state //
   EnterDiagramModeButton.isEnabled = State

   // set the game explorer snapshot button to the given state //
   GameExplorerSnapshotButton.isEnabled = State

   // set the game explorer game to movie button to the given state //
   GameExplorerMovieButton.isEnabled = State

   // set the exit game explorer button to the given state //
   ExitGameExplorerButton.isEnabled = State

   // set the game explorer slider to the given state //
   GameExplorerSlider.isEnabled = State

   // set the game explorer left button to the given state //
   GameExplorerLeft.isEnabled = State

   // set the game explorer right button to the given state //
   GameExplorerRight.isEnabled = State
}

// run the core animation of piece moving on the board //
```

```
// run the core animation of piece moving on the board //
func RunCoreAnimation()
{
   // if currently reviewing a game //
   if ReviewingGame
   {
      // check if move list index hit the maximum value allowed //
      if Index == MoveList.count
      {
         // finish up the game review //
         FinishUpGameReview()

         // exit function since no longer reviewing the game //
         return
      }

      // get the move from the move list to animate //
      UCIMove = MoveList[Index]

      // increment the index //
      Index += 1

      // increment the ply count //
      PlyCount += 1
   }

   // set that currently core animating //
   CoreAnimating = true

   // if not undoing an illegal move //
   if !UndoingIllegalMove
   {
      // make the move on the board //
      GCMakeMoveOnBoard(UCIMove)

      // if move does not exist //
      if !MoveExists
      {
         // finish up the game review since move does not exist //
         FinishUpGameReview()

         // exit animation since move does not exist //
         return
      }

      // convert the UCI move to origin and destination squares //
      ProcessUCIMove()
   }

   // set the animation path and duration //
   SetAnimationPathAndDuration()

   // add the animation to the animation layer //
   AnimationLayer.add(MoveAnimation, forKey:"position")
}

// set the animation path and duration //
func SetAnimationPathAndDuration()
{
   // create an animation layer //
   AnimationLayer = CALayer()

   // set the animation layer bounds to the square size //
   AnimationLayer.bounds = CGRect(x:0, y:0, width:CGFloat(SquareSize), height:
                                  CGFloat(SquareSize))

   // set the animation layer shadow opacity to 1 for shadow effect //
   AnimationLayer.shadowOpacity = 1

   // add the core animation layer to the nsview //
   self.layer!.addSublayer(AnimationLayer)

   // add the piece image to the animation layer //
   AnimationLayer.contents = GetPieceImage(MovingPiece)

   // if the board was rotated //
   if RotateBoard
   {
      // set animation starting x value //
      AnimStart.x = (7 - StartSquare.x) * CGFloat(SquareSize) + CGFloat(SquareOffset) - 0.5

      // set animation starting y value //
      AnimStart.y = (7 - StartSquare.y) * CGFloat(SquareSize) + CGFloat(SquareOffset) - 0.5
   }
   // board is not rotated //
   else
   {
      // set animation starting x value //
      AnimStart.x = StartSquare.x * CGFloat(SquareSize) + CGFloat(SquareOffset) - 0.5

      // set animation starting y value //
      AnimStart.y = StartSquare.y * CGFloat(SquareSize) + CGFloat(SquareOffset) - 0.5
   }

   // if board was rotated //
   if RotateBoard
   {
      // set animation ending x value //
      AnimEnd.x = (7 - EndSquare.x) * CGFloat(SquareSize) + CGFloat(SquareOffset) - 0.5

      // set animation ending y value //
      AnimEnd.y = (7 - EndSquare.y) * CGFloat(SquareSize) + CGFloat(SquareOffset) - 0.5
   }
   // board was not rotated //
   else
   {
      // set animation ending x value //
      AnimEnd.x = EndSquare.x * CGFloat(SquareSize) + CGFloat(SquareOffset) - 0.5

      // set animation ending y value //
      AnimEnd.y = EndSquare.y * CGFloat(SquareSize) + CGFloat(SquareOffset) - 0.5
   }

   // create the animation fromvalue //
   MoveAnimation.fromValue = NSValue(point:AnimStart)

   // create the animation tovalue //
   MoveAnimation.toValue = NSValue(point:AnimEnd)

   // set the animation final position //
   AnimationLayer.position = NSPointToCGPoint(AnimEnd)

   // if undoing an illegal move //
   if UndoingIllegalMove
   {
      // set the animation duration to 0.25 seconds //
      MoveAnimation.duration = 0.25
   }
   // move is legal //
   else
   {
      // set the animation duration //
      MoveAnimation.duration = CFTimeInterval(sqrt(pow(Float(EndSquare.x - StartSquare.x), 2) +
                                              pow(Float(EndSquare.y - StartSquare.y), 2)) *
                                              Float(101 - AnimationRate) / 50.0)
   }
}

// animation did start delegate //
func animationDidStart(_ TheAnimation:CAAnimation)
{
   // update the board and captured pieces //
   UpdateBoardAndCapturedPieces()
}

// animation did stop delegate //
func animationDidStop(_ TheAnimation:CAAnimation, finished Flag:Bool)
{
   // remove the animation layer from the super layer //
   AnimationLayer.removeFromSuperlayer()

   // check if move was legal and had a promotion //
   if !UndoingIllegalMove && PromotionType != 0
   {
      // set the piece moving to the type of piece pawn promoted to //
      MovingPiece = PromotionType
   }

   // put the piece that moved onto the destination square //
   UpdateDestinationSquare()

   // update the board and captured pieces //
   UpdateBoardAndCapturedPieces()

   // play the sound of a piece thumping on the board //
   PlayGameSoundThread("thump")

   // if undoing an illegal move //
   if UndoingIllegalMove
   {
      // set that no longer undoing an illegal move //
      UndoingIllegalMove = false

      // set that no longer core animating //
      CoreAnimating = false

      // replay all the moves //
      ReplayMoves(0, End:MoveList.count, Display:false)
   }
   // doing a legal move //
   else
   {
      // perform en passant move and capture if needed //
      PerformEnpassantMoveIfNeeded()

      // perform castling move if had one //
      PerformCastleIfNeeded()

      // update the pieces layer for an illustration frame //
      UpdateIllustrationPiecesLayer()

      // update the board and captured pieces //
      UpdateBoardAndCapturedPieces()

      // if reviewing a game //
      if ReviewingGame
      {
         // update the game explorer slider value to the current move location //
         GameExplorerSlider.intValue = Int32(Index)

         // print the game information //
         PrintGameInfo()

         // if want to pause the game review //
         if PauseGameReview
         {
            // set that game no longer paused //
            PauseGameReview = false

            // finish up the game review //
            FinishUpGameReview()
         }
         // not pausing game review //
         else
         {
            // run core animation for the next move //
            RunCoreAnimation()
         }
      }
      // not reviewing a game //
      else
      {
         // set that no longer core animating //
         CoreAnimating = false

         // update the board and captured pieces //
         UpdateBoardAndCapturedPieces()

         // set that the chess engine is done thinking //
         EngineThinking = false

         // print the game information //
         PrintGameInfo()
      }
   }

   // if playing an engine match //
   if PlayingEngineMatch
   {
      // make an engine match move //
      MakeEngineMatchMove()
   }
}

// make an engine match move //
func MakeEngineMatchMove()
{
   // if the game is over //
   if BoardGameOver(Board)
   {
      // set that done playing an engine match //
      PlayingEngineMatch = false

      // turn off the engine match menu item //
      EngineMatchMenuItem.state = NSControl.StateValue(rawValue:0)

      // exit function since game is over and no longer playing an engine match //
      return
   }
   // game is not over //
   else
   {
      // set that currently playing an engine match //
      PlayingEngineMatch = true
   }

   // start an engine move //
   StartEngineMove()
}

// override mouse down can move window //
override var mouseDownCanMoveWindow:Bool
{
   // return that a mouse down can't move the window //
   return false
}

// the hide info panel menu item action //
@IBAction func HideInfoMenuItemAction(_:AnyObject)
{
   // add a timer to run the hide info menu item action //
   RunLoop.current.add(Timer(timeInterval:0.01, target:self, selector:#selector(GreenChess
                       .TheHideInfoMenuItemAction), userInfo:nil, repeats:false), forMode:
                       RunLoop.Mode.commonModes)
}

// the hide info menu item action //
@objc func TheHideInfoMenuItemAction()
{
   // hide the info panel //
   InfoPanelAction(true, Small:false, Large:false)
}

// the small info panel menu item action //
@IBAction func SmallInfoMenuItemAction(_:AnyObject)
{
   // add a timer to run the small info menu item action //
   RunLoop.current.add(Timer(timeInterval:0.01, target:self, selector:#selector(GreenChess
                       .TheSmallInfoMenuItemAction), userInfo:nil, repeats:false), forMode:
                       RunLoop.Mode.commonModes)
}

// the small info menu item action //
@objc func TheSmallInfoMenuItemAction()
{
   // set the info panel size to small //
   InfoPanelAction(false, Small:true, Large:false)
}

// the large info panel menu item action //
@IBAction func LargeInfoMenuItemAction(_:AnyObject)
{
   // add a timer to run the large info menu item action //
   RunLoop.current.add(Timer(timeInterval:0.01, target:self, selector:#selector(GreenChess
                       .TheLargeInfoMenuItemAction), userInfo:nil, repeats:false), forMode:
                       RunLoop.Mode.commonModes)
}

// the large info menu item action //
@objc func TheLargeInfoMenuItemAction()
{
   // set the info panel size to large //
   InfoPanelAction(false, Small:false, Large:true)
}

// hide the captured pieces in the panel menu item action //
@IBAction func HideCapturedPiecesInPanelMenuItemAction(_:AnyObject)
{
   // add a timer to run the hide captured pieces in panel menu item action //
   RunLoop.current.add(Timer(timeInterval:0.01, target:self, selector:#selector(GreenChess
                       .TheHideCapturedPiecesInPanelMenuItemAction), userInfo:nil, repeats:
                       false), forMode:RunLoop.Mode.commonModes)
}

// the hide captured pieces in panel menu item action //
@objc func TheHideCapturedPiecesInPanelMenuItemAction()
{
   // toggle the hide captured pieces in panel //
   HideCapturedPiecesInPanel = !HideCapturedPiecesInPanel

   // hide or unhide the captured pieces in the panel //
   HideCapturedPiecesInPanelAction()

   // update the main window //
   UpdateMainWindow()
}

// hide the game clock in panel menu item action //
@IBAction func HideGameClockInPanelMenuItemAction(_:AnyObject)
{
   // add a timer to run the hide game clock in panel menu item action //
   RunLoop.current.add(Timer(timeInterval:0.01, target:self, selector:#selector(GreenChess
                       .TheHideGameClockInPanelMenuItemAction), userInfo:nil, repeats:false),
                       forMode:RunLoop.Mode.commonModes)
}

// the hide game clock in panel menu item action //
@objc func TheHideGameClockInPanelMenuItemAction()
{
   // toggle the hide game clock in panel //
   HideGameClockInPanel = !HideGameClockInPanel

   // hide or unhide the game clock in the panel //
   HideGameClockInPanelAction()

   // update the main window //
   UpdateMainWindow()
}

// hide the captured pieces in the panel action //
func HideCapturedPiecesInPanelAction()
{
   // update the captured pieces white view hidden state //
   CapturedPiecesWhiteView.isHidden = HideCapturedPiecesInPanel

   // update the captured pieces black view hidden state //
   CapturedPiecesBlackView.isHidden = HideCapturedPiecesInPanel

   // update the hide captured pieces in panel preference //
   AppPreferences.set(HideCapturedPiecesInPanel, forKey:"HideCapturedPiecesInPanel")

   // update the hide captured pieces in panel menu item state //
   HideCapturedPiecesInPanelMenuItem.state = NSControl.StateValue(rawValue:
                                                   BoolToInt(!HideCapturedPiecesInPanel))
}

// hide the game clock in panel action //
func HideGameClockInPanelAction()
{
   // update the game clock hidden state //
   GameClock.isHidden = HideGameClockInPanel

   // update the hide game clock in panel preference //
   AppPreferences.set(HideGameClockInPanel, forKey:"HideGameClockInPanel")

   // update the hide game clock in panel menu item state //
   HideGameClockInPanelMenuItem.state = NSControl.StateValue(rawValue:
                                                   BoolToInt(!HideGameClockInPanel))
}

// the info panel action //
func InfoPanelAction(_ Hide:Bool, Small:Bool, Large:Bool)
{
   // update the hide info panel state //
   HideInfoPanel = Hide

   // update the small info panel state //
   SmallInfoPanel = Small

   // update the large info panel state //
   LargeInfoPanel = Large

   // update the main window //
   UpdateMainWindow()
}

// the window should close delegate //
func windowShouldClose(_ Window:NSWindow) -> Bool
{
   // if is the main window //
   if Window == MainWindow
   {
      // set that should close the main window //
      ShouldCloseMainWindow = true
   }

   // return that should close the main window //
   return true
}

// the custom windows to use when entering full screen //
func customWindowsToEnterFullScreen(for Window:NSWindow) -> [NSWindow]?
{
   // return the main window //
   return [MainWindow]
}

// start the custom animation when entering full screen //
func window(_ Window:NSWindow, startCustomAnimationToEnterFullScreenWithDuration Duration:
            TimeInterval)
{
}

// the custom windows to use when exit full screen //
func customWindowsToExitFullScreen(for Window:NSWindow) -> [NSWindow]?
{
   // return the main window //
   return [MainWindow]
}

// start the custom animation when exiting full screen //
func window(_ Window:NSWindow, startCustomAnimationToExitFullScreenWithDuration Duration:
            TimeInterval)
{
}

// update the main window //
func UpdateMainWindow()
{
   // clear the pgn moves list //
   PGNMovesList.removeAll()

   // update the move list //
   UpdateMoveList()

   // clear the game info list //
   GameInfoList.removeAll()

   // update the game info list //
   UpdateGameInfoList()

   // set the hide info menu item state //
   HideInfoMenuItem.state = NSControl.StateValue(rawValue:BoolToInt(HideInfoPanel))

   // update the large info menu item state //
   LargeInfoMenuItem.state = NSControl.StateValue(rawValue:BoolToInt(LargeInfoPanel))

   // update the small info menu item state //
   SmallInfoMenuItem.state = NSControl.StateValue(rawValue:BoolToInt(SmallInfoPanel))

   // save the hide info panel value to preferences //
   AppPreferences.set(HideInfoPanel, forKey:"HideInfoPanel")

   // save the small info panel value to preferences //
   AppPreferences.set(SmallInfoPanel, forKey:"SmallInfoPanel")

   // save the large info panel value to preferences //
   AppPreferences.set(LargeInfoPanel, forKey:"LargeInfoPanel")

   // set the window minimum size //
   WindowMinSize = self.frame.size

   // set the window maximum size //
   WindowMaxSize = NSMakeSize(CGFloat(660 + BoolToInt(!HideInfoPanel) *
                              BoolToInt(SmallInfoPanel) * 340 + BoolToInt(!HideInfoPanel) *
                              BoolToInt(LargeInfoPanel) * 600), 660)

   // set the main window content minimum size //
   MainWindow.contentMinSize = WindowMinSize

   // set the main window content maximum size //
   MainWindow.contentMaxSize = WindowMaxSize

   // set the main window fullsize content minimum size //
   MainWindow.minFullScreenContentSize = WindowMinSize

   // set the main window fullsize content maximum size //
   MainWindow.maxFullScreenContentSize = WindowMaxSize

   // if not in full screen mode //
   if !FullScreen
   {
      // set the main window content size //
      MainWindow.setContentSize(WindowMaxSize)
   }

   // get the current hide game clock in panel value //
   HideClock = CGFloat(BoolToInt(HideGameClockInPanel))

   // get the current hide pieces in panel value //
   HidePieces = CGFloat(BoolToInt(HideCapturedPiecesInPanel))

   // if have a small info panel //
   if !HideInfoPanel && SmallInfoPanel
   {
      // update the game clock view frame //
      GameClock.frame = NSMakeRect(674, 525, 310, 130)

      // set the captured white pieces size //
      CapturedPiecesWhiteView.Width = 20

      // update the captured pieces white view frame //
      CapturedPiecesWhiteView.frame = NSMakeRect(692, 500 + HideClock * 120, 308, 20)

      // set the captured black pieces size //
      CapturedPiecesBlackView.Width = 20

      // update the captured pieces black view frame //
      CapturedPiecesBlackView.frame = NSMakeRect(692, 480 + HideClock * 120, 308, 20)

      // update the pgn moves table view frame //
      PGNMovesTableView.enclosingScrollView?.frame = NSMakeRect(692, 30, 275, 215 + HideClock *
                                                               60 + HidePieces * 20)

      // update the game info table view frame //
      GameInfoTableView.enclosingScrollView?.frame = NSMakeRect(692, 255 + HideClock * 60 +
                                                               HidePieces * 20, 275, 215 +
                                                               HideClock * 60 + HidePieces * 20)
   }
   // if have a large info panel //
   else if !HideInfoPanel && LargeInfoPanel
   {
      // update the game clock view frame //
      GameClock.frame = NSMakeRect(795, 525, 310, 130)

      // update the captured white pieces size //
      CapturedPiecesWhiteView.Width = 35

      // update the captured pieces white view frame //
      CapturedPiecesWhiteView.frame = NSMakeRect(670, 485 + HideClock * 120, 590, 35)

      // update the captured black pieces size //
      CapturedPiecesBlackView.Width = 35

      // update the captured pieces black view frame //
      CapturedPiecesBlackView.frame = NSMakeRect(670, 450 + HideClock * 120, 590, 35)

      // update the pgn moves table view frame //
      PGNMovesTableView.enclosingScrollView?.frame = NSMakeRect(670, 30, 275, 410 + HideClock *
                                                               120 + HidePieces * 70)

      // update the game info table view frame //
      GameInfoTableView.enclosingScrollView?.frame = NSMakeRect(955, 30, 275, 410 + HideClock *
                                                               120 + HidePieces * 70)
   }

   // clear the tables //
   ClearTables()

   // set the pgn moves table view background color to show main window color //
   PGNMovesTableView.backgroundColor = NSColor(red:1, green:1, blue:1, alpha:0.4)

   // set the game info table view background color to show main window color //
   GameInfoTableView.backgroundColor = NSColor(red:1, green:1, blue:1, alpha:0.4)

   // make the main window key and order front //
   MainWindow.makeKeyAndOrderFront(nil)
}

// update the sheet sizes //
func UpdateSheetSizes()
{
   // if don't need to resize the sheets //
   if GameExplorerWindow.frame.size.width == MainWindow.frame.size.width
   {
      // exit since don't need to resize the sheets //
      return
   }

   // set the gallery view frame width //
   GameGalleryView.FrameWidth = Double(MainWindow.frame.size.width) / Double(Int(MainWindow
                                .frame.size.width / 220))

   // set the grid view frame width //
   GameGridView.FrameWidth = GameGalleryView.FrameWidth

   // set size of the game explorer window to fit the size of the main window //
   GameExplorerWindow.setContentSize(NSMakeSize(MainWindow.frame.size.width, 22))

   // set size of the database explorer window to fit the size of main window //
   DatabaseExplorerWindow.setContentSize(NSMakeSize(MainWindow.frame.size.width, 25))

   // set the size of the game gallery window to fit the size of the main window //
   GameGalleryWindow.setContentSize(NSMakeSize(MainWindow.frame.size.width,
                                    CGFloat(GameGalleryView.FrameWidth) + 25))

   // set the size of the game grid window to fit the size of the main window //
   GameGridWindow.setContentSize(NSMakeSize(MainWindow.frame.size.width,
                                 CGFloat(GameGridView.FrameWidth) * 2 + 25))

   // set size of the game explorer slider to fit the size of the main window //
   GameExplorerSlider.setFrameSize(NSMakeSize(MainWindow.frame.size.width - 30, 12))

   // set the frame of the game gallery slider to fit the size of the main window //
   GameGallerySlider.frame = NSMakeRect(15, 6, MainWindow.frame.size.width - 30, 13)

   // set the frame of the game grid slider to fit the size of the main window //
   GameGridSlider.frame = NSMakeRect(15, 6, MainWindow.frame.size.width - 30, 13)

   // set size of the database explorer slider to fit size of main window //
   DatabaseExplorerSlider.setFrameSize(NSMakeSize(MainWindow.frame.size.width - 30, 12))

   // set size of database game explorer slider to fit size of main window //
   DatabaseGameExplorerSlider.setFrameSize(NSMakeSize(MainWindow.frame.size.width - 100, 12))

   // set the frame of the game gallery view to fit the size of the main window //
   GameGalleryView.frame = NSMakeRect(0, 25, MainWindow.frame.size.width,
                                      CGFloat(GameGalleryView.FrameWidth))

   // set the frame of the game grid view to fit the size of the main window //
   GameGridView.frame = NSMakeRect(0, 25, MainWindow.frame.size.width, CGFloat(GameGridView
                                   .FrameWidth) * 2)
}

// intercept a key down event //
func InterceptKeyDown(_ Event:NSEvent)
{
   // if key pressed was the up or right arrow //
   if Event.keyCode == 126 || Event.keyCode == 124
   {
      // if the database explorer window is visible //
      if IsAttached(DatabaseExplorerWindow)
      {
         // go right in the database explorer //
         DatabaseExplorerRightAction(self)
      }
      // if the game explorer window is visible and not currently reviewing a game //
      else if IsAttached(GameExplorerWindow) && !ReviewingGame
      {
         // go right in the game explorer //
         GameExplorerRightAction(self)
      }
      // if the load position window is visible //
      else if IsAttached(LoadPositionWindow)
      {
         // go to the right in the load position window view //
         PositionButtonRightAction(self)
      }
      // if the game gallery or grid window is visible and not reviewing //
      else if (IsAttached(GameGalleryWindow) || IsAttached(GameGridWindow)) &&
              !GalleryGridView.Reviewing
      {
         // go right one frame in the game gallery or grid //
         GameGalleryGridRightAction(self)
      }
   }
   // if key pressed was the down or left arrow //
   else if Event.keyCode == 125 || Event.keyCode == 123
   {
      // if the database explorer window is visible //
      if IsAttached(DatabaseExplorerWindow)
      {
         // go left in the database explorer //
         DatabaseExplorerLeftAction(self)
      }
      // if the game explorer window is visible and not currently reviewing a game //
      else if IsAttached(GameExplorerWindow) && !ReviewingGame
      {
         // go left in the game explorer //
         GameExplorerLeftAction(self)
      }
      // if the load position window is visible //
      else if IsAttached(LoadPositionWindow)
      {
         // go to the left in the load position window view //
         PositionButtonLeftAction(self)
      }
      // if the game gallery or grid window is visible and not reviewing //
      else if (IsAttached(GameGalleryWindow) || IsAttached(GameGridWindow)) &&
              !GalleryGridView.Reviewing
      {
         // go left one frame in the game gallery or grid //
         GameGalleryGridLeftAction(self)
      }
   }
   // if the spacebar was pressed //
   else if Event.keyCode == 49
   {
      // if in game explorer //
      if IsAttached(GameExplorerWindow)
      {
         // review the game //
         ReviewGame(self)
      }
      // if the game gallery or grid window is visible //
      else if IsAttached(GameGalleryWindow) || IsAttached(GameGridWindow)
      {
         // review the gallery grid view //
         ReviewGalleryGrid(self)
      }
   }
}

// initialize all the data sources for the table views //
func InitializeDataSources()
{
   // set the data source for the pgn moves table view //
   PGNMovesTableView.dataSource = PGNMovesData

   // set the data source for the game list //
   GameListTableView.dataSource = GameListData

   // set the game info data source //
   GameInfoTableView.dataSource = GameInfoData
}

// return if these specific menu items //
func TheseMenuItems(_ MenuItem:NSMenuItem) -> Bool
{
   // return if these specific menu items //
   return MenuItem == HideInfoMenuItem || MenuItem == SmallInfoMenuItem || MenuItem ==
          LargeInfoMenuItem || MenuItem == RotateBoardMenuItem || MenuItem ==
          BoardSnapshotMenuItem || MenuItem == HideGameClockInPanelMenuItem || MenuItem ==
          HideCapturedPiecesInPanelMenuItem
}

// remove the extra edit items //
@objc func RemoveEditItems()
{
   // assign the edit menu //
   let EditMenu = NSApplication.shared.mainMenu!.item(withTitle:"Edit")

   // if the edit menu exists //
   if EditMenu != nil
   {
      // if there are fourteen edit menu items //
      if EditMenu!.submenu!.numberOfItems == 14
      {
         // loop over the edit menu items to remove //
         for _ in 1 ... 3
         {
            // remove the extra edit menu items //
            EditMenu!.submenu!.removeItem(at:11)
         }
      }
   }
}

// clear the selected square if need to //
func ClearSelectedSquareIfNeeded()
{
   // if moving by selecting squares and have selected squares //
   if MoveBySelectingSquares && SelectedSquares > 0
   {
      // reset the moving piece to zero //
      MovingPiece = 0

      // reset the valid piece to false //
      ValidPiece = false

      // reset the selected squares to zero //
      SelectedSquares = 0

      // replay all the moves //
      ReplayMoves(0, End:MoveList.count, Display:false)
   }
}

// validate whether the corresponding menu item should be active or inactive //
override func validateMenuItem(_ MenuItem:NSMenuItem) -> Bool
{
   // clear the selected square if needed //
   ClearSelectedSquareIfNeeded()

   // if is the gostop menu item //
   if MenuItem == GoStopMenuItem
   {
      // if the engine is currently thinking //
      if EngineThinking
      {
         // set go stop menu item so user can stop the current engine move //
         GoStopMenuItem.title = "Stop"
      }
      // if the engine is not currently thinking //
      else
      {
         // set go stop menu item title so user can start an engine move //
         GoStopMenuItem.title = "Engine"
      }
   }

   // if the main window is not visible //
   if !MainWindow.occlusionState.contains(NSWindow.OcclusionState.visible)
   {
      // set the hide menu item title to reopen //
      HideMenuItem.title = "Reopen"
   }
   // the main window is visible //
   else
   {
      // set the hide menu item title to hide //
      HideMenuItem.title = "Hide"
   }

   // if is the info panel or hide menu item and in full screen //
   if (MenuItem == HideInfoMenuItem || MenuItem == SmallInfoMenuItem || MenuItem ==
      LargeInfoMenuItem || MenuItem == HideMenuItem) && FullScreen
   {
      // return that game info menu item is not valid //
      return false
   }
   // if is the hide menu item //
   else if MenuItem == HideMenuItem
   {
      // return that hide menu item is always valid //
      return true
   }
   // if in a panel or moving a piece or core animating and not reviewing game //
   else if ThreadIsRunning() || InPanel || (((MovingPiece != 0) || CoreAnimating) &&
           !ReviewingGame)
   {
      // return that all menu items are false except for the hide menu item //
      return false
   }
   // if the main window is not visible or have an attached sheet or quit or about item //
   else if !MainWindow.occlusionState.contains(NSWindow.OcclusionState.visible) ||
           HaveAttachedSheet() || MenuItem == QuitMenuItem || MenuItem == AboutMenuItem
   {
      // return that only the hide, quit, and about menu items are valid //
      return MenuItem == QuitMenuItem || MenuItem == AboutMenuItem
   }

   // if menu item matches a specific menu item //
   if (MenuItem == GoStopMenuItem && BoardGameOver(Board)) || (EngineThinking && MenuItem !=
      GoStopMenuItem) || (HumanOpponent && MenuItem == GoStopMenuItem) || (AnalyzingGame &&
      (MenuItem == TimedGameMenuItem || MenuItem == PositionEditorMenuItem || MenuItem ==
      SearchGameListMenuItem || MenuItem == OpenGameMenuItem || MenuItem ==
      DatabaseExplorerMenuItem || MenuItem == GameAnnotatorMenuItem)) || (Diagramming &&
      !(MenuItem == GameExplorerMenuItem || TheseMenuItems(MenuItem) || MenuItem ==
      PreferencesMenuItem)) || (PlayingTimedGame && !(MenuItem == TimedGameMenuItem ||
      TheseMenuItems(MenuItem) || MenuItem == SaveGameMenuItem || MenuItem == AppendGameMenuItem
      || MenuItem == PreferencesMenuItem)) || (InPositionEditor && !(TheseMenuItems(MenuItem) ||
      MenuItem == PreferencesMenuItem))
   {
      // return that menu item is not valid //
      return false
   }

   // return that menu item should be activated //
   return true
}

// make a move on the board //
func GCMakeMoveOnBoard(_ Move:String)
{
   // if were able to find the move //
   if BoardFindMove(Board, Move)
   {
      // make the move on the board //
      BoardMakeMove(Board, 0)

      // set that the move exists and was able to be found from possible moves //
      MoveExists = true
   }
   // if not able to find the move //
   else
   {
      // set that move does not exist //
      MoveExists = false
   }
}

// make a move on the data board //
func GCMakeMoveOnDataBoard(_ Move:String)
{
   // if were able to find the move //
   if BoardFindMove(GameDataBoard, Move)
   {
      // make the move on the data board //
      BoardMakeMove(GameDataBoard, 0)

      // set that the move exists and was able to be found from possible moves //
      MoveExists = true
   }
   // if not able to find the move //
   else
   {
      // set that move does not exist //
      MoveExists = false
   }
}

// make a chess engine move outside the uci chess engine //
@objc func GCEngineMove()
{
   // initialize the chess board with the initial fen string //
   BoardInitializeAttributes(Board, Attributes.FENLayout, Attributes.FENCastlingRights,
                             Attributes.FENColor, Attributes.FENEnpassantSquare)

   // loop over the current move list //
   for Index in 0 ..< MoveList.count
   {
      // make the move on the board //
      GCMakeMoveOnBoard(MoveList[Index])
   }

   // if is the very first move in a game from initial fen position //
   if MoveList.isEmpty && Attributes.FENLayout == DefaultFENLayout && Attributes.FENColor == "w"
      && !UseEcoBook
   {
      // set the first move for white to be a standard opening //
      UCIMove = "e2e4"
   }
   // not the very first move in a game from initial fen position //
   else
   {
      // if will be using the encyclopedia of chess openings book //
      if UseEcoBook
      {
         // wait until the entire chess opening book is done being read in //
         while ReadOpeningBookThread.isExecuting
         {
         }
      }

      // if a move exists for the current board position and using the opening book //
      if OpeningBookMoveExists(OpeningBook, BoardGetZobristHash(Board)) && UseEcoBook
      {
         // get the move from the chess opening book //
         UCIMove = String(validatingUTF8:OpeningBookGetNextMove(OpeningBook))!

         // if can't find the move from the opening book on the board //
         if !BoardFindMove(Board, UCIMove)
         {
            // get best chess move from the chess engine with max ply depth of 100 //
            UCIMove = String(validatingUTF8:EngineFindBestMove(Engine, Board,
                             CInt(DynamicSearchTime()), 100))!
         }
      }
      // not using the opening book or move not in the opening book //
      else
```

```
         else
         {
            // get the best chess move from the chess engine with max ply depth of 100 //
            UCIMove = String(validatingUTF8:EngineFindBestMove(Engine, Board,
                             Cint(DynamicSearchTime()), 100))!
         }
      }

      // add the uci move to the move list //
      MoveList.append(UCIMove)

      // add a placeholder annotation to the annotation list //
      AnnotationList.append("")

      // store the autosave data //
      StoreAutosaveData("")

      // perform an engine move on the main thread //
      DispatchQueue.main.async
      {
         // perform the engine move //
         self.PerformEngineMove()
      }
   }

// modify the search time based on how much time left to play in timed game or not //
func DynamicSearchTime() -> Int
{
   // if not playing a timed game //
   if !PlayingTimedGame
   {
      // return the search time //
      return SearchTime
   }

   // if playing a timed game against black //
   if TimedGameColor == 0
   {
      // get the time left in the game for the chess engine playing as black //
      TimeLeft = RemainingBlackTime
   }
   // playing a timed game against white //
   else
   {
      // get the time left in the game for the chess engine playing as white //
      TimeLeft = RemainingWhiteTime
   }

   // if less than five minutes left in the game //
   if TimeLeft < 3000
   {
      // divide the time left by twenty moves //
      return TimeLeft / 200 + 10
   }
   // not in time trouble yet //
   else
   {
      // divide game time by 60 moves //
      return TotalGameTime / 600
   }
}

// redo the last move that had been made or undone //
@IBAction func RedoLastMoveMenuItemAction(_:AnyObject)
{
   // add a timer to run the redo last move menu item action //
   RunLoop.current.add(Timer(timeInterval:0.01, target:self, selector:#selector(GreenChess
                       .TheRedoLastMoveMenuItemAction), userInfo:nil, repeats:false), forMode:
                       RunLoopMode.commonModes)
}

// the redo last move menu item action //
@objc func TheRedoLastMoveMenuItemAction()
{
   // if there are no moves in the previous move list //
   if PreviousMoveList.isEmpty
   {
      // exit function since there are no moves to redo in previous move list //
      return
   }

   // set the move list to the previous move list to restore the last move //
   MoveList = PreviousMoveList

   // set the annotation list to the previous annotation list to restore last annotations //
   AnnotationList = PreviousAnnotationList

   // replay the last move that the user wanted redone //
   ReplayMoves(0, End:MoveList.count, Display:false)

   // store the autosave data //
   StoreAutosaveData("")
}

// update the list of captured pieces //
func StoreCapturedPiece(_ CapturedPiece:Int)
{
   // if piece captured was a white queen //
   if CapturedPiece == 5
   {
      // increment that a white queen was captured //
      CapturedWhitePieces[0] += 1
   }
   // if piece captured was a white rook //
   else if CapturedPiece == 4
   {
      // increment that a white rook was captured //
      CapturedWhitePieces[1] += 1
   }
   // if piece captured was a white bishop //
   else if CapturedPiece == 3
   {
      // increment that a white bishop was captured //
      CapturedWhitePieces[2] += 1
   }
   // if piece captured was a white knight //
   else if CapturedPiece == 2
   {
      // increment that a white knight was captured //
      CapturedWhitePieces[3] += 1
   }
   // if piece captured was a white pawn by regular or en passant capture //
   else if (CapturedPiece == 1) || (BoardIsEnpassant(Board) && BoardIsWhite(Board))
   {
      // increment that a white pawn was captured //
      CapturedWhitePieces[4] += 1
   }
   // if piece captured was a black queen //
   else if CapturedPiece == -5
   {
      // increment that a black queen was captured //
      CapturedBlackPieces[0] += 1
   }
   // if piece captured was a black rook //
   else if CapturedPiece == -4
   {
      // increment that a black rook was captured //
      CapturedBlackPieces[1] += 1
   }
   // if piece captured was a black bishop //
   else if CapturedPiece == -3
   {
      // increment that a black bishop was captured //
      CapturedBlackPieces[2] += 1
   }
   // if piece captured was a black knight //
   else if CapturedPiece == -2
   {
      // increment that a black knight was captured //
      CapturedBlackPieces[3] += 1
   }
   // if piece captured was a black pawn by regular or en passant capture //
   else if (CapturedPiece == -1) || (BoardIsEnpassant(Board) && BoardIsBlack(Board))
   {
      // increment that a black pawn was captured //
      CapturedBlackPieces[4] += 1
   }
}

// update the destination square that the piece moves to //
func UpdateDestinationSquare()
{
   // not waiting for pawn promotion or undoing illegal move //
   if !QueryPawnPromotion && !UndoingIllegalMove
   {
      // store the captured piece if any //
      StoreCapturedPiece(Layout[Int(EndSquare.x)][Int(EndSquare.y)])
   }

   // put the piece that moved onto the destination square //
   Layout[Int(EndSquare.x)][Int(EndSquare.y)] = MovingPiece

   // reset the piece moving to be none //
   MovingPiece = 0
}

// promote a pawn if need to //
func PromotePawnIfNeeded()
{
   // if piece isn't white pawn at eighth rank or not black pawn at first rank //
   if !((WhitePawnIsMoving() && UCIMove.last! == "8") || (BlackPawnIsMoving() && UCIMove.last!
      == "1"))
   {
      // exit the function early since a promotion didn't occur //
      return
   }

   // update the moving piece to the new location on the board //
   UpdateDestinationSquare()

   // update the chess board //
   UpdateBoardAndCapturedPieces()

   // set that currently querying for a pawn promotion //
   QueryPawnPromotion = true

   // create the needed alert buttons //
   AlertButtons = ["Queen", "Rook", "Bishop", "Knight"]

   // play the piece thump sound thread //
   PlayGameSoundThread("thump")

   // alert the user as to which type of piece they want to promote pawn to //
   MakeAlert(AlertButtons, Query:"Promote Pawn To")
}

// make an alert from the given buttons and the user query message //
func MakeAlert(_ Buttons:[String], Query:String)
{
   // if already in a panel //
   if InPanel
   {
      // exit function since already in a panel //
      return
   }

   // close the attached sheet if there is one //
   CloseAttachedSheet()

   // create the user alert //
   Alert = NSAlert()

   // loop over all the alert buttons //
   for Index in 0 ..< Buttons.count
   {
      // add the button to the alert panel //
      Alert.addButton(withTitle:Buttons[Index])
   }

   // set the alert query message //
   Alert.messageText = Query

   // if querying to quit the application //
   if QueryQuitApplication
   {
      // set the alert icon to the message icon //
      Alert.icon = MessageIcon
   }
   // if currently in the position editor //
   else if InPositionEditor
   {
      // if the fen layout is legal for the board position //
      if FENLayoutIsLegal()
      {
         // set the alert icon to show position is validated //
         Alert.icon = ValidatedIcon
      }
      // the fen layout is illegal //
      else
      {
         // set the alert icon to show position is invalid //
         Alert.icon = WarningIcon
      }
   }
   // if querying whether uci chess engine was loaded or not //
   else if QueryUCIEngineLoad
   {
      // if uci chess engine loaded successfully //
      if UCIEngineLoaded
      {
         // set alert icon to show that uci chess engine loaded successfully //
         Alert.icon = ValidatedIcon
      }
      // uci chess engine failed to load //
      else
      {
         // set alert icon to show that uci chess engine did not load successfully //
         Alert.icon = WarningIcon
      }
   }
   // if query save position //
   else if QuerySavePosition
   {
      // set the snapshot image to the current board //
      Alert.icon = SnapshotImage
   }
   // if loading the saved position which doesn't currently exist //
   else if QueryLoadPosition
   {
      // set the alert icon //
      Alert.icon = WarningIcon
   }
   // is a default query //
   else
   {
      // set the alert icon to show a message icon //
      Alert.icon = MessageIcon
   }

   // run the alert and reopen the previously attached sheet if there was one //
   Alert.beginSheetModal(for:MainWindow, completionHandler:{Result -> Void in
                         self.ReturnCode = Result.rawValue; self.ProcessAlert()
                         self.ReopenPreviouslyAttachedSheet()})
}

// return whether the specified window is attached or not //
func IsAttached(_ Window:NSWindow) -> Bool
{
   // if the main window has an attached sheet //
   if MainWindow.attachedSheet != nil
   {
      // return whether the specified window is attached or not //
      return MainWindow.attachedSheet! == Window
   }
   // the main window doesn't have an attached sheet //
   else
   {
      // return that there is no attached sheet //
      return false
   }
}

// reset the game clock //
func ResetGameClock()
{
   // reset the game time for white //
   ElapsedWhiteTime = TimedGameOffset

   // reset the game time for black //
   ElapsedBlackTime = TimedGameOffset
}

// update the progress //
func UpdateProgress()
{
   // if animating the progress //
   if AnimateProgress
   {
      // disable the animation effect //
      CATransaction.setDisableActions(true)

      // if at the maximum opacity for the progress circle //
      if ProgressCircle.opacity >= 0.8
      {
         // set the progress direction to be down //
         ProgressDirectionUp = false
      }
      // if at minimum opacity for the progress circle //
      else if ProgressCircle.opacity <= 0.2
      {
         // set the progress direction to be up //
         ProgressDirectionUp = true
      }

      // if the progress direction is up //
      if ProgressDirectionUp
      {
         // increment the progress circle opacity by 0.05 //
         ProgressCircle.opacity += 0.05
      }
      // progress direction is down //
      else
      {
         // decrement the progress circle opacity by 0.05 //
         ProgressCircle.opacity -= 0.05
      }
   }
   // if not animating progress and progress circle opacity is nonzero //
   else if ProgressCircle.opacity != 0
   {
      // disable the animation effect //
      CATransaction.setDisableActions(true)

      // reset the progress circle opacity to zero //
      ProgressCircle.opacity = 0
   }
}

// update the game clock //
@objc func UpdateGameClock()
{
   // update the progress //
   UpdateProgress()

   // if querying user that timed game is over //
   if QueryTimedGameOver
   {
      // exit function and don't update the game clock //
      return
   }

   // if in position editor, diagramming, in gallery, in grid, or have attached sheet //
   if InPositionEditor || Diagramming || InGameGallery || InGameGrid || RecordingMovie ||
      HaveAttachedSheet()
   {
      // if the alpha value of the game clock is greater than zero //
      if GameClock.alphaValue > 0
      {
         // decrement the alpha value of the game clock //
         GameClock.alphaValue -= 0.05
      }

      // exit the function since don't want to update the game clock //
      return
   }
   // need to update game clock if alpha value less than 1 //
   else if GameClock.alphaValue < 1
   {
      // increment the alpha value of the game clock //
      GameClock.alphaValue += 0.05
   }

   // if the game clock has started //
   if StartClock
   {
      // if it is currently white's turn //
      if (BoardIsWhite(Board) && (!CoreAnimating || UndoingIllegalMove)) || (BoardIsBlack(Board)
         && CoreAnimating && MovingPiece > 0)
      {
         // increment the time for white //
         ElapsedWhiteTime += 1
      }
      // if it is currently black's turn //
      else
      {
         // increment the time for black //
         ElapsedBlackTime += 1
      }
   }

   // if playing a timed game //
   if PlayingTimedGame
   {
      // calculate the time remaining for white //
      RemainingWhiteTime = 216000 - ElapsedWhiteTime

      // calculate the time remaining for black //
      RemainingBlackTime = 216000 - ElapsedBlackTime

      // if hit the max time by white allowed for the game //
      if ElapsedWhiteTime == 216000
      {
         // set current game state to show that white lost on time //
         CurrentGameState = 5
      }
      // if hit the max time by black allowed for the game //
      else if ElapsedBlackTime == 216000
      {
         // set current game state to show that black lost on time //
         CurrentGameState = 6
      }
   }

   // if playing a timed game and current game state is nonzero //
   if CurrentGameState > 0
   {
      // set that querying user that timed game is now over //
      QueryTimedGameOver = true

      // set the alert button to okay //
      AlertButtons = ["Okay"]

      // if game was lost or won on time //
      if CurrentGameState == 5 || CurrentGameState == 6
      {
         // play an alert sound //
         PlayGameSoundThread("warning")
      }

      // if black won //
      if CurrentGameState == 1
      {
         // print that black won //
         MakeAlert(AlertButtons, Query:"Game Over and Black Wins")
      }
      // if white won //
      else if CurrentGameState == 2
      {
         // print that white won //
         MakeAlert(AlertButtons, Query:"Game Over and White Wins")
      }
      // if had a stalemate //
      else if CurrentGameState == 3
      {
         // print that had a stalemate //
         MakeAlert(AlertButtons, Query:"Game Over and Stalemate")
      }
      // if game over from threefold repetition //
      else if CurrentGameState == 4
      {
         // print that game is over from threefold repetition //
         MakeAlert(AlertButtons, Query:"Game Over by Draw from Threefold Repetition")
      }
      // if black won on time //
      else if CurrentGameState == 5
      {
         // print that black won on time //
         MakeAlert(AlertButtons, Query:"Black Wins on Time")
      }
      // if white won on time //
      else if CurrentGameState == 6
      {
         // print that white won on time //
         MakeAlert(AlertButtons, Query:"White Wins on Time")
      }
      // if there is insufficient material for a checkmate //
      else if CurrentGameState == 7
      {
         // print that game is over by insufficient material for mate //
         MakeAlert(AlertButtons, Query:
                   "Game Over by Draw from Insufficient Material for Mate")
      }
   }

   // update the game clock //
   GameClock.UpdateClock(Float(ElapsedWhiteTime) / 10.0, ElapsedBlackTime:
                         Float(ElapsedBlackTime) / 10.0)
}

// start a new game //
func StartNewGame(_ Display:Bool)
{
   // if uci engine loaded and playing against uci engine //
   if UCIEngineLoaded && UCIEngineOpponent
   {
      // tell uci engine to start a new game //
      WriteToEngine("ucinewgame\n")
   }

   // initialize the game attributes //
   Attributes.Initialize()

   // initialize all the booleans //
   InitializeBooleans()

   // initialize all the strings //
   InitializeStrings()

   // initialize all the text fields //
   InitializeTextFields()

   // initialize all the menu items //
   InitializeMenuItems()

   // initialize all the integers //
   InitializeIntegers()

   // initialize all the arrays //
   InitializeArrays()

   // initialize all the buttons //
   InitializeButtons()

   // initialize the date //
   InitializeDate()

   // initialize the board //
   InitializeBoard()

   // initialize the transposition table //
   EngineInitializeTransTable(Engine)

   // remove all the frames //
   Frames.removeAll()

   // remove all the illustrations //
   Illustrations.removeAll()

   // if need to display the board //
   if Display
   {
      // print the initial game information //
      PrintGameInfo()

      // update the chessboard //
      UpdateBoardAndCapturedPieces()
   }

   // set main window title to greenchess //
   MainWindow.title = "GreenChess"
}

// initialize the chessboard //
func InitializeBoard()
{
   // set the initial layout of the pieces on the chessboard //
   SetInitialLayout()

   // initialize the chessboard with the initial fen strings //
   BoardInitializeAttributes(Board, Attributes.FENLayout, Attributes.FENCastlingRights,
                             Attributes.FENColor, Attributes.FENEnpassantSquare)

   // set the start square to the end square so can tell when need to highlight a move //
   StartSquare = EndSquare
}

// initialize the date //
func InitializeDate()
{
   // specify the date format to use //
   TheDateFormatter.dateFormat = "yyyy.MM.dd"

   // set the game date to be the current date //
   GameDate.dateValue = Date()

   // get the current date to use as default time for when game played //
   Attributes.DatePlayed = TheDateFormatter.string(from:GameDate.dateValue)
}

// check whether the specified enpassant square from the position editor is valid //
func CheckEnpassantSquare()
{
   // initially set that the FEN enpassant square from position editor isn't valid //
   ValidFENEnpassantSquare = false

   // if the size of the FEN EP square is of a valid size for square position //
   if Attributes.FENEnpassantSquare.count == 2
   {
      // get the file from the FEN enpassant target square location //
      File = RankOrFileIndex(Attributes.FENEnpassantSquare.first!)

      // get the rank from the FEN enpassant target square location //
      Rank = RankOrFileIndex(Attributes.FENEnpassantSquare.last!)

      // if the file for the enpassant square location is within the board //
      if File >= 0 && File <= 7
      {
         // check if the enpassant square specified is a valid one //
         if (Rank == 2 && Layout[File][3] == 1 && Layout[File][2] == 0 && Layout[File][1]
            == 0 && Attributes.FENColor == "b") || (Rank == 5 && Layout[File][4] == -1
            && Layout[File][5] == 0 && Layout[File][6] == 0 && Attributes.FENColor == "w")
         {
            // set that the enpassant square specified is a valid one //
            ValidFENEnpassantSquare = true

            // save the enpassant square rank //
            EnpassantSquareRank = Rank

            // save the enpassant square file //
            EnpassantSquareFile = File
         }
      }
   }

   // if the FEN enpassant square specified by the user is not valid //
   if !ValidFENEnpassantSquare
   {
      // set that the FEN string won't have a valid enpassant square specified //
      Attributes.FENEnpassantSquare = "-"
   }
}

// determine if the fen layout from the position editor is legal or not //
func FENLayoutIsLegal() -> Bool
{
   // initially set that the fen layout from the position editor is legal //
   LegalFENLayout = true

   // initialize the number of white pieces that were potentially promoted //
   PromotedWhitePieces = 0

   // initialize the number of black pieces that were potentially promoted //
   PromotedBlackPieces = 0

   // convert the current board layout to Forsyth-Edwards notation //
   ConvertLayoutToFEN()

   // create the castling rights in terms of FEN notation //
   CreateCastlingRights()

   // check whether the FEN enpassant target square specified is valid or not //
   CheckEnpassantSquare()

   // initialize different piece types in fen layout from position editor //
   PiecesInFENLayout = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

   // loop over all the files on the board //
   for File in 0 ..< 8
   {
      // loop over all the ranks on the board //
      for Rank in 0 ..< 8
      {
         // increment the number of piece types on the board //
         PiecesInFENLayout[Layout[File][Rank] + 6] += 1
      }
   }

   // loop over all the files //
   for File in 0 ..< 8
   {
      // if have any pawns on the first and eighth ranks which is illegal //
      if abs(Layout[File][0]) == 1 || abs(Layout[File][7]) == 1
      {
         // set that the current fen layout from the edited position is illegal //
         LegalFENLayout = false
      }
   }

   // if there is not just one white king and one black king on the board //
   if PiecesInFENLayout[0] != 1 || PiecesInFENLayout[12] != 1
   {
      // set that the current fen layout from the edited position is illegal //
      LegalFENLayout = false
   }

   // if there are more than eight pawns on the board for either color //
   if PiecesInFENLayout[5] > 8 || PiecesInFENLayout[7] > 8
   {
      // set that the current fen layout from the edited position is illegal //
      LegalFENLayout = false
   }

   // the max number of pieces that could have potentially been promoted by white //
   AllowedWhitePromotedPieces = 8 - PiecesInFENLayout[7]

   // the max number of piece that could have potentially been promoted by black //
   AllowedBlackPromotedPieces = 8 - PiecesInFENLayout[5]

   // the number of pieces that were possibly promoted by white //
   PromotedWhitePieces = PositiveDifference(PiecesInFENLayout[8] - 2) +
                         PositiveDifference(PiecesInFENLayout[9] - 2) +
                         PositiveDifference(PiecesInFENLayout[10] - 2) +
                         PositiveDifference(PiecesInFENLayout[11] - 1)

   // the number of pieces that were possibly promoted by black //
   PromotedBlackPieces = PositiveDifference(PiecesInFENLayout[1] - 1) +
                         PositiveDifference(PiecesInFENLayout[2] - 2) +
                         PositiveDifference(PiecesInFENLayout[3] - 2) +
                         PositiveDifference(PiecesInFENLayout[4] - 2)

   // if white or black promoted more than number of pieces that could promote //
   if PromotedWhitePieces > AllowedWhitePromotedPieces || PromotedBlackPieces >
      AllowedBlackPromotedPieces
   {
      // set that the current fen layout from the edited board position is illegal //
      LegalFENLayout = false
   }

   // if the fen layout still has a possibility of being a legal position //
   if LegalFENLayout
   {
      // initialize the king captured test board with fen color //
      BoardInitializeAttributes(KingCapturedTestBoard, Attributes.FENLayout, Attributes
                                .FENCastlingRights, Attributes.FENColor, Attributes
                                .FENEnpassantSquare)

      // if the king could be captured in this fen layout position //
      if BoardKingIsCaptured(KingCapturedTestBoard)
      {
         // set that current fen layout from the edited board position is illegal //
         LegalFENLayout = false
      }
   }

   // return whether the current fen layout from the edited board position is legal //
   return LegalFENLayout
}

// get the difference value only if it was positive //
func PositiveDifference(_ Difference:Int) -> Int
{
   // if the difference was greater than zero //
   if Difference > 0
   {
      // return the differnce which is positive //
      return Difference
   }
   // the difference was zero or negative //
   else
   {
      // return a value of zero for the difference //
      return 0
   }
}

// the total pieces on the board //
func TotalPiecesOnBoard() -> Int
{
   // initialize total pieces on the board to zero //
   TotalPieces = 0

   // loop over all the files on the board //
   for File in 0 ..< 8
   {
      // loop over all the ranks on the board //
      for Rank in 0 ..< 8
      {
         // if there is a piece on this square //
         if Layout[File][Rank] != 0
         {
            // increment the total number of pieces on the board //
            TotalPieces += 1
         }
      }
   }

   // return the total number of pieces on the board //
   return TotalPieces
}

// save and exit the position editor //
@IBAction func PositionEditorSaveAction(_:AnyObject)
{
   // if the board currently has no pieces on it //
   if TotalPiecesOnBoard() == 0
   {
      // play the sound of a button click //
      PlayGameSoundThread("buttonclick")

      // exit the position editor //
      ExitPositionEditor()

      // exit function since not saving board position //
      return
   }

   // set that currently querying if want to save the edited board position //
   QuerySaveEditedPosition = true

   // if the fen position from the position editor is legal //
   if FENLayoutIsLegal()
   {
      // create the buttons for the query if want to save the edited position //
      AlertButtons = ["Save", "Edit", "Exit"]

      // play the sound of a switch //
      PlayGameSoundThread("switch")

      // make the query if want to save the edited position alert //
      MakeAlert(AlertButtons, Query:"Position Validated")
   }
   // the fen position from the position editor is illegal //
   else
   {
      // create the button to give user option to edit or exit position editor //
      AlertButtons = ["Edit", "Exit"]

      // play the sound of a warning //
      PlayGameSoundThread("warning")

      // tell that the edited board position is illegal and they need to fix it //
      MakeAlert(AlertButtons, Query:"Invalid Position")
   }
}

// convert the edited board position to a fen string representation //
func ConvertLayoutToFEN()
{
   // initialize the fen layout string //
   Attributes.FENLayout = ""

   // loop over all the ranks on the board //
   for Rank in (0 ..< 8).reversed()
   {
      // initialize the count of the empty squares //
      Count = 0

      // loop over all the files on the board //
      for File in 0 ..< 8
      {
         // get the piece type at the current position on the board //
         PieceType = Layout[File][Rank]

         // if encountered an empty square //
         if PieceType == 0
         {
            // increment the count of the empty squares encountered //
            Count += 1
         }

         // if had empty squares and encountered piece or end of the rank //
         if Count > 0 && (PieceType != 0 || (PieceType == 0 && File == 7))
         {
            // append the number of empty squares to the fen layout string //
            Attributes.FENLayout += "\(Count)"

            // reset the empty square count to zero //
            Count = 0
         }

         // if piece type is a black pawn //
         if PieceType == -1
         {
            // add a black pawn to the fen board layout string //
            Attributes.FENLayout += "p"
         }
         // if piece type is a black knight //
         else if PieceType == -2
         {
            // add a black knight to the fen board layout string //
            Attributes.FENLayout += "n"
         }
         // if piece type is a black bishop //
         else if PieceType == -3
         {
            // add a black bishop to the fen board layout string //
            Attributes.FENLayout += "b"
         }
         // if piece type is a black rook //
         else if PieceType == -4
         {
            // add a black rook to the fen board layout string //
            Attributes.FENLayout += "r"
         }
         // if piece type is a black queen //
         else if PieceType == -5
         {
            // add a black queen to the fen board layout string //
            Attributes.FENLayout += "q"
         }
         // if piece type is a black king //
         else if PieceType == -6
         {
            // add a black king to the fen board layout string //
            Attributes.FENLayout += "k"
         }
         // if piece type is a white pawn //
         else if PieceType == 1
         {
            // add a white pawn to the fen board layout string //
            Attributes.FENLayout += "P"
         }
         // if piece type is a white knight //
         else if PieceType == 2
         {
            // add a white knight to the fen board layout string //
            Attributes.FENLayout += "N"
         }
         // if piece type is a white bishop //
         else if PieceType == 3
         {
            // add a white bishop to the fen board layout string //
            Attributes.FENLayout += "B"
         }
         // if piece type is a white rook //
         else if PieceType == 4
         {
            // add a white rook to the fen board layout string //
            Attributes.FENLayout += "R"
         }
         // if piece is a white queen //
         else if PieceType == 5
         {
            // add a white queen to the fen board layout string //
            Attributes.FENLayout += "Q"
         }
         // if piece type is a white king //
         else if PieceType == 6
         {
            // add a white king to the fen board layout string //
            Attributes.FENLayout += "K"
         }

         // reached the end of a rank and not on the first rank //
         if File == 7 && Rank != 0
         {
            // add a forward slash to the fen board layout string for end of rank //
            Attributes.FENLayout += "/"
         }
      }
   }
}

// initialize all the buttons //
func InitializeButtons()
{
   // set the white won button state to off //
   WhiteWonButton.state = NSControl.StateValue(rawValue:0)

   // set the black won button state to off //
   BlackWonButton.state = NSControl.StateValue(rawValue:0)
```

```
   BlackWonButton.state = NSControl.StateValue(rawValue:0)

   // set the draw button state to off //
   DrawButton.state = NSControl.StateValue(rawValue:0)
}

// clear all the arrays //
func InitializeArrays()
{
   // clear the raw stockfish analysis move list //
   RawStockfishMoveList.removeAll()

   // clear the stockfish analysis move list //
   StockfishMoveList.removeAll()

   // clear the entered fen string array //
   EnteredFENStringArray.removeAll()

   // clear the game info list //
   GameInfoList.removeAll()

   // clear the move list //
   MoveList.removeAll()

   // clear the annotation list //
   AnnotationList.removeAll()

   // clear the previous move list //
   PreviousMoveList.removeAll()

   // clear the previous annotation list //
   PreviousAnnotationList.removeAll()

   // clear the fen string array //
   FENStringArray.removeAll()

   // clear the diagram shapes //
   Shapes.removeAll()
}

// get the user saved preference for the specified key //
func PreferenceInt(_ Key:String, Default:Int, Min:Int, Max:Int) -> Int
{
   // if the user preference already exists //
   if AppPreferences.bool(forKey:Key + "Exists")
   {
      // if the user preference is outside the valid range //
      if AppPreferences.integer(forKey:Key) < Min || AppPreferences.integer(forKey:Key) > Max
      {
         // return the default value //
         return Default
      }
      // the user preference was within the valid range //
      else
      {
         // return the user perference for this key //
         return AppPreferences.integer(forKey:Key)
      }
   }
   // the user preference doesn't exist yet //
   else
   {
      // set that the user preference exists now //
      AppPreferences.set(true, forKey:Key + "Exists")

      // set the default value for the specified key //
      AppPreferences.set(Default, forKey:Key)

      // return the default value for the key //
      return Default
   }
}

// get the user saved preference for the key //
func PreferenceBool(_ Key:String, Default:Bool) -> Bool
{
   // if the user preference exists //
   if AppPreferences.bool(forKey:Key + "Exists")
   {
      // return the saved user preference //
      return AppPreferences.bool(forKey:Key)
   }
   // the user preference does not yet exist //
   else
   {
      // set that the user preference exists now //
      AppPreferences.set(true, forKey:Key + "Exists")

      // set the default value for the specified key //
      AppPreferences.set(Default, forKey:Key)

      // return the default value for this key //
      return Default
   }
}

// get the user saved preference for this key //
func PreferenceString(_ Key:String, Default:String) -> String
{
   // if the user saved preference exists //
   if AppPreferences.bool(forKey:Key + "Exists")
   {
      // return the user saved preference for this key //
      return AppPreferences.string(forKey:Key)!
   }
   // the user saved preference does not yet exist //
   else
   {
      // set that the user preference now exists for this key //
      AppPreferences.set(true, forKey:Key + "Exists")

      // set the default value for this key //
      AppPreferences.set(Default, forKey:Key)

      // return the default value for this key //
      return Default
   }
}

// initialize the app preferences //
func InitializePreferences()
{
   // get the background color set //
   BackgroundColorSet = PreferenceInt("BackgroundColorSet", Default:6, Min:1, Max:7)

   // get the rainbow background color //
   RainbowBackgroundColor = PreferenceInt("RainbowBackgroundColor", Default:78, Min:0, Max:
                                          RainbowBackgroundColors.count - 1)

   // store the rainbow background colors //
   RainbowBackgroundView.StoreColorSet(RainbowBackgroundColors, ChosenColor:
                                       RainbowBackgroundColor)

   // get the blue background color //
   BlueBackgroundColor = PreferenceInt("BlueBackgroundColor", Default:200, Min:0, Max:
                                       BlueBackgroundColors.count - 1)

   // store the blue background colors //
   BlueBackgroundView.StoreColorSet(BlueBackgroundColors, ChosenColor:BlueBackgroundColor)

   // get the brown background color //
   BrownBackgroundColor = PreferenceInt("BrownBackgroundColor", Default:200, Min:0, Max:
                                        BrownBackgroundColors.count - 1)

   // store the brown background colors //
   BrownBackgroundView.StoreColorSet(BrownBackgroundColors, ChosenColor:BrownBackgroundColor)

   // get the green background color //
   GreenBackgroundColor = PreferenceInt("GreenBackgroundColor", Default:200, Min:0, Max:
                                        GreenBackgroundColors.count - 1)

   // store the green background colors //
   GreenBackgroundView.StoreColorSet(GreenBackgroundColors, ChosenColor:GreenBackgroundColor)

   // get the teal background color //
   TealBackgroundColor = PreferenceInt("TealBackgroundColor", Default:200, Min:0, Max:
                                       TealBackgroundColors.count - 1)

   // store the teal background colors //
   TealBackgroundView.StoreColorSet(TealBackgroundColors, ChosenColor:TealBackgroundColor)

   // get the yellow background color //
   YellowBackgroundColor = PreferenceInt("YellowBackgroundColor", Default:200, Min:0, Max:
                                         YellowBackgroundColors.count - 1)

   // store the yellow background colors //
   YellowBackgroundView.StoreColorSet(YellowBackgroundColors, ChosenColor:YellowBackgroundColor)

   // get the gray background color //
   GrayBackgroundColor = PreferenceInt("GrayBackgroundColor", Default:200, Min:0, Max:
                                       GrayBackgroundColors.count - 1)

   // store the gray background colors //
   GrayBackgroundView.StoreColorSet(GrayBackgroundColors, ChosenColor:GrayBackgroundColor)

   // get the window texture opacity //
   WindowTextureOpacity = PreferenceInt("WindowTextureOpacity", Default:0, Min:0, Max:100)

   // update the window texture opacity slider //
   WindowTextureOpacitySlider.intValue = Int32(WindowTextureOpacity)

   // display the window texture opacity //
   DisplayWindowTextureOpacity()

   // set the current background color from the current color set //
   SetBackgroundColor(BackgroundColorSet)

   // retrieve the saved label color //
   LabelColor = PreferenceInt("LabelColor", Default:FontColors.count - 1, Min:0, Max:
                              FontColors.count - 1)

   // update the board labels color view //
   BoardLabelsColorView.StoreColorSet(FontColors, ChosenColor:LabelColor)

   // retrieve the saved info color //
   InfoColor = PreferenceInt("InfoColor", Default:36, Min:0, Max:FontColors.count - 1)

   // update the game info color view //
   GameInfoColorView.StoreColorSet(FontColors, ChosenColor:InfoColor)

   // retrieve the saved move list color //
   MoveListColor = PreferenceInt("MoveListColor", Default:36, Min:0, Max:FontColors.count - 1)

   // update the move list color view //
   MoveListColorView.StoreColorSet(FontColors, ChosenColor:MoveListColor)

   // retrieve the saved game list color //
   GameListColor = PreferenceInt("GameListColor", Default:36, Min:0, Max:FontColors.count - 1)

   // update the game list color view //
   GameListColorView.StoreColorSet(FontColors, ChosenColor:GameListColor)

   // retrieve the saved game clock color //
   GameClockColor = PreferenceInt("GameClockColor", Default:4, Min:0, Max:GameClockColors
                                  .count - 1)

   // store the game clock color set //
   ClockColorView.StoreColorSet(GameClockColors, ChosenColor:GameClockColor)

   // update the game clock color //
   GameClock.UpdateClockColor(GameClockColors[GameClockColor])

   // update the game clock preview border color //
   GameClockPreview.UpdateClockColor(GameClockColors[GameClockColor])

   // retrieve the saved game clock background color //
   GameClockBackgroundColor = PreferenceInt("GameClockBackgroundColor", Default:4, Min:0, Max:
                                            GameClockColors.count - 1)

   // store the game clock background color set //
   ClockBackgroundColorView.StoreColorSet(GameClockColors, ChosenColor:GameClockBackgroundColor)

   // update the game clock background color //
   GameClock.UpdateClockBackgroundColor(GameClockColors[GameClockBackgroundColor])

   // update the game clock preview background color //
   GameClockPreview.UpdateClockBackgroundColor(GameClockColors[GameClockBackgroundColor])

   // update the clock background visibility value //
   ClockBackgroundVisibility = PreferenceInt("ClockBackgroundVisibility", Default:10, Min:0,
                                             Max:100)

   // update the clock background visibility slider //
   ClockBackgroundVisibilitySlider.intValue = Int32(ClockBackgroundVisibility)

   // display the clock background visibility //
   DisplayClockBackgroundVisibility()

   // update the game clock background visibility //
   GameClock.UpdateClockBackgroundVisibility(ClockBackgroundVisibility)

   // update the game clock preview background visibility //
   GameClockPreview.UpdateClockBackgroundVisibility(ClockBackgroundVisibility)

   // update the game clock preview //
   GameClockPreview.UpdateClock(0, ElapsedBlackTime:0)

   // retrieve the chess piece set type preference //
   ChessPieceSetType = PreferenceInt("ChessPieceSetType", Default:0, Min:0, Max:3)

   // update the piece set button for the chosen piece set //
   PieceSetButton.selectItem(at:ChessPieceSetType)

   // set the default engine to greenchess to play against //
   Opponent(false, Engine:true, UCIEngine:false)

   // retrieve the board type preference //
   BoardType = PreferenceInt("BoardType", Default:0, Min:0, Max:18)

   // update the board textures button to the chosen board type //
   BoardTexturesButton.selectItem(at:BoardType)

   // retrieve the board labels font type //
   BoardLabelsFontType = PreferenceInt("BoardLabelsFontType", Default:17, Min:0, Max:Fonts.count
                                       - 1)

   // set the current board labels font type //
   BoardLabelsFontButton.selectItem(at:BoardLabelsFontType)

   // retrieve the move list font type //
   MoveListFontType = PreferenceInt("MoveListFontType", Default:17, Min:0, Max:Fonts.count - 1)

   // set the current move list font type //
   MoveListFontButton.selectItem(at:MoveListFontType)

   // retrieve the game info font type //
   GameInfoFontType = PreferenceInt("GameInfoFontType", Default:17, Min:0, Max:Fonts.count - 1)

   // set the current game info font type //
   GameInfoFontButton.selectItem(at:GameInfoFontType)

   // retrieve the game list font type //
   GameListFontType = PreferenceInt("GameListFontType", Default:17, Min:0, Max:Fonts.count - 1)

   // set the current game list font type //
   GameListFontButton.selectItem(at:GameListFontType)

   // retrieve the board labels bold setting //
   BoardLabelsBold = PreferenceBool("BoardLabelsBold", Default:false)

   // set the current board labels bold setting //
   BoardLabelsBoldButton.selectItem(at:BoolToInt(BoardLabelsBold))

   // retrieve the move list bold setting //
   MoveListBold = PreferenceBool("MoveListBold", Default:false)

   // set the current move list bold setting //
   MoveListBoldButton.selectItem(at:BoolToInt(MoveListBold))

   // retrieve the game info bold setting //
   GameInfoBold = PreferenceBool("GameInfoBold", Default:false)

   // set the current game info bold setting //
   GameInfoBoldButton.selectItem(at:BoolToInt(GameInfoBold))

   // retrieve the game list bold setting //
   GameListBold = PreferenceBool("GameListBold", Default:false)

   // set the current game list bold setting //
   GameListBoldButton.selectItem(at:BoolToInt(GameListBold))

   // retrieve the board labels font sizes setting //
   BoardLabelsSizes = PreferenceInt("BoardLabelsSizes", Default:2, Min:0, Max:2)

   // set the current board labels font sizes setting //
   BoardLabelsSizesButton.selectItem(at:BoardLabelsSizes)

   // retrieve the move list font sizes setting //
   MoveListSizes = PreferenceInt("MoveListSizes", Default:1, Min:0, Max:2)

   // set the current move list font sizes setting //
   MoveListSizesButton.selectItem(at:MoveListSizes)

   // retrieve the game info sizes setting //
   GameInfoSizes = PreferenceInt("GameInfoSizes", Default:1, Min:0, Max:2)

   // set the current game info sizes setting //
   GameInfoSizesButton.selectItem(at:GameInfoSizes)

   // retrieve the game list sizes setting //
   GameListSizes = PreferenceInt("GameListSizes", Default:1, Min:0, Max:2)

   // set the current game list sizes setting //
   GameListSizesButton.selectItem(at:GameListSizes)

   // retrieve the board animation rate preference //
   AnimationRate = PreferenceInt("AnimationRate", Default:75, Min:1, Max:100)

   // update the board animation rate slider //
   AnimationRateSlider.intValue = Int32(AnimationRate)

   // display the board animation rate //
   DisplayAnimationRate()

   // retrieve the gallery and grid animation rate preference //
   GalleryGridAnimationRate = PreferenceInt("GalleryGridAnimationRate", Default:75, Min:1, Max:
                                            100)

   // update the gallery and grid animation rate slider //
   GalleryGridAnimationRateSlider.intValue = Int32(GalleryGridAnimationRate)

   // display the gallery and grid animation rate //
   DisplayGalleryGridAnimationRate()

   // set the game gallery animation rate //
   GameGalleryView.SetAnimationRate(GalleryGridAnimationRate)

   // set the game grid animation rate //
   GameGridView.SetAnimationRate(GalleryGridAnimationRate)

   // retrieve the chess board border visibility //
   BorderVisibility = PreferenceInt("BorderVisibility", Default:100, Min:0, Max:100)

   // update the chess board border visibility slider //
   BorderVisibilitySlider.intValue = Int32(BorderVisibility)

   // retrieve the board tint visibility slider //
   BoardTintVisibility = PreferenceInt("BoardTintVisibility", Default:0, Min:0, Max:100)

   // update the board tint visibility slider //
   BoardTintVisibilitySlider.intValue = Int32(BoardTintVisibility)

   // get the board roughness //
   BoardRoughness = PreferenceInt("BoardRoughness", Default:0, Min:0, Max:100)

   // update the board roughness slider //
   BoardRoughnessSlider.intValue = Int32(BoardRoughness)

   // get the square border //
   SquareBorder = PreferenceInt("SquareBorder", Default:0, Min:0, Max:100)

   // update the square border slider //
   SquareBorderSlider.intValue = Int32(SquareBorder)

   // retrieve the board tint color //
   BoardTintColor = PreferenceInt("BoardTintColor", Default:0, Min:0, Max:BoardTintColors.count
                                  - 1)

   // update the board tint color slider //
   BoardTintColorSlider.intValue = Int32(BoardTintColor)

   // retrieve the chess board border texture //
   BorderTexture = PreferenceInt("BorderTexture", Default:100, Min:0, Max:100)

   // update the chess board border texture slider //
   BorderTextureSlider.intValue = Int32(BorderTexture)

   // retrieve the uci engine difficulty preference //
   UCIEngineDifficulty = PreferenceInt("UCIEngineDifficulty", Default:20, Min:0, Max:20)

   // update the uci engine difficulty slider //
   UCIEngineDifficultySlider.intValue = Int32(UCIEngineDifficulty)

   // display the uci engine difficulty //
   DisplayUCIEngineDifficulty()

   // retrieve the uci engine hash size preference //
   UCIEngineHash = PreferenceInt("UCIEngineHash", Default:16, Min:16, Max:512)

   // update the uci engine hash slider //
   UCIEngineHashSlider.intValue = Int32(UCIEngineHash)

   // display the uci engine hash size //
   DisplayUCIEngineHash()

   // retrieve the uci engine riskiness preference //
   UCIEngineRiskiness = PreferenceInt("UCIEngineRiskiness", Default:0, Min:-100, Max:100)

   // update the uci engine riskiness slider //
   UCIEngineRiskinessSlider.intValue = Int32(UCIEngineRiskiness)

   // display the uci engine riskiness //
   DisplayUCIEngineRiskiness()

   // retrieve the search time preference //
   SearchTime = PreferenceInt("SearchTime", Default:5, Min:5, Max:120)

   // update the search time slider //
   SearchTimeSlider.intValue = Int32(SearchTime)

   // display the search time //
   DisplaySearchTime()

   // retrieve the sound off preference //
   SoundOff = PreferenceBool("SoundOff", Default:false)

   // update the game sounds button //
   GameSoundsButton.state = NSControl.StateValue(rawValue:BoolToInt(!SoundOff))

   // retrieve the move by selecting squares preference //
   MoveBySelectingSquares = PreferenceBool("MoveBySelectingSquares", Default:false)

   // update the move by selecting squares button //
   MoveBySelectingSquaresButton.state = NSControl.StateValue(rawValue:
                                        BoolToInt(MoveBySelectingSquares))

   // retrieve the show stockfish analysis preference //
   ShowStockfishAnalysis = PreferenceBool("ShowStockfishAnalysis", Default:false)

   // update the show stockfish analysis button //
   ShowStockfishAnalysisButton.state = NSControl.StateValue(rawValue:
                                       BoolToInt(ShowStockfishAnalysis))

   // retrieve the autoload stockfish preference //
   AutoloadStockfish = PreferenceBool("AutoloadStockfish", Default:false)

   // update the autoload stockfish button state //
   AutoloadStockfishButton.state = NSControl.StateValue(rawValue:BoolToInt(AutoloadStockfish))

   // if want to autoload the stockfish engine //
   if AutoloadStockfish
   {
      // autoload the stockfish engine //
      UCIEngineOpponentMenuItemAction(self)
   }

   // retrieve the hide info panel value //
   HideInfoPanel = PreferenceBool("HideInfoPanel", Default:false)

   // update the small info panel value //
   SmallInfoPanel = PreferenceBool("SmallInfoPanel", Default:false)

   // update the large info panel value //
   LargeInfoPanel = PreferenceBool("LargeInfoPanel", Default:true)

   // update the hide game clock in panel value //
   HideGameClockInPanel = PreferenceBool("HideGameClockInPanel", Default:false)

   // update the game clock in panel hidden state //
   HideGameClockInPanelAction()

   // retrieve the hide pieces in panel value //
   HideCapturedPiecesInPanel = PreferenceBool("HideCapturedPiecesInPanel", Default:false)

   // update the captured pieces in panel hidden state //
   HideCapturedPiecesInPanelAction()

   // retrieve the board rotation preference //
   RotateBoard = PreferenceBool("RotateBoard", Default:false)

   // update the rotated board menu item //
   RotateBoardMenuItem.state = NSControl.StateValue(rawValue:BoolToInt(RotateBoard))

   // retrieve the use eco book preference //
   UseEcoBook = PreferenceBool("UseEcoBook", Default:false)

   // update the use eco book button //
   EcoBookButton.state = NSControl.StateValue(rawValue:BoolToInt(UseEcoBook))

   // retrieve the autosave game preference //
   AutosaveGame = PreferenceBool("AutosaveGame", Default:false)

   // update the autosave game button //
   AutosaveGameButton.state = NSControl.StateValue(rawValue:BoolToInt(AutosaveGame))

   // retrieve the highlight last move made preference //
   HighlightLastMoveMade = PreferenceBool("HighlightLastMoveMade", Default:false)

   // update the highlight last move made button //
   HighlightLastMoveMadeButton.state = NSControl.StateValue(rawValue:
                                       BoolToInt(HighlightLastMoveMade))

   // retrieve the highlight captured pieces preference //
   HighlightCapturedPieces = PreferenceBool("HighlightCapturedPieces", Default:false)

   // update the highlight captured pieces button //
   HighlightCapturedPiecesButton.state = NSControl.StateValue(rawValue:
                                         BoolToInt(HighlightCapturedPieces))

   // retrieve the highlight threatened pieces preference //
   HighlightThreatenedPieces = PreferenceBool("HighlightThreatenedPieces", Default:false)

   // update the highlight threatened pieces button //
   HighlightThreatenedPiecesButton.state = NSControl.StateValue(rawValue:
                                           BoolToInt(HighlightThreatenedPieces))

   // retrieve the multithread uci engine preference //
   MultithreadUCIEngine = PreferenceBool("MultithreadUCIEngine", Default:false)

   // update the multithread uci engine button //
   MultithreadUCIEngineButton.state = NSControl.StateValue(rawValue:
                                      BoolToInt(MultithreadUCIEngine))

   // set the number of threads to use for the uci engine //
   SetThreadsToUse()

   // get the current saved position slot value //
   SavedPositionSlot = PreferenceInt("SavedPositionSlot", Default:1, Min:1, Max:10)

   // loop over all the saved positions //
   for PositionListIndex in 1 ... 10
   {
      // initialize game attributes for the given position index //
      Attributes.Initialize("\(PositionListIndex)")

      // if the position image is not empty //
      if Attributes.PositionImage.tiffRepresentation?.count != nil
      {
         // append the saved position image //
         SavedPositionImages.append(Attributes.PositionImage)

         // retrieve the move list if it was stored //
         MoveList = Attributes.MoveListString.components(separatedBy:"*********")

         // retrieve the annotation list if it was stored //
         AnnotationList = Attributes.AnnotationListString.components(separatedBy:"*********")

         // clear the move and annotation lists if they turned out to be empty //
         ClearEmptyMoveList()

         // convert the current game to game data //
         ConvertToGameData()

         // create a new element at the end of the saved positions data array //
         SavedPositionData.append(DataNew())

         // copy the game data to the end of the saved positions data array //
         DataCopy(SavedPositionData[SavedPositionData.count - 1], GameData)
      }
   }

   // get the saved preferences selection //
   PreferencesSelection = PreferenceInt("PreferencesSelection", Default:20, Min:20, Max:25)

   // if the preferences selection is 20 //
   if PreferencesSelection == 20
   {
      // set the background preferences //
      BackgroundPreferencesAction(self)
   }
   // if the preferences selection is 21 //
   else if PreferencesSelection == 21
   {
      // set the board preferences //
      BoardPreferencesAction(self)
   }
   // if the preferences selection is 22 //
   else if PreferencesSelection == 22
   {
      // set the font preferences //
      FontPreferencesAction(self)
   }
   // if the preferences selection is 23 //
   else if PreferencesSelection == 23
   {
      // set the stockfish preferences //
      StockfishPreferencesAction(self)
   }
   // if the preferences selection is 24 //
   else if PreferencesSelection == 24
   {
      // set the speed preferences //
      SpeedPreferencesAction(self)
   }
   // if the preferences selection is 25 //
   else if PreferencesSelection == 25
   {
      // set the clock preferences //
      ClockPreferencesAction(self)
   }

   // set the initial preferences toolbar item //
   PreferencesToolbar.selectedItemIdentifier = NSToolbarItem.Identifier(rawValue:
                                                "\(PreferencesSelection)")

   // get the game attributes //
   Attributes.Initialize("")

   // retrieve the move list if it was stored //
   MoveList = Attributes.MoveListString.components(separatedBy:"*********")

   // retrieve the annotation list if it was stored //
   AnnotationList = Attributes.AnnotationListString.components(separatedBy:"*********")

   // clear the move and annotation lists if they turned out to be empty //
   ClearEmptyMoveList()

   // if doing autosave of game //
   if AutosaveGame
   {
      // convert autosave data to game data for storage if autosave is turned on //
      ConvertToGameData()
   }

   // update the background color layer //
   UpdateBackgroundColorLayer()

   // update the chess board border layer //
   UpdateBorderLayer(self)

   // update the chess board border visibility //
   UpdateBorderVisibility(self)

   // update the board labels layer //
   UpdateBoardLabelsLayer()

   // update the chess board type layer //
   UpdateBoardTypeLayer(self)

   // update the board tint layer //
   UpdateBoardTintLayer(self)

   // update the board tint visibility //
   UpdateBoardTintVisibility(self)

   // update the board roughness value //
   UpdateBoardRoughness(self)

   // update the square border value //
   UpdateSquareBorder(self)
}

// clear an empty move and annotation list //
func ClearEmptyMoveList()
{
   // if the move list just has an empty value in it //
   if MoveList.count == 1
   {
      // if the move is an empty string //
      if MoveList[0].isEmpty
      {
         // remove all the move list objects //
         MoveList.removeAll()

         // remove all the annotation list objects //
         AnnotationList.removeAll()
      }
   }
}

// clear an empty stockfish analysis move list //
func RemoveEmptyStockfishMoveListElements()
{
   // clear the stockfish analysis move list //
   StockfishMoveList.removeAll()

   // loop over all the moves in the raw stockfish analysis move list //
   for Index in 0 ..< RawStockfishMoveList.count
   {
      // if the stockfish analysis move list element is not empty //
      if !RawStockfishMoveList[Index].isEmpty
      {
         // save the raw stockfish analysis move list element //
         StockfishMoveList.append(RawStockfishMoveList[Index])
      }
   }

   // if there is at least one element in the stockfish move list //
   if StockfishMoveList.count >= 1
   {
      // remove the first stockfish move list element since don't need first move //
      StockfishMoveList.remove(at:0)
   }
}

// store the autosave data //
func StoreAutosaveData(_ Tag:String)
{
   // if not autosaving data or currently reviewing a game and not saving position //
   if !AutosaveGame || (ExploringGame && !QuerySavePosition)
   {
      // exit function since not autosaving data //
      return
   }

   // initialize the move list string //
   Attributes.MoveListString = ""

   // initialize the annotation list string //
   Attributes.AnnotationListString = ""

   // loop over the move list //
   for AutosaveIndex in 0 ..< MoveList.count
   {
      // append the move to the move list string //
      Attributes.MoveListString += MoveList[AutosaveIndex]

      // append the annotation to the annotation list string //
      Attributes.AnnotationListString += AnnotationList[AutosaveIndex]

      // if not the last move in the move list //
      if AutosaveIndex < MoveList.count - 1
      {
         // append a separator to the move list string //
         Attributes.MoveListString += "*********"

         // append a separator to the annotation list string //
         Attributes.AnnotationListString += "*********"
      }
   }

   // store the autosave data for the game attributes //
   Attributes.StoreAutosaveData(Tag)
}

// initialize all the integers //
func InitializeIntegers()
{
   // set the current game state to zero //
   CurrentGameState = 0

   // initialize the remaining white time to zero //
   RemainingWhiteTime = 0

   // initialize the remaining black time to zero //
   RemainingBlackTime = 0

   // set the timed game offset to zero //
   TimedGameOffset = 0

   // set total game time to zero //
   TotalGameTime = 0

   // set the timed game color to zero //
   TimedGameColor = 0

   // set the timed game time to zero //
   TimedGameTime = 0

   // set the timed game opponent to zero //
   TimedGameOpponent = 0

   // set the initial board square offset //
   SquareOffset = 68

   // set the initial moving piece to none //
   MovingPiece = 0

   // initialize the selected squares to zero //
   SelectedSquares = 0
}

// initialize all the text fields //
func InitializeTextFields()
{
   // clear the whitename text field //
   WhiteNameTextField.stringValue = ""

   // clear the blackname text field //
   BlackNameTextField.stringValue = ""

   // clear the sitelocation text field //
   SiteLocationTextField.stringValue = ""

   // clear the event title text field //
   EventTitleTextField.stringValue = ""

   // clear the round number text field //
   RoundNumberTextField.stringValue = ""

   // clear the time control text field //
   TimeControlTextField.stringValue = ""

   // clear the opening text field //
   OpeningTextField.stringValue = ""

   // clear the section text field //
   SectionTextField.stringValue = ""

   // clear the board number text field //
   BoardNumberTextField.stringValue = ""

   // clear the white elo text field //
   WhiteEloTextField.stringValue = ""

   // clear the black elo text field //
   BlackEloTextField.stringValue = ""

   // clear the white uscf text field //
   WhiteUSCFTextField.stringValue = ""

   // clear the black uscf text field //
   BlackUSCFTextField.stringValue = ""

   // clear the annotator text field //
   AnnotatorTextField.stringValue = ""

   // clear the eco text field //
   ECOTextField.stringValue = ""

   // clear the game list search box text field //
   GameListSearchBox.stringValue = ""
}

// display the game list count to the game count view //
func DisplayGameCount()
{
   // if the game list is empty or haven't read in a game list //
   if GameListData.numberOfRows(in:NSTableView()) == 0 || !ReadInGameList
   {
      // display that have no games in the game list //
      MainWindow.title = "No Games"
   }
   // if only have one game in the game list //
   else if GameListData.numberOfRows(in:NSTableView()) == 1
   {
      // display that only have one game in the game list //
      MainWindow.title = "1 Game"
   }
   // have more than one game in the game list //
   else
   {
      // display number of games in the current game list //
      MainWindow.title = "\(GameListData.numberOfRows(in:NSTableView())) Games"
   }
}

// update the game list //
func UpdateGameList()
{
   // get the search string from the game list search box when it is updated //
   SearchString = StringFromTextField(GameListSearchBox.stringValue)

   // condense the game collection data which matches the given search string //
   GameListData.CondenseGameList(SearchString)

   // update the game list table view //
   UpdateTheGameList()

   // display the current game count //
   DisplayGameCount()
}

// if the text in a text field box changed //
override func controlTextDidChange(_ TheNotification:Notification)
{
   // if currently searching the game list //
   if SearchingGameList
   {
      // initialize the previously selected row to zero //
      PreviouslySelectedRow = 0

      // update the game list //
      UpdateGameList()
   }
}

// initialize all the menu items //
func InitializeMenuItems()
{
   // set the game explorer menu item title //
   GameExplorerMenuItem.title = "Explorer"

   // set the go stop menu item title //
   GoStopMenuItem.title = "Engine"

   // set the hide menu item title //
   HideMenuItem.title = "Hide"

   // set the timed game menu item title //
   TimedGameMenuItem.title = "Timed Game"

   // turn off the new game menu item //
   NewGameMenuItem.state = NSControl.StateValue(rawValue:0)

   // turn off the board snapshot menu item //
   BoardSnapshotMenuItem.state = NSControl.StateValue(rawValue:0)

   // turn off the engine match menu item //
   EngineMatchMenuItem.state = NSControl.StateValue(rawValue:0)

   // turn off the hide menu item //
   HideMenuItem.state = NSControl.StateValue(rawValue:0)

   // turn off the quit menu item //
   QuitMenuItem.state = NSControl.StateValue(rawValue:0)

   // turn off the current position editor menu item //
   CurrentPositionEditorMenuItem.state = NSControl.StateValue(rawValue:0)

   // turn off the go stop menu item //
   GoStopMenuItem.state = NSControl.StateValue(rawValue:0)

   // turn off the about menu item //
   AboutMenuItem.state = NSControl.StateValue(rawValue:0)

   // turn off the save position menu item //
   SavePositionMenuItem.state = NSControl.StateValue(rawValue:0)

   // turn off the load position menu item //
   LoadPositionMenuItem.state = NSControl.StateValue(rawValue:0)

   // turn off the game gallery menu item //
   GameGalleryMenuItem.state = NSControl.StateValue(rawValue:0)

   // turn off the game grid menu item //
   GameGridMenuItem.state = NSControl.StateValue(rawValue:0)

   // turn off the open game menu item //
   OpenGameMenuItem.state = NSControl.StateValue(rawValue:0)

   // turn off the save game menu item //
   SaveGameMenuItem.state = NSControl.StateValue(rawValue:0)

   // turn off the append game menu item //
   AppendGameMenuItem.state = NSControl.StateValue(rawValue:0)

   // turn off the time game menu item //
   TimedGameMenuItem.state = NSControl.StateValue(rawValue:0)
```

```
   // turn off the enter fen menu item //
   EnterFENMenuItem.state = NSControl.StateValue(rawValue:0)

   // change the database explorer menu item to off //
   DatabaseExplorerMenuItem.state = NSControl.StateValue(rawValue:0)

   // turn off the game annotator menu item //
   GameAnnotatorMenuItem.state = NSControl.StateValue(rawValue:0)

   // change the position editor menu item to off //
   PositionEditorMenuItem.state = NSControl.StateValue(rawValue:0)

   // turn the search game list menu item off //
   SearchGameListMenuItem.state = NSControl.StateValue(rawValue:0)

   // turn the game explorer menu item off //
   GameExplorerMenuItem.state = NSControl.StateValue(rawValue:0)

   // turn the enter game info menu item off //
   EnterGameInfoMenuItem.state = NSControl.StateValue(rawValue:0)

   // turn the preferences menu item off //
   PreferencesMenuItem.state = NSControl.StateValue(rawValue:0)
}

// initialize all the strings //
func InitializeStrings()
{
   // initialize the stockfish principle variation best line to empty //
   StockfishPV = ""

   // initialize the stockfish analysis string to empty //
   StockfishAnalysisString = ""

   // initialize the entered fen string to empty //
   EnteredFENString = ""

   // initialize the engine command string //
   EngineCommand = ""

   // initialize the fen piece //
   FENPiece = ""
}

// set the default fen strings //
func SetDefaultFENStrings()
{
   // set the starting fen layout for the board //
   Attributes.FENLayout = DefaultFENLayout

   // initialize the FEN string color //
   Attributes.FENColor = "w"

   // initialize the FEN castling rights //
   Attributes.FENCastlingRights = "KQkq"

   // initialize the FEN enpassant target square //
   Attributes.FENEnpassantSquare = "-"

   // initialize the FEN starting move number for the game //
   Attributes.FENStartMoveNumber = "1"
}

// initialize all the booleans //
func InitializeBooleans()
{
   // set that initially haven't started the clock //
   StartClock = false

   // set valid entered fen string to false //
   ValidFENString = false

   // set that not currently recording a movie //
   RecordingMovie = false

   // set that not currently playing an engine match //
   PlayingEngineMatch = false

   // set that not initially diagramming //
   Diagramming = false

   // set that not currently a valid piece moving //
   ValidPiece = false

   // set that not currently a valid diagram shape //
   ValidShape = false

   // set that not yet determined if will save the edited board position //
   WillSaveEditedPosition = false

   // set that not initially in the position editor //
   InPositionEditor = false

   // set that not initially editing nonempty board position //
   EditingNonemptyPosition = false

   // set that not initially in the database explorer //
   InDatabaseExplorer = false

   // set that not initially in the enter fen window //
   InEnterFENWindow = false

   // set that white has not won the game yet //
   WhiteWonGame = false

   // set that black has not won the game yet //
   BlackWonGame = false

   // set that the game has not been drawn yet //
   DrawnGame = false

   // set that not currently entering game info //
   EnteringGameInfo = false

   // whether in the about window or not //
   InAboutWindow = false

   // set loading saved position to false //
   LoadingSavedPosition = false

   // set query save position to false //
   QuerySavePosition = false

   // set query load position to false //
   QueryLoadPosition = false

   // set that not initially searching the game list //
   SearchingGameList = false

   // set that not currently entering in preferences //
   EnteringPreferences = false

   // set that not currently in the game gallery //
   InGameGallery = false

   // set that not currently in the game grid //
   InGameGrid = false

   // set that not currently creating the game gallery grid //
   CreatingGalleryGrid = false

   // set that initially not querying the user for a pawn promotion //
   QueryPawnPromotion = false

   // set that not initially querying the user to start a new game //
   QueryStartNewGame = false

   // set that not initially querying whether the uci chess engine loaded or not //
   QueryUCIEngineLoad = false

   // set that not querying user if timed game is over //
   QueryTimedGameOver = false

   // set that not initially not querying whether want to open game file //
   QueryOpenGameFile = false

   // set that not initially not querying if user wants to save edited board position //
   QuerySaveEditedPosition = false

   // set that not initially querying the user to quit the application //
   QueryQuitApplication = false

   // set that game review not currently paused //
   PauseGameReview = false

   // set that user is not currently reviewing the game //
   ReviewingGame = false

   // set that user is not currently exploring the game //
   ExploringGame = false

   // set that not currently annotating the game //
   AnnotatingGame = false

   // set that not currently playing timed game //
   PlayingTimedGame = false

   // set that not currently entering timed game info //
   EnteringTimedGameInfo = false

   // set that not currently analyzing the game //
   AnalyzingGame = false

   // set that initially user is not undoing an illegal move //
   UndoingIllegalMove = false

   // set that engine is currently not thinking //
   EngineThinking = false

   // set that not currently core animating //
   CoreAnimating = false
}

// set the initial chessboard layout //
func SetInitialLayout()
{
   // set that fen string is initially true //
   ValidFENString = true

   // initialize the captured white pieces to display //
   CapturedWhitePieces = [0, 0, 0, 0, 0]

   // initialize the captured black pieces to display //
   CapturedBlackPieces = [0, 0, 0, 0, 0]

   // initially have no moving piece at beginning of game //
   MovingPiece = 0

   // set that initially have no promotion type at beginning of game //
   PromotionType = 0

   // initialize the board layout //
   Layout = BlankLayout

   // set the initial rank value on the board //
   Rank = 7

   // set the initial file value on the board //
   File = 0

   // loop over the entire fen layout string //
   for Index in 0 ..< Attributes.FENLayout.count
   {
      // if the fen layout was corrupted and out of range //
      if Rank < 0 || File < 0 || Rank > 7 || File > 7
      {
         // set that the fen string was not valid //
         ValidFENString = false

         // exit loop since fen layout was corrupted and rank or file out or range //
         break
      }

      // get the current piece from the fen string //
      FENPiece = String(Attributes.FENLayout.prefix(Index + 1).last!)

      // if the piece is a black pawn //
      if FENPiece == "p"
      {
         // add a black pawn to the board layout //
         Layout[File][Rank] = -1

         // increment the file //
         File += 1
      }
      // if the piece is a black knight //
      else if FENPiece == "n"
      {
         // add a black knight to the board layout //
         Layout[File][Rank] = -2

         // increment the file //
         File += 1
      }
      // if the piece is a black bishop //
      else if FENPiece == "b"
      {
         // add a black bishop to the board layout //
         Layout[File][Rank] = -3

         // increment the file //
         File += 1
      }
      // if the piece is a black rook //
      else if FENPiece == "r"
      {
         // add a black rook to the board layout //
         Layout[File][Rank] = -4

         // increment the file //
         File += 1
      }
      // if the piece is a black queen //
      else if FENPiece == "q"
      {
         // add a black queen to the board layout //
         Layout[File][Rank] = -5

         // increment the file //
         File += 1
      }
      // if the piece is a black king //
      else if FENPiece == "k"
      {
         // add a black king to the board layout //
         Layout[File][Rank] = -6

         // increment the file //
         File += 1
      }
      // if the piece is a white pawn //
      else if FENPiece == "P"
      {
         // add a white pawn to the board layout //
         Layout[File][Rank] = 1

         // increment the file //
         File += 1
      }
      // if the piece is a white knight //
      else if FENPiece == "N"
      {
         // add a white knight to the board layout //
         Layout[File][Rank] = 2

         // increment the file //
         File += 1
      }
      // if the piece is a white bishop //
      else if FENPiece == "B"
      {
         // add a white bishop to the board layout //
         Layout[File][Rank] = 3

         // increment the file //
         File += 1
      }
      // if the piece is a white rook //
      else if FENPiece == "R"
      {
         // add a white rook to the board layout //
         Layout[File][Rank] = 4

         // increment the file //
         File += 1
      }
      // if the piece is a white queen //
      else if FENPiece == "Q"
      {
         // add a white queen to the board layout //
         Layout[File][Rank] = 5

         // increment the file //
         File += 1
      }
      // if the piece is a white king //
      else if FENPiece == "K"
      {
         // add a white king to the board layout //
         Layout[File][Rank] = 6

         // increment the file //
         File += 1
      }
      // if the fen piece is just a count of how many empty squares there are //
      else if FENPiece != "/" && FENPiece >= "1" && FENPiece <= "8"
      {
         // loop over the count of empty squares in the fen string //
         for _ in 0 ..< Int(FENPiece)!
         {
            // if the fen layout was corrupted and out of range //
            if Rank < 0 || File < 0 || Rank > 7 || File > 7
            {
               // set that the fen string was not valid //
               ValidFENString = false

               // exit loop since rank or file out or range //
               break
            }

            // set the current layout to an empty square //
            Layout[File][Rank] = 0

            // increment the file //
            File += 1
         }
      }

      // if we reached the end of a rank //
      if File >= 8
      {
         // reset the file to the starting value //
         File = 0

         // decrement rank to go to the next rank on the board //
         Rank -= 1
      }
   }

   // if the fen start move number is nil //
   if Int(Attributes.FENStartMoveNumber) == nil
   {
      // set the fen start move number to be one //
      Attributes.FENStartMoveNumber = "1"
   }
   // if the fen start move number is out of range //
   else if Int(Attributes.FENStartMoveNumber)! < 1 || Int(Attributes.FENStartMoveNumber)!
           > 1000
   {
      // set the fen start move number to be one //
      Attributes.FENStartMoveNumber = "1"
   }

   // initialize the castles //
   InitializeCastles()

   // if the fen color is not white or black //
   if Attributes.FENColor != "w" && Attributes.FENColor != "b"
   {
      // set the default fen color to be white //
      Attributes.FENColor = "w"
   }

   // if the fen string not valid //
   if !ValidFENString
   {
      // set the default fen strings //
      SetDefaultFENStrings()

      // set the initial layout again //
      SetInitialLayout()
   }
   // if the fen layout is not legal //
   else if !FENLayoutIsLegal()
   {
      // set the default fen string //
      SetDefaultFENStrings()

      // set the initial layout again //
      SetInitialLayout()
   }
}

// initialize the castles //
func InitializeCastles()
{
   // set the white kingside castle to false //
   WhiteKingsideCastle = false

   // set the white queenside castle to false //
   WhiteQueensideCastle = false

   // set the black kingside castle to false //
   BlackKingsideCastle = false

   // set the black queenside castle to false //
   BlackQueensideCastle = false

   // loop over the castling rights //
   for Index in 0 ..< Attributes.FENCastlingRights.count
   {
      // if have a white kingside castle in castling rights //
      if Attributes.FENCastlingRights.prefix(Index + 1).last! == "K"
      {
         // set white kingside castle to true //
         WhiteKingsideCastle = true
      }
      // if have white queenside castle in castling rights //
      else if Attributes.FENCastlingRights.prefix(Index + 1).last! == "Q"
      {
         // set white queenside castle to true //
         WhiteQueensideCastle = true
      }
      // if have black kingside castle in castling rights //
      else if Attributes.FENCastlingRights.prefix(Index + 1).last! == "k"
      {
         // set black kingside castle to true //
         BlackKingsideCastle = true
      }
      // if have black queenside castle in castling rights //
      else if Attributes.FENCastlingRights.prefix(Index + 1).last! == "q"
      {
         // set black queenside castle to true //
         BlackQueensideCastle = true
      }
   }
}

// create the square roughness and border and circles //
func CreateSquareRoughnessAndBorderAndCircles()
{
   // initialize the light square roughness image //
   SquareRoughnessLight = NSImage(size:NSSize(width:SquareSize, height:SquareSize))

   // initialize the dark square roughness image //
   SquareRoughnessDark = NSImage(size:NSSize(width:SquareSize, height:SquareSize))

   // initialize the square border image //
   SquareBorderImage = NSImage(size:NSSize(width:SquareSize, height:SquareSize))

   // create the board tint white image //
   BoardTintWhiteImage = NSImage(size:NSSize(width:SquareSize, height:SquareSize))

   // create the board tint black image //
   BoardTintBlackImage = NSImage(size:NSSize(width:SquareSize, height:SquareSize))

   // lock the light square roughness focus //
   SquareRoughnessLight.lockFocus()

   // loop over the roughness x index //
   for RoughnessXIndex in 0 ..< 25
   {
      // loop over the roughness y index //
      for RoughnessYIndex in 0 ..< 25
      {
         // if is an even square //
         if (RoughnessXIndex + RoughnessYIndex) % 2 == 0
         {
            // append a circle into the roughness circles //
            RoughnessCircles.appendOval(in:NSMakeRect(CGFloat(RoughnessXIndex * 3),
                                                      CGFloat(RoughnessYIndex * 3), 3, 3))
         }
      }
   }

   // set the stroke color to black //
   NSColor.black.set()

   // draw the roughness circles //
   RoughnessCircles.stroke()

   // unlock the light square roughness focus //
   SquareRoughnessLight.unlockFocus()

   // lock the dark square roughness focus //
   SquareRoughnessDark.lockFocus()

   // set the stroke color to white //
   NSColor.white.set()

   // draw the roughness circles //
   RoughnessCircles.stroke()

   // unlock the dark square roughness focus //
   SquareRoughnessDark.unlockFocus()

   // update the chess board roughness layer //
   BoardLayers[7] = CreateChessBoardGrid(SquareRoughnessLight, Dark:SquareRoughnessDark)

   // lock the square border focus //
   SquareBorderImage.lockFocus()

   // set the stroke color to black //
   NSColor.black.set()

   // draw a black square //
   NSBezierPath(rect:NSMakeRect(0, 0, CGFloat(SquareSize), CGFloat(SquareSize))).stroke()

   // set the stroke color to white //
   NSColor.white.set()

   // draw a white square //
   NSBezierPath(rect:NSMakeRect(1, 1, CGFloat(SquareSize - 2), CGFloat(SquareSize - 2)))
      .stroke()

   // set the stroke color to gray //
   NSColor.gray.set()

   // draw a gray square //
   NSBezierPath(rect:NSMakeRect(2, 2, CGFloat(SquareSize - 4), CGFloat(SquareSize - 4)))
      .stroke()

   // unlock the square border focus //
   SquareBorderImage.unlockFocus()

   // update the square border layer //
   BoardLayers[8] = CreateChessBoardGrid(SquareBorderImage, Dark:SquareBorderImage)

   // lock the board tint white image focus //
   BoardTintWhiteImage.lockFocus()

   // set the background color to white //
   NSColor.white.setFill()

   // fill rectangle with the background color //
   NSBezierPath(rect:NSMakeRect(0, 0, CGFloat(SquareSize), CGFloat(SquareSize))).fill()

   // unlock the board tint white image focus //
   BoardTintWhiteImage.unlockFocus()

   // lock the board tint black image focus //
   BoardTintBlackImage.lockFocus()

   // set the background color to black //
   NSColor.black.setFill()

   // fill the rectangle with the background color //
   NSBezierPath(rect:NSMakeRect(0, 0, CGFloat(SquareSize), CGFloat(SquareSize))).fill()

   // unlock the board tint black image focus //
   BoardTintBlackImage.unlockFocus()

   // update the chess board tint contrast layer //
   BoardLayers[6] = CreateChessBoardGrid(BoardTintWhiteImage, Dark:BoardTintBlackImage)

   // create the last move highlight image //
   LastMoveHighlight = NSImage(size:NSSize(width:SquareSize, height:SquareSize))

   // lock the last move highlight focus //
   LastMoveHighlight.lockFocus()

   // set the last move highlight color //
   NSColor(red:1, green:1, blue:0.65, alpha:1).setFill()

   // fill in the last move highlight color into the image //
   NSBezierPath(ovalIn:NSMakeRect(4, 4, CGFloat(SquareSize - 8), CGFloat(SquareSize - 8)))
      .fill()

   // unlock the last move highlight image //
   LastMoveHighlight.unlockFocus()

   // set the size of an illustration piece to hold the last move highlight //
   IllustrationPiece = NSImage(size:NSSize(width:Int(IllustrationSquareSize / WindowScale),
                               height:Int(IllustrationSquareSize / WindowScale)))

   // lock the illustration piece focus //
   IllustrationPiece.lockFocus()

   // draw the last move highlight into the illustration piece //
   LastMoveHighlight.draw(in:CGRect(x:0, y:0, width:IllustrationSquareSize / WindowScale,
                          height:IllustrationSquareSize / WindowScale))

   // unlock the illustration piece focus //
   IllustrationPiece.unlockFocus()

   // set the illustration last move highlight //
   IllustrationLastMoveHighlight = IllustrationPiece.cgImage(forProposedRect:nil, context:nil,
                                                             hints:nil)!

   // create the selected square highlight image //
   SelectedSquareHighlight = NSImage(size:NSSize(width:SquareSize, height:SquareSize))

   // lock the selected square highlight focus //
   SelectedSquareHighlight.lockFocus()

   // set the selected square highlight color //
   NSColor(red:0.2, green:0.7, blue:1, alpha:1).setFill()

   // fill in the selected square highlight color into the image //
   NSBezierPath(rect:NSMakeRect(1, 1, CGFloat(SquareSize - 2), CGFloat(SquareSize - 2))).fill()

   // unlock the selected square highlight image //
   SelectedSquareHighlight.unlockFocus()

   // create the captured highlight image //
   CapturedHighlight = NSImage(size:NSSize(width:SquareSize, height:SquareSize))

   // lock the captured highlight focus //
   CapturedHighlight.lockFocus()

   // set the captured highlight color //
   NSColor(red:1, green:0.3, blue:0.3, alpha:1).setFill()

   // fill in the captured highlight color into the image //
   NSBezierPath(ovalIn:NSMakeRect(2, 2, CGFloat(SquareSize - 4), CGFloat(SquareSize - 4)))
      .fill()

   // unlock the captured highlight image //
   CapturedHighlight.unlockFocus()

   // create the threatened highlight image //
   ThreatenedHighlight = NSImage(size:NSSize(width:SquareSize, height:SquareSize))

   // lock the threatened highlight focus //
   ThreatenedHighlight.lockFocus()

   // set the threatened highlight color //
   NSColor(red:1, green:0.6, blue:0, alpha:1).setFill()

   // fill in the threatened highlight color into the image //
   NSBezierPath(ovalIn:NSMakeRect(2, 2, CGFloat(SquareSize - 4), CGFloat(SquareSize - 4)))
      .fill()

   // unlock the threatened highlight image //
   ThreatenedHighlight.unlockFocus()

   // set the progress image size //
   ProgressImage = NSImage(size:NSSize(width:50, height:50))

   // lock the progress image focus //
   ProgressImage.lockFocus()

   // set the progress circle color //
   NSColor(red:0, green:0.6, blue:1, alpha:1).setFill()

   // fill in the progress circle with the specified color //
   NSBezierPath(ovalIn:NSMakeRect(0, 0, 50, 50)).fill()

   // unlock the progress image focus //
   ProgressImage.unlockFocus()

   // add the progress circle to the view //
   self.layer?.addSublayer(ProgressCircle)

   // set the progress circle bounds //
   ProgressCircle.bounds = CGRect(x:0, y:0, width:50, height:50)

   // set the progress circle position //
   ProgressCircle.position = CGPoint(x:330, y:330)

   // disable the layer animation //
   CATransaction.setDisableActions(true)

   // set the progress circle contents to the progress image //
   ProgressCircle.contents = ProgressImage

   // disable the layer animation //
   CATransaction.setDisableActions(true)

   // initialize the progress circle opacity to zero //
   ProgressCircle.opacity = 0

   // create the enpassant highlight image //
   EnpassantHighlight = NSImage(size:NSSize(width:SquareSize, height:SquareSize))

   // lock the enpassant highlight focus //
   EnpassantHighlight.lockFocus()

   // set the enpassant highlight color //
   NSColor(red:0.45, green:0.45, blue:1, alpha:1).setFill()

   // fill in the en passant highlight color into the image //
   NSBezierPath(ovalIn:NSMakeRect(2, 2, CGFloat(SquareSize - 4), CGFloat(SquareSize - 4)))
      .fill()

   // unlock the en passant highlight image //
   EnpassantHighlight.unlockFocus()

   // set the position editor toolbar item color for en passant square //
   PositionEditorEP.image = EnpassantHighlight
}

// composite the top and bottom board layers //
func CompositeTopBottomBoardLayers(_ Layer:Int, Span:Int)
{
   // initialize the bottom board layer //
   BottomBoardLayer = NSImage(size:NSSize(width:660, height:660))

   // initialize the top board layer //
   TopBoardLayer = NSImage(size:NSSize(width:660, height:660))

   // lock the bottom board layer focus //
   BottomBoardLayer.lockFocus()

   // loop over the bottom board layers //
   for Index in 0 ..< Layer
   {
      // draw the bottom board layers into the bottom board layer //
      BoardLayers[Index].draw(in:self.bounds, from:NSZeroRect, operation:NSCompositingOperation
                              .sourceOver, fraction:CGFloat(BoardLayerOpacities[Index]))
   }

   // unlock the bottom board layer focus //
   BottomBoardLayer.unlockFocus()

   // lock the top board layer focus //
   TopBoardLayer.lockFocus()

   // loop over the top board layers //
   for Index in Layer + Span ..< BoardLayers.count
   {
      // draw the top board layers into the top board layer //
      BoardLayers[Index].draw(in:self.bounds, from:NSZeroRect, operation:NSCompositingOperation
                              .sourceOver, fraction:CGFloat(BoardLayerOpacities[Index]))
   }

   // unlock the top board layer focus //
   TopBoardLayer.unlockFocus()
}

// create the illustration sized board composite //
func CreateIllustrationSizedBoardComposite()
{
   // create a sized board composite for the illustration size //
   CreateSizedBoardComposite(IllustrationSize)

   // set the illustration sized board composite //
   IllustrationSizedBoardComposite = SizedBoardComposite.cgImage(forProposedRect:nil, context:
                                                                 nil, hints:nil)!
}

// create the sized board composite //
func CreateSizedBoardComposite(_ Width:CGFloat)
{
   // initialize the sized board composite //
   SizedBoardComposite = NSImage(size:NSSize(width:Width, height:Width))

   // lock the sized board composite focus//
   SizedBoardComposite.lockFocus()

   // draw the board composite into the sized board composite //
   BoardComposite.draw(in:NSMakeRect(0, 0, Width, Width))

   // unlock the sized board composite focus //
   SizedBoardComposite.unlockFocus()
}

// update board preview //
func UpdateBoardPreview()
{
   // take a board snapshot of size 400 //
   TakeBoardSnapshot(400)

   // set the board preview image to the current board snapshot //
   BoardPreview.image = SnapshotImage
}

// recomposite the board //
func RecompositeBoard(_ Layer:Int, Span:Int, Width:CGFloat)
{
   // if the layer number or the number of layers changing are different //
   if Layer != PreviousLayer || Span != PreviousSpan
   {
      // composite the top and bottom board layers //
      CompositeTopBottomBoardLayers(Layer, Span:Span)

      // save the previous layer number //
      PreviousLayer = Layer

      // save the previous layer span //
      PreviousSpan = Span
   }

   // initialize the board composite //
   BoardComposite = NSImage(size:NSSize(width:Width, height:Width))

   // lock the board composite focus //
   BoardComposite.lockFocus()

   // if have a bottom board composite layer //
   if Layer != 0
   {
      // draw the bottom board layer into the board composite //
      BottomBoardLayer.draw(in:NSMakeRect(0, 0, Width, Width))
   }

   // loop over the board layers that changed //
   for Index in Layer ..< Layer + Span
   {
      // draw the board layers for the given span into the board composite //
      BoardLayers[Index].draw(in:NSMakeRect(0, 0, Width, Width), from:NSZeroRect, operation:
                              NSCompositingOperation.sourceOver, fraction:
                              CGFloat(BoardLayerOpacities[Index]))
   }

   // if have a top board composite layer //
   if Layer != BoardLayers.count - 1
   {
      // draw the top board layer into the board composite //
      TopBoardLayer.draw(in:NSMakeRect(0, 0, Width, Width))
   }

   // unlock the board composite focus //
   BoardComposite.unlockFocus()

   // create the illustration sized board composite //
   CreateIllustrationSizedBoardComposite()

   // set that needs to update the display //
   self.needsDisplay = true

   // display if needed //
   displayIfNeeded()

   // if entering preferences //
   if EnteringPreferences
   {
      // update the board preview //
      UpdateBoardPreview()
   }
}

// create the chess board layer images //
func CreateBoardLayerImages()
{
   // loop over all nine board layers //
   for _ in 0 ... 8
   {
      // append a board layer to the board layers //
      BoardLayers.append(NSImage(size:NSSize(width:660, height:660)))

      // append an opacity of one to the board layer opacities //
      BoardLayerOpacities.append(1)
   }
}

// initialize the images, core animation layers, and the sounds for the game //
func InitializeMedia()
{
   // set the window scale //
   WindowScale = MainWindow.backingScaleFactor

   // set the illustration pieces layer offset to account for the border of the board //
   IllustrationOffset = WindowScale * 30.0 / 660.0 * IllustrationSize

   // set the illustration square size //
   IllustrationSquareSize = WindowScale * (IllustrationSize - 2 * 30.0 / 660.0 *
                            IllustrationSize) / 8.0

   // set the illustration context //
   IllustrationContext = CGContext(data:nil, width:Int(IllustrationSize * WindowScale), height:
                                   Int(IllustrationSize * WindowScale), bitsPerComponent:8,
                                   bytesPerRow:0, space:CGColorSpaceCreateDeviceRGB(),
                                   bitmapInfo:CGImageAlphaInfo.premultipliedFirst.rawValue)

   // set the game annotation text field font to be system font size 18 //
   GameAnnotationTextField.font = NSFont.systemFont(ofSize:18)

   // set the game annotation text field alignment to be justified //
   GameAnnotationTextField.alignment = .justified

   // create the board image and animation layers //
   CreateBoardLayerImages()

   // the app name and version //
   AppNameView.stringValue = "GreenChess 8.0.2"

   // set load position window to close when quitting application //
   LoadPositionWindow.preventsApplicationTerminationWhenModal = false

   // set timed game window to close when quitting application //
   TimedGameWindow.preventsApplicationTerminationWhenModal = false

   // set fen window to close when quitting application //
   FENWindow.preventsApplicationTerminationWhenModal = false

   // set game gallery window to close when quitting application //
   GameGalleryWindow.preventsApplicationTerminationWhenModal = false

   // set game grid window to close when quitting application //
   GameGridWindow.preventsApplicationTerminationWhenModal = false

   // set enter game info window to close when quitting application //
   EnterGameInfoWindow.preventsApplicationTerminationWhenModal = false

   // set about window to close when quitting application //
   AboutWindow.preventsApplicationTerminationWhenModal = false

   // set preferences window to close when quitting application //
   PreferencesWindow.preventsApplicationTerminationWhenModal = false

   // set game list window to close when quitting application //
   GameListWindow.preventsApplicationTerminationWhenModal = false

   // set game explorer window to close when quitting application //
   GameExplorerWindow.preventsApplicationTerminationWhenModal = false

   // set game annotator window to close when quitting application //
   GameAnnotatorWindow.preventsApplicationTerminationWhenModal = false

   // set database explorer window to close when quitting application //
   DatabaseExplorerWindow.preventsApplicationTerminationWhenModal = false

   // set main window to close when quitting application //
   MainWindow.preventsApplicationTerminationWhenModal = false

   // add the default toolbar items //
   AddDefaultItems()

   // remove all the board labels font items from the board labels font button //
   BoardLabelsFontButton.removeAllItems()

   // remove all the move list font items from the move list font button //
   MoveListFontButton.removeAllItems()

   // remove all the game info font items from the game info font button //
   GameInfoFontButton.removeAllItems()

   // remove all the game list font items from the game list font button //
   GameListFontButton.removeAllItems()
```

```
      // remove all the board texture items from the board textures button //
      BoardTexturesButton.removeAllItems()

      // remove all the piece items from the piece set button //
      PieceSetButton.removeAllItems()

      // loop over all the piece sets //
      for Index in 0 ..< ChessPieceSetNames.count
      {
         // add the piece set item to the piece set button //
         PieceSetButton.addItem(withTitle:ChessPieceSetNames[Index])
      }

      // loop over all the chess board names //
      for Index in 0 ..< ChessBoardNames.count
      {
         // add the board texture item to the board textures button //
         BoardTexturesButton.addItem(withTitle:ChessBoardNames[Index])
      }

      // loop over all the fonts //
      for Index in 0 ..< Fonts.count
      {
         // add the item to the board labels font button //
         BoardLabelsFontButton.addItem(withTitle:Fonts[Index])

         // add the item to the move list font button //
         MoveListFontButton.addItem(withTitle:Fonts[Index])

         // add the item to the game info font button //
         GameInfoFontButton.addItem(withTitle:Fonts[Index])

         // add the item to the game list font button //
         GameListFontButton.addItem(withTitle:Fonts[Index])
      }

      // set the rotation with perspective value to 0.004 //
      RotationWithPerspective.m34 = 0.004

      // add the annotation board layer left to the annotation view //
      AnnotationView.layer?.addSublayer(AnnotationBoardLayerLeft)

      // add the annotation board layer middle to the annotation view //
      AnnotationView.layer?.addSublayer(AnnotationBoardLayerMiddle)

      // add the annotation board layer right to the annotation view //
      AnnotationView.layer?.addSublayer(AnnotationBoardLayerRight)

      // add the annotation pieces layer left to the annotation view //
      AnnotationView.layer?.addSublayer(AnnotationPiecesLayerLeft)

      // add the annotation pieces layer middle to the annotation view //
      AnnotationView.layer?.addSublayer(AnnotationPiecesLayerMiddle)

      // add the annotation pieces layer right to the annotation view //
      AnnotationView.layer?.addSublayer(AnnotationPiecesLayerRight)

      // set the annotation board layer middle bounds //
      AnnotationBoardLayerMiddle.bounds = CGRect(x:0, y:0, width:250, height:250)

      // set the annotation board layer middle position //
      AnnotationBoardLayerMiddle.position = CGPoint(x:330, y:125)

      // set the annotation board layer left bounds //
      AnnotationBoardLayerLeft.bounds = CGRect(x:0, y:0, width:250, height:250)

      // set the annotation board layer left position //
      AnnotationBoardLayerLeft.position = CGPoint(x:204, y:125)

      // set the annotation board layer left anchor point //
      AnnotationBoardLayerLeft.anchorPoint = CGPoint(x:1, y:0.5)

      // set the annotation board layer left transform for 3d rotation with perspective //
      AnnotationBoardLayerLeft.transform = CATransform3DRotate(RotationWithPerspective, 0.35, 0,
                                                                1, 0)

      // set the annotation board layer right bounds //
      AnnotationBoardLayerRight.bounds = CGRect(x:0, y:0, width:250, height:250)

      // set the annotation board layer right position //
      AnnotationBoardLayerRight.position = CGPoint(x:456, y:125)

      // set the annotation board layer right anchor point //
      AnnotationBoardLayerRight.anchorPoint = CGPoint(x:0, y:0.5)

      // set the annotation board layer right transform for 3d rotation with perspective //
      AnnotationBoardLayerRight.transform = CATransform3DRotate(RotationWithPerspective, -0.35, 0,
                                                                 1, 0)

      // set the annotation pieces layer middle bounds //
      AnnotationPiecesLayerMiddle.bounds = CGRect(x:0, y:0, width:250, height:250)

      // set the annotation pieces layer middle position //
      AnnotationPiecesLayerMiddle.position = CGPoint(x:330, y:125)

      // set the annotation pieces layer left bounds //
      AnnotationPiecesLayerLeft.bounds = CGRect(x:0, y:0, width:250, height:250)

      // set the annotation pieces layer left position //
      AnnotationPiecesLayerLeft.position = CGPoint(x:204, y:125)

      // set the annotation pieces layer left anchor point //
      AnnotationPiecesLayerLeft.anchorPoint = CGPoint(x:1, y:0.5)

      // set the annotation pieces layer left transform for 3d rotation with perspective //
      AnnotationPiecesLayerLeft.transform = CATransform3DRotate(RotationWithPerspective, 0.35, 0,
                                                                 1, 0)

      // set the annotation pieces layer right bounds //
      AnnotationPiecesLayerRight.bounds = CGRect(x:0, y:0, width:250, height:250)

      // set the annotation pieces layer right position //
      AnnotationPiecesLayerRight.position = CGPoint(x:456, y:125)

      // set the annotation pieces layer right anchor point //
      AnnotationPiecesLayerRight.anchorPoint = CGPoint(x:0, y:0.5)

      // set the annotation pieces layer right transform for 3d rotation with perspective //
      AnnotationPiecesLayerRight.transform = CATransform3DRotate(RotationWithPerspective, -0.35, 0,
                                                                  1, 0)

      // add the first position layer to the load position view //
      LoadPositionView.layer?.addSublayer(PositionLayers[0])

      // add the second position layer to the load position view //
      LoadPositionView.layer?.addSublayer(PositionLayers[1])

      // set the size of position layer one //
      PositionLayers[0].bounds = CGRect(x:0, y:0, width:500, height:500)

      // set position layer one border width //
      PositionLayers[0].borderWidth = 1

      // set position layer one border color to white //
      PositionLayers[0].borderColor = NSColor.white.cgColor

      // set position layer animation time duration //
      PositionAnimation.duration = 1

      // add the animation layer to the game gallery view //
      GameGalleryView.layer?.addSublayer(GameGalleryView.AnimationLayer)

      // add the animation layer to the game grid view //
      GameGridView.layer?.addSublayer(GameGridView.AnimationLayer)

      // create the game clock //
      GameClock.CreateGameClock()

      // create the game clock preview //
      GameClockPreview.CreateGameClock()

      // set that nsview wants a core animation layer //
      self.wantsLayer = true

      // set the move animation delegate //
      MoveAnimation.delegate = self

      // loop over all the light chess square names //
      for Index in 0 ..< ChessLightSquareNames.count
      {
         // add the corresponding chess square image from the chess square names //
         ChessLightSquareImages.append(NSImage(named:NSImage.Name(rawValue:
                                      ChessLightSquareNames[Index]))!)
      }

      // loop over all the dark chess square names //
      for Index in 0 ..< ChessDarkSquareNames.count
      {
         // add the corresponding chess square image from the chess square names //
         ChessDarkSquareImages.append(NSImage(named:NSImage.Name(rawValue:
                                     ChessDarkSquareNames[Index]))!)
      }

      // loop over all the chess piece names //
      for Index in 0 ..< ChessPieceNames.count
      {
         // add the corresponding chess piece image from the chess piece names //
         ChessPieceImages.append(NSImage(named:NSImage.Name(rawValue:ChessPieceNames[Index]))!)

         // create an illustration piece //
         IllustrationPiece = NSImage(size:NSSize(width:Int(IllustrationSquareSize / WindowScale),
                                                 height:Int(IllustrationSquareSize / WindowScale)))

         // lock the illustration piece focus //
         IllustrationPiece.lockFocus()

         // draw the current chess piece image into the illustration piece //
         ChessPieceImages.last?.draw(in:CGRect(x:0, y:0, width:IllustrationSquareSize /
                                     WindowScale, height:IllustrationSquareSize / WindowScale))

         // unlock the illustration piece focus //
         IllustrationPiece.unlockFocus()

         // append the cgimage version of the piece image //
         ChessPieceCGImages.append((IllustrationPiece.cgImage(forProposedRect:nil, context:nil,
                                   hints:nil))!)
      }

      // loop over all the game sounds //
      for Index in 0 ..< GameSoundNames.count
      {
         // add the corresponding game sound from the game sound names //
         GameSounds.append(NSSound(named:NSSound.Name(rawValue:GameSoundNames[Index]))!)
      }

      // save the chess piece images to the captured white pieces view //
      CapturedPiecesWhiteView.SetChessPieceSetImages(ChessPieceImages)

      // save the chess piece images to the captured black pieces view //
      CapturedPiecesBlackView.SetChessPieceSetImages(ChessPieceImages)

      // initialize color set index to 0 //
      Index = 0

      // loop over red color //
      for Red in 0 ..< 6
      {
         // loop over green color //
         for Green in 0 ..< 6
         {
            // loop over blue color //
            for Blue in 0 ..< 6
            {
               // the window color scaling value //
               ColorScaling = CGFloat(Index) / 215

               // append the rainbow background color //
               RainbowBackgroundColors.append(NSColor(red:0.7 + CGFloat(Red) / 5.0 * 0.3, green:
                                              0.7 + CGFloat(Green) / 5.0 * 0.3, blue:0.7 +
                                              CGFloat(Blue) / 5.0 * 0.3, alpha:1))

               // append the game clock colors //
               GameClockColors.append(NSColor(red:CGFloat(Red) / 5.0, green:CGFloat(Green) /
                                      5.0, blue:CGFloat(Blue) / 5.0, alpha:1))

               // append the board tint colors //
               BoardTintColors.append(NSColor(red:CGFloat(Red) / 5.0, green:CGFloat(Green)
                                      / 5.0, blue:CGFloat(Blue) / 5.0, alpha:1))

               // append the font color //
               FontColors.append(NSColor(red:CGFloat(Red) / 5.0 * 0.7, green:CGFloat(Green) /
                                 5.0 * 0.7, blue:CGFloat(Blue) / 5.0 * 0.7, alpha:1))

               // append the blue background color //
               BlueBackgroundColors.append(NSColor(red:0, green:0.1 + ColorScaling * 0.7, blue:1,
                                           alpha:1))

               // append the green background color //
               GreenBackgroundColors.append(NSColor(red:0.1, green:0.3 + ColorScaling * 0.5,
                                            blue:0.3 + ColorScaling * 0.4, alpha:1))

               // append the yellow background color //
               YellowBackgroundColors.append(NSColor(red:1, green:0.95, blue:0.2 + ColorScaling *
                                             0.65, alpha:1))

               // append the gray background color //
               GrayBackgroundColors.append(NSColor(red:0.25 + ColorScaling * 0.75, green:0.25 +
                                           ColorScaling * 0.75, blue:0.25 + ColorScaling * 0.75,
                                           alpha:1))

               // append the brown background color //
               BrownBackgroundColors.append(NSColor(red:0.45 + ColorScaling * 0.35, green:0.4 +
                                            ColorScaling * 0.35, blue:0.35 + ColorScaling * 0.35,
                                            alpha:1))

               // append the teal background color //
               TealBackgroundColors.append(NSColor(red:0, green:0.5 + ColorScaling * 0.4, blue:
                                           0.5 + ColorScaling * 0.4, alpha:1))

               // increment the color set index //
               Index += 1
            }
         }
      }

      // loop over all the border colors //
      for Index in 0 ... 100
      {
         // append a color to the chess board border colors //
         BorderColors.append(NSColor(red:sin(CGFloat(Index) * CGFloat.pi / 200) / 2, green:
                             sin(CGFloat(Index) * CGFloat.pi / 200 + CGFloat.pi / 4) / 2, blue:
                             sin(CGFloat(Index) * CGFloat.pi / 200 + CGFloat.pi / 2) / 2, alpha:1))
      }

      // mark board labels color view as for board labels color //
      BoardLabelsColorView.InLabelColors = true

      // mark clock color view as for the clock border colors //
      ClockColorView.InGameClockColors = true

      // mark clock background color view as for the background colors //
      ClockBackgroundColorView.InGameClockBackgroundColors = true

      // mark game info color view as for game info color //
      GameInfoColorView.InInfoColors = true

      // mark move list color view as for the move list color //
      MoveListColorView.InMoveListColors = true

      // mark game list color view as for the game list color //
      GameListColorView.InGameListColors = true

      // set the rainbow background view color set to 1 //
      RainbowBackgroundView.ColorSet = 1

      // set the blue background view color set to 2 //
      BlueBackgroundView.ColorSet = 2

      // set the brown background view color set to 3 //
      BrownBackgroundView.ColorSet = 3

      // set the green background view color set to 4 //
      GreenBackgroundView.ColorSet = 4

      // set the teal background view color set to 5 //
      TealBackgroundView.ColorSet = 5

      // set the yellow background view color set to 6 //
      YellowBackgroundView.ColorSet = 6

      // set the gray background view color set to 7 //
      GrayBackgroundView.ColorSet = 7

      // create the square roughness and border and circles //
      CreateSquareRoughnessAndBorderAndCircles()

      // create the chess board images //
      CreateChessBoardImages()

      // create the chess board border overlay //
      CreateBorderOverlay()

      // create the window background overlay //
      CreateWindowBackgroundOverlay()
   }

   // update the chess board background color layer //
   func UpdateBackgroundColorLayer()
   {
      // create the chess board background color layer //
      BoardLayers[0] = NSImage(size:NSSize(width:660, height:660))

      // lock the chess board background color image //
      BoardLayers[0].lockFocus()

      // set the chess board background color to the main window color //
      MainWindow.backgroundColor.setFill()

      // fill the image with the background color //
      NSBezierPath(rect:self.bounds).fill()

      // unlock the board image background layer focus //
      BoardLayers[0].unlockFocus()
   }

   // update the board labels layer //
   func UpdateBoardLabelsLayer()
   {
      // create the board labels image //
      BoardLayers[3] = NSImage(size:NSSize(width:660, height:660))

      // lock the board labels layer focus //
      BoardLayers[3].lockFocus()

      // set the board labels font size from the board labels font sizes value //
      BoardLabelsSize = CGFloat(16 + 4 * BoardLabelsSizes)

      // set the file labels offset //
      FileLabelOffset = CGFloat(30 * SquareSize / 2)

      // set the rank labels offset //
      RankLabelOffset = CGFloat(30 * SquareSize / 2 - Int(BoardLabelsSize) / 2)

      // get the board label font //
      BoardLabelFont = GetFont(BoardLabelsFontType, Size:BoardLabelsSize, Bold:BoardLabelsBold)

      // set the board label attributes //
      BoardLabelsAttributes = [NSAttributedStringKey.font:BoardLabelFont, NSAttributedStringKey
                              .foregroundColor:FontColors[LabelColor]]

      // loop over all the board labels //
      for LabelIndex in 0 ..< 8
      {
         // if the board is rotated //
         if RotateBoard
         {
            // set the file label for the board //
            FileLabel = NSAttributedString(string:FileSymbols[7 - LabelIndex], attributes:
                                           BoardLabelsAttributes as? [NSAttributedStringKey:Any])

            // set the rank label for the board //
            RankLabel = NSAttributedString(string:RankSymbols[7 - LabelIndex], attributes:
                                           BoardLabelsAttributes as? [NSAttributedStringKey:Any])
         }
         // the board is not rotated //
         else
         {
            // set the file label for the board //
            FileLabel = NSAttributedString(string:FileSymbols[LabelIndex], attributes:
                                           BoardLabelsAttributes as? [NSAttributedStringKey:Any])

            // set the rank label for the board //
            RankLabel = NSAttributedString(string:RankSymbols[LabelIndex], attributes:
                                           BoardLabelsAttributes as? [NSAttributedStringKey:Any])
         }

         // draw the file label at the specified point //
         FileLabel.draw(at:NSPoint(x:CGFloat(LabelIndex * SquareSize) + FileLabelOffset - FileLabel
                        .size().width / 2, y:15 - BoardLabelsSize / 1.8))

         // draw the rank label at the specified point //
         RankLabel.draw(at:NSPoint(x:15 - RankLabel.size().width / 2, y:CGFloat(LabelIndex *
                        SquareSize) + RankLabelOffset))
      }

      // unlock the board labels layer focus //
      BoardLayers[3].unlockFocus()
   }

   // create the chess board border overlay //
   func CreateBorderOverlay()
   {
      // lock the border overlay image //
      BoardLayers[2].lockFocus()

      // set the border overlay color to dark gray //
      NSColor(red:0.25, green:0.25, blue:0.25, alpha:1).set()

      // loop over fifteen rectangles //
      for Index in 0 ..< 14
      {
         // draw the rectangle //
         NSBezierPath(rect:NSMakeRect(CGFloat(2 * Index + 1), CGFloat(2 * Index + 1), 660 - 2 *
                      CGFloat(2 * Index + 1), 660 - 2 * CGFloat(2 * Index + 1))).stroke()
      }

      // unlock the border overlay focus //
      BoardLayers[2].unlockFocus()
   }

   // create a chess board grid from light and dark squares //
   func CreateChessBoardGrid(_ Light:NSImage, Dark:NSImage) -> NSImage
   {
      // create the layer image //
      LayerImage = NSImage(size:NSSize(width:660, height:660))

      // lock the layer image focus //
      LayerImage.lockFocus()

      // loop over all files //
      for File in 0 ..< 8
      {
         // loop over all ranks //
         for Rank in 0 ..< 8
         {
            // if is a dark square //
            if (Rank + File) % 2 == 0
            {
               // draw the dark square //
               Dark.draw(in:NSMakeRect(30 + CGFloat(File * SquareSize), 30 + CGFloat(Rank *
                         SquareSize), CGFloat(SquareSize), CGFloat(SquareSize)), from:NSZeroRect,
                         operation:NSCompositingOperation.sourceOver, fraction:1)
            }
            // if is a light square //
            else
            {
               // draw the light square //
               Light.draw(in:NSMakeRect(30 + CGFloat(File * SquareSize), 30 + CGFloat(Rank *
                          SquareSize), CGFloat(SquareSize), CGFloat(SquareSize)), from:NSZeroRect,
                          operation:NSCompositingOperation.sourceOver, fraction:1)
            }
         }
      }

      // unlock the layer image focus //
      LayerImage.unlockFocus()

      // return the layer image //
      return LayerImage
   }

   // create the chess board images //
   func CreateChessBoardImages()
   {
      // loop over all the types of chess board square //
      for Index in 0 ..< ChessLightSquareImages.count
      {
         // create a grid of light and dark chess board square //
         ChessBoardImages.append(CreateChessBoardGrid(ChessLightSquareImages[Index], Dark:
                                 ChessDarkSquareImages[Index]))
      }
   }

   // get the font from the specified font type, size, and bold setting //
   func GetFont(_ Type:Int, Size:CGFloat, Bold:Bool) -> NSFont
   {
      // if font is bold //
      if Bold
      {
         // assign bold to the font bold string //
         FontBoldString = " Bold"
      }
      // font is not bold //
      else
      {
         // clear the font bold string //
         FontBoldString = ""
      }

      // if the font exists //
      if NSFont(name:Fonts[Type] + FontBoldString, size:Size) != nil
      {
         // return the font //
         return NSFont(name:Fonts[Type] + FontBoldString, size:Size)!
      }
      // the font doesn't exist //
      else
      {
         // if the font is bold //
         if Bold
         {
            // return a default bold system font //
            return NSFont.boldSystemFont(ofSize:Size)
         }
         // the font is not bold //
         else
         {
            // return a default system font //
            return NSFont.boldSystemFont(ofSize:Size)
         }
      }
   }

   // update the illustration pieces layer //
   func UpdateIllustrationPiecesLayer()
   {
      // draw the illustration sized board composite //
      IllustrationContext?.draw(IllustrationSizedBoardComposite, in:CGRect(x:0, y:0, width:
                                IllustrationSize * WindowScale, height:IllustrationSize *
                                WindowScale), byTiling:false)

      // if need to highlight last move made //
      if StartSquare != EndSquare
      {
         // if the board is rotated //
         if RotateBoard
         {
            // draw the illustration last move highlight for the start square //
            IllustrationContext?.draw(IllustrationLastMoveHighlight, in:CGRect(x:CGFloat(7 -
                                      Int(StartSquare.x)) * IllustrationSquareSize +
                                      IllustrationOffset, y:CGFloat(7 - Int(StartSquare.y)) *
                                      IllustrationSquareSize + IllustrationOffset, width:
                                      IllustrationSquareSize, height:IllustrationSquareSize),
                                      byTiling:false)

            // draw the illustration last move highlight for the end square //
            IllustrationContext?.draw(IllustrationLastMoveHighlight, in:CGRect(x:CGFloat(7 -
                                      Int(EndSquare.x)) * IllustrationSquareSize +
                                      IllustrationOffset, y:CGFloat(7 - Int(EndSquare.y)) *
                                      IllustrationSquareSize + IllustrationOffset, width:
                                      IllustrationSquareSize, height:IllustrationSquareSize),
                                      byTiling:false)
         }
         // board is not rotated //
         else
         {
            // draw the illustration last move highlight for the start square //
            IllustrationContext?.draw(IllustrationLastMoveHighlight, in:CGRect(x:
                                      CGFloat(StartSquare.x) * IllustrationSquareSize +
                                      IllustrationOffset, y:CGFloat(StartSquare.y) *
                                      IllustrationSquareSize + IllustrationOffset, width:
                                      IllustrationSquareSize, height:IllustrationSquareSize),
                                      byTiling:false)

            // draw the illustration last move highlight for the end square //
            IllustrationContext?.draw(IllustrationLastMoveHighlight, in:CGRect(x:
                                      CGFloat(EndSquare.x) * IllustrationSquareSize +
                                      IllustrationOffset, y:CGFloat(EndSquare.y) *
                                      IllustrationSquareSize + IllustrationOffset, width:
                                      IllustrationSquareSize, height:IllustrationSquareSize),
                                      byTiling:false)
         }
      }

      // loop over all files //
      for File in 0 ..< 8
      {
         // loop over all ranks //
         for Rank in 0 ..< 8
         {
            // set the illustration piece type //
            IllustrationPieceType = Layout[File][Rank]

            // if there is a piece on this square //
            if IllustrationPieceType != 0
            {
               // if piece is white //
               if IllustrationPieceType > 0
               {
                  // set the chess piece image index corresponding to piece and chess piece set //
                  ChessPieceImageIndex = IllustrationPieceType - 1 + ChessPieceSetType * 12
               }
               // piece is black //
               else
               {
                  // set the chess piece image index corresponding to piece and chess piece set //
                  ChessPieceImageIndex = abs(IllustrationPieceType) + 5 + ChessPieceSetType * 12
               }

               // if the board is rotated //
               if RotateBoard
               {
                  // draw the illustration piece on the rotated board //
                  IllustrationContext?.draw(ChessPieceCGImages[ChessPieceImageIndex], in:CGRect(x:
                                            CGFloat(7 - File) * IllustrationSquareSize +
                                            IllustrationOffset, y:CGFloat(7 - Rank) *
                                            IllustrationSquareSize + IllustrationOffset, width:
                                            IllustrationSquareSize, height:IllustrationSquareSize),
                                            byTiling:false)
               }
               // board is not rotated //
               else
               {
                  // draw the illustration piece on the unrotated board //
                  IllustrationContext?.draw(ChessPieceCGImages[ChessPieceImageIndex], in:CGRect(x:
                                            CGFloat(File) * IllustrationSquareSize +
                                            IllustrationOffset, y:CGFloat(Rank) *
                                            IllustrationSquareSize + IllustrationOffset,
                                            width:IllustrationSquareSize, height:
                                            IllustrationSquareSize), byTiling:false)
               }
            }
         }
      }

      // convert the illustration pieces layer into an nsimage //
      Illustrations.append(NSImage(cgImage:(IllustrationContext?.makeImage())!, size:NSSize(width:
                           IllustrationSize, height:IllustrationSize)))
   }

   // update the pieces layer //
   func UpdatePiecesLayer(_ Width:CGFloat)
   {
      // set the pieces layer offset to account for the border of the board //
      PiecesLayerOffset = 30.0 / 660.0 * Width

      // set the pieces layer square size //
      PiecesLayerSquareSize = (Width - 2 * PiecesLayerOffset) / 8

      // initialize the pieces layer //
      PiecesLayer = NSImage(size:NSSize(width:Width, height:Width))

      // lock the pieces layer focus //
      PiecesLayer.lockFocus()

      // if in the position editor //
      if InPositionEditor
      {
         // if the fen enpassant square is not empty //
         if Attributes.FENEnpassantSquare != "-"
         {
            // display the enpassant highlight image //
            DisplayImage(EnpassantSquareFile, Rank:EnpassantSquareRank, Image:EnpassantHighlight)
         }
      }

      // if chose to highlight captured pieces //
      if HighlightCapturedPieces && (ReviewingGame && !InPositionEditor && !CoreAnimating &&
         Width != IllustrationSize
      {
         // loop over the list of squares where captures can occur //
         for ListIndex in 0 ..< ListOfCaptureSquares.count
         {
            // display the captured highlight image on this square //
            DisplayImage(Int(ListOfCaptureSquares[ListIndex]) % 8, Rank:
                         Int(ListOfCaptureSquares[ListIndex]) / 8, Image:CapturedHighlight)
         }
      }

      // if chose to highlight threatened pieces //
      if HighlightThreatenedPieces && !ReviewingGame && !InPositionEditor && !CoreAnimating &&
         Width != IllustrationSize
      {
         // loop over the list of squares where pieces can be threatened //
         for ListIndex in 0 ..< ListOfThreatenSquares.count
         {
            // display the threatened highlight image on this square //
            DisplayImage(Int(ListOfThreatenSquares[ListIndex]) % 8, Rank:
                         Int(ListOfThreatenSquares[ListIndex]) / 8, Image:ThreatenedHighlight)
         }
      }

      // if need to highlight last move made //
      if (StartSquare != EndSquare) && ((HighlightLastMoveMade && !InPositionEditor) ||
         CreatingGalleryGrid || Width == IllustrationSize || AnnotatingGame)
      {
         // display the last move made image on this square //
         DisplayImage(Int(StartSquare.x), Rank:Int(StartSquare.y), Image:LastMoveHighlight)

         // display the last move made image on this square //
         DisplayImage(Int(EndSquare.x), Rank:Int(EndSquare.y), Image:LastMoveHighlight)
      }

      // if move by selecting squares and have a selected square //
      if MoveBySelectingSquares && SelectedSquares > 0
      {
         // display the selected square image //
         DisplayImage(Int(RankAndFile.x), Rank:Int(RankAndFile.y), Image:SelectedSquareHighlight)
      }

      // loop over all files //
      for File in 0 ..< 8
      {
         // loop over all ranks //
         for Rank in 0 ..< 8
         {
            // if there is a piece on this square //
            if Layout[File][Rank] != 0
            {
               // display the piece currently on this square //
               DisplayImage(File, Rank:Rank, Image:GetPieceImage(Layout[File][Rank]))
            }
         }
      }

      // unlock the pieces layer focus //
      PiecesLayer.unlockFocus()
   }

   // display the image //
   func DisplayImage(_ File:Int, Rank:Int, Image:NSImage)
   {
      // if the board has been rotated //
      if RotateBoard
      {
         // draw the image for the piece on this given square //
         Image.draw(in:NSMakeRect(CGFloat(7 - File) * PiecesLayerSquareSize + PiecesLayerOffset,
                    CGFloat(7 - Rank) * PiecesLayerSquareSize + PiecesLayerOffset,
                    PiecesLayerSquareSize, PiecesLayerSquareSize), from:NSZeroRect, operation:
                    NSCompositingOperation.sourceOver, fraction:1)
      }
      // the board has not been rotated //
      else
      {
         // draw the image for the piece on this given square //
         Image.draw(in:NSMakeRect(CGFloat(File) * PiecesLayerSquareSize + PiecesLayerOffset,
                    CGFloat(Rank) * PiecesLayerSquareSize + PiecesLayerOffset,
                    PiecesLayerSquareSize, PiecesLayerSquareSize), from:NSZeroRect, operation:
                    NSCompositingOperation.sourceOver, fraction:1)
      }
   }

   // keep the marker pen in bounds //
   func KeepMarkerInBounds()
   {
      // if the x coordinate is less than 30 //
      if MouseLocation.x < 30
      {
         // reset the mouse location x coordinate to 32 //
         MouseLocation.x = 32
      }

      // if the y coordinate is less than 30 //
      if MouseLocation.y < 30
      {
         // reset the mouse location y coordinate to 32 //
         MouseLocation.y = 32
      }

      // if the x coordinate is greater than 630 //
      if MouseLocation.x > 630
      {
         // reset the mouse location x coordinate to 628 //
         MouseLocation.x = 628
      }

      // if the y coordinate is greater than 630 //
      if MouseLocation.y > 630
      {
         // reset the mouse location y coordinate to 628 //
         MouseLocation.y = 628
      }
   }

   // update the mouse cursor //
   func UpdateCursor(_ Cursor:NSPoint, Moving:Bool)
   {
      // if the mouse location is off the board or have attached sheet //
      if Cursor.x < 30 || Cursor.x > 630 || Cursor.y < 30 || Cursor.y > 630 || HaveAttachedSheet()
      {
         // set the mouse to the arrow cursor //
         NSCursor.arrow.set()
      }
      // the mouse location is on the board //
      else
      {
         // if in diagramming mode //
         if Diagramming
         {
            // set the cursor to the diagram cursor //
            DiagramCursor.set()
         }
         // not in diagramming mode and mouse down //
         else if Moving
         {
            // set the mouse to the pointing hand cursor //
            NSCursor.pointingHand.set()
         }
         // if in the position editor //
         else if InPositionEditor
         {
            // set current cursor to the position cursor //
            PosCursor.set()
         }
         // not in diagramming mode and not mouse down //
         else
         {
            // set the mouse to the open hand cursor //
            NSCursor.openHand.set()
         }
      }
   }

   // if a thread is running //
   func ThreadIsRunning() -> Bool
   {
      // return whether a thread is running or not //
      return SaveGameMovieThread.isExecuting || SaveGalleryGridMovieThread.isExecuting ||
             GalleryGridThread.isExecuting || OpenGameThread.isExecuting
   }

   // function to intercept a mouse down event //
   override func mouseDown(with Event:NSEvent)
   {
      // update the mouse cursor type //
      UpdateCursor(Event.locationInWindow, Moving:true)

      // if moving by selecting squares and not done selecting squares //
      if MoveBySelectingSquares && SelectedSquares == 1
      {
         // increment the selected squares //
         SelectedSquares += 1

         // exit function since still selecting destination square //
         return
      }

      // set that piece not initially valid //
      ValidPiece = false

      // set that not currently a valid draw shape //
      ValidShape = false

      // if a thread is running //
      if ThreadIsRunning()
      {
         // exit function since a thread is running //
         return
      }

      // get the mouse location //
      MouseLocation = Event.locationInWindow

      // if the mouse location is not on the chess board //
      if MouseLocation.x < 30 || MouseLocation.y < 30 || MouseLocation.x > 630 || MouseLocation.y >
         630
      {
         // exit the function since mouse location is not within the chess board //
         return
      }

      // if in diagramming mode //
      if Diagramming
      {
         // set that the current shape is valid //
         ValidShape = true

         // play the sound of a marker pen starting to write //
         PlayGameSoundThread("marker")

         // keep the marker in bounds //
         KeepMarkerInBounds()

         // create a shape layer //
         Shape = CAShapeLayer()

         // create a new mutable shape path //
         ShapePath = CGMutablePath()

         // set the shape line width //
         Shape.lineWidth = 4

         // set the shape line cap style to round //
         Shape.lineCap = "kCALineCapRound"

         // set the shape line join style to round //
         Shape.lineJoin = "kCALineJoinRound"

         // set the shape stroke color //
         Shape.strokeColor = DiagramColor.cgColor

         // set the shape fill color //
         Shape.fillColor = NSColor.clear.cgColor

         // set the shape path starting point //
         ShapePath.move(to:CGPoint(x:MouseLocation.x, y:MouseLocation.y))

         // assign the shape path to the shape //
         Shape.path = ShapePath

         // append a new shape layer //
         Shapes.append(Shape)

         // add the new shape layer to the view layer //
         self.layer?.addSublayer(Shape)

         // return since done with diagramming portion of mouse down //
         return
      }

      // if shouldn't be accepting any mouse down input from the user //
      if EngineThinking || CoreAnimating || (BoardGameOver(Board) && !InPositionEditor)
      {
         // exit function since engine running or waiting for another process //
         return
      }

      // keep the piece within bounds of board //
      KeepInBounds()

      // get the starting square for the move //
      FromSquare = SquareLabel()

      // if in the position editor //
      if InPositionEditor
      {
         // set the moving piece to be the position editor moving piece //
         MovingPiece = PositionMovingPiece

         // if placing an enpassant square //
         if MovingPiece == 10
         {
            // set the chosen en passant square //
            Attributes.FENEnpassantSquare = FromSquare

            // set the enpassant square file //
            EnpassantSquareFile = Int(RankAndFile.x)

            // set the enpassant square rank //
            EnpassantSquareRank = Int(RankAndFile.y)
         }
         // not placing an en passant square //
         else
         {
            // if moving piece is the eraser //
            if MovingPiece == 0
            {
               // if the eraser lands on the enpassant square //
               if FromSquare == Attributes.FENEnpassantSquare
```

```
            {
                // set the fen enpassant square to - //
                Attributes.FENEnpassantSquare = "-"
            }

            // play the sound of a pop //
            PlayGameSoundThread("pop")
        }
        // placing a chess piece piece //
        else
        {
            // play the sound of a thmp //
            PlayGameSoundThread("thump")
        }

        // add the piece to the chosen square on the board //
        Layout[Int(RankAndFile.x)][Int(RankAndFile.y)] = MovingPiece
    }

    // reset the piece moving to be none //
    MovingPiece = 0

    // update the board and captured pieces //
    UpdateBoardAndCapturedPieces()

    // exit function since done placing piece in position editor //
    return
}

// the piece that the user has chosen to move //
MovingPiece = Layout[Int(RankAndFile.x)][Int(RankAndFile.y)]

// if the color currently moving has been selected to move //
if ((MovingPieceIsWhite() && BoardIsWhite(Board)) || (MovingPieceIsBlack() &&
    BoardIsBlack(Board))) && !EngineThinking
{
    // set that is a valid piece moving //
    ValidPiece = true

    // set that clock has started //
    StartClock = true

    // play the clicking sound of a chess piece being picked up //
    PlayGameSoundThread("click")
}
// if a valid piece hasn't been selected for the move to make //
else
{
    // set that there is no piece currently moving //
    MovingPiece = 0

    // exit from function since there is no valid piece to move //
    return
}

// if not moving pieces by selecting squares //
if !MoveBySelectingSquares
{
    // allocate the moving piece layer //
    MovingPieceLayer = CALayer()

    // set the moving piece layer bounds //
    MovingPieceLayer.bounds = CGRect(x:0, y:0, width:CGFloat(SquareSize), height:
                              CGFloat(SquareSize))

    // remove the animation effects //
    CATransaction.setDisableActions(true)

    // set the moving piece layer position //
    MovingPieceLayer.position = CGPoint(x:MouseLocation.x - 0.5, y:MouseLocation.y - 0.5)

    // assign the piece moving to the moving piece layer //
    MovingPieceLayer.contents = GetPieceImage(MovingPiece)

    // set the moving piece layer shadow opacity to have a shadow //
    MovingPieceLayer.shadowOpacity = 1

    // add the moving piece layer to the board //
    self.layer?.addSublayer(MovingPieceLayer)

    // erase the piece from the starting square since it is moving now //
    Layout[Int(RankAndFile.x)][Int(RankAndFile.y)] = 0

    // update the board and captured pieces //
    UpdateBoardAndCapturedPieces()
}
// moving pieces by selecting squares //
else
{
    // increment the selected squares //
    SelectedSquares += 1

    // update the board and captured pieces //
    UpdateBoardAndCapturedPieces()

    // erase the piece from the starting square since it is moving now //
    Layout[Int(RankAndFile.x)][Int(RankAndFile.y)] = 0
}
}

// function to intercept a mouse drag //
override func mouseDragged(with Event:NSEvent)
{
    // update the mouse cursor type //
    UpdateCursor(Event.locationInWindow, Moving:true)

    // get the current location of the mouse //
    MouseLocation = Event.locationInWindow

    // if in diagramming mode and is a valid shape //
    if Diagramming && ValidShape
    {
        // keep the marker pen in bounds //
        KeepMarkerInBounds()

        // add the line to the shape path //
        ShapePath.addLine(to:CGPoint(x:MouseLocation.x, y:MouseLocation.y))

        // assign the shape path to the shape //
        Shapes[Shapes.count - 1].path = ShapePath

        // return since done with diagramming portion of mouse dragged //
        return
    }

    // if piece trying to move is not a valid piece or moving by selecting squares //
    if !ValidPiece || MoveBySelectingSquares
    {
        // exit since another process running or waiting for a process to finish //
        return
    }

    // keep the mouse location located inside the board //
    KeepInBounds()

    // remove the animation effects //
    CATransaction.setDisableActions(true)

    // set the moving piece layer position //
    MovingPieceLayer.position = CGPoint(x:MouseLocation.x - 0.5, y:MouseLocation.y - 0.5)
}

// perform a castling move if needed //
func PerformCastleIfNeeded()
{
    // if there was a white kingside castle //
    if (String(validatingUTF8:BoardPGNLabel(Board)) == "O-O" ||
        String(validatingUTF8:BoardPGNLabel(Board)) == "O-O+" ||
        String(validatingUTF8:BoardPGNLabel(Board)) == "O-O#") && BoardIsBlack(Board)
    {
        // assign UCI notation for white kingside castle //
        UCIMove = "h1f1"
    }
    // if there was a white queenside castle //
    else if (String(validatingUTF8:BoardPGNLabel(Board)) == "O-O-O" ||
             String(validatingUTF8:BoardPGNLabel(Board)) == "O-O-O+" ||
             String(validatingUTF8:BoardPGNLabel(Board)) == "O-O-O#") && BoardIsBlack(Board)
    {
        // assign UCI notation for white queenside castle //
        UCIMove = "a1d1"
    }
    // if there was a black kingside castle //
    else if (String(validatingUTF8:BoardPGNLabel(Board)) == "O-O" ||
             String(validatingUTF8:BoardPGNLabel(Board)) == "O-O+" ||
             String(validatingUTF8:BoardPGNLabel(Board)) == "O-O#") && BoardIsWhite(Board)
    {
        // assign UCI notation for black kingside castle //
        UCIMove = "h8f8"
    }
    // if there was a black queenside castle //
    else if (String(validatingUTF8:BoardPGNLabel(Board)) == "O-O-O" ||
             String(validatingUTF8:BoardPGNLabel(Board)) == "O-O-O+" ||
             String(validatingUTF8:BoardPGNLabel(Board)) == "O-O-O#") && BoardIsWhite(Board)
    {
        // assign UCI notation for black queenside castle //
        UCIMove = "a8d8"
    }
    // move was not a castling move //
    else
    {
        // exit function since was not a castling move //
        return
    }

    // convert the UCI move to origin and destination squares //
    ProcessUCIMove()

    // put the piece that moved onto the destination square //
    UpdateDestinationSquare()
}

// perform an en passant pawn move and capture if needed //
func PerformEnpassantMoveIfNeeded()
{
    // if an en passant move occurred //
    if BoardIsEnpassant(Board)
    {
        // remove the pawn captured en passant //
        Layout[Int(EndSquare.x)][Int(StartSquare.y)] = 0
    }
}

// the engine match menu item action //
@IBAction func EngineMatchMenuItemAction(_:AnyObject)
{
    // turn on the engine match menu item //
    EngineMatchMenuItem.state = NSControl.StateValue(rawValue:1)

    // set that clock has started //
    StartClock = true

    // set the current engine to uci chess engine //
    Opponent(false, Engine:false, UCIEngine:true)

    // make an engine match move to have the uci engine self play //
    MakeEngineMatchMove()
}

// start a move by the chess engine //
func StartEngineMove()
{
    // set that engine is currently thinking //
    EngineThinking = true

    // start the progress indicator //
    AnimateProgress = true

    // if currently running in the uci chess engine mode //
    if UCIEngineOpponent
    {
        // clear the stockfish pv line //
        StockfishPV = ""

        // if will be using the encyclopedia of chess openings book //
        if UseEcoBook
        {
            // wait until the entire chess opening book is done being read in //
            while ReadOpeningBookThread.isExecuting
            {
            }
        }

        // if a move exists for the current board position and using the opening book //
        if OpeningBookMoveExists(OpeningBook, BoardGetZobristHash(Board)) && UseEcoBook
        {
            // get the move from the chess opening book //
            UCIMove = String(validatingUTF8:OpeningBookGetNextMove(OpeningBook))!

            // if found the move from the opening book on the board //
            if BoardFindMove(Board, UCIMove)
            {
                // copy the opening book move to the engine message //
                EngineMessageTokens = ["bestmove", UCIMove]

                // set that used an opening book move //
                StockfishAnalysisString = "Opening Book Move"

                // run on the main thread //
                DispatchQueue.main.async
                {
                    // perform the engine move //
                    self.PerformEngineMove()
                }

                // exit function since found opening book move //
                return
            }
        }

        // set the uci threads to use option //
        EngineCommand = "setoption name Threads value \(ThreadsToUse)\n"

        // set the uci engine difficulty option //
        EngineCommand += "setoption name Skill Level value \(UCIEngineDifficulty)\n"

        // set the uci engine hash option //
        EngineCommand += "setoption name Hash value \(UCIEngineHash)\n"

        // set the uci engine riskiness option //
        EngineCommand += "setoption name Contempt value \(UCIEngineRiskiness)\n"

        // set the uci engine starting position and moves //
        EngineCommand += "position fen \(Attributes.FENLayout) \(Attributes.FENColor) " +
                         "\(Attributes.FENCastlingRights)"
                         + " \(Attributes.FENEnpassantSquare) 0 1 moves"

        // loop over all moves in the move list //
        for Index in 0 ..< MoveList.count
        {
            // if the move is not an invalid move //
            if MoveList[Index] != "Invalid Move"
            {
                // add the move made to the engine command //
                EngineCommand += " \(MoveList[Index])"
            }
        }

        // add a newline to the engine command string //
        EngineCommand += "\n"

        // if playing against uci engine //
        if UCIEngineOpponent
        {
            // write fen string to engine //
            WriteToEngine(EngineCommand)

            // if playing a timed game //
            if PlayingTimedGame
            {
                // tell engine to make a move in 5 seconds //
                WriteToEngine("go movetime 5000\n")
            }
            // not playing a timed game //
            else
            {
                // tell uci engine to make a move //
                WriteToEngine("go movetime \(SearchTime * 1000)\n")
            }
        }
    }
    // not running in the uci chess engine mode //
    else
    {
        // create the chess engine thread //
        GCEngineThread = Thread(target:self, selector:#selector(GreenChess.GCEngineMove), object:
                         nil)

        // set the chess engine thread to the highest priority //
        GCEngineThread.threadPriority = 1

        // start the chess engine thread //
        GCEngineThread.start()
    }
}

// undo the illegal move //
func UndoIllegalMove()
{
    // set that will be undoing an illegal move //
    UndoingIllegalMove = true

    // assign the start square to a temporary square //
    TempSquare = StartSquare

    // assign the end square to the start square //
    StartSquare = EndSquare

    // assign the start square to the end square //
    EndSquare = TempSquare

    // run core animation of piece moving back to original position //
    RunCoreAnimation()
}

// function to intercept a mouse up //
override func mouseUp(with Event:NSEvent)
{
    // update the mouse cursor type //
    UpdateCursor(Event.locationInWindow, Moving:false)

    // get the current location of the mouse //
    MouseLocation = Event.locationInWindow

    // if in diagramming mode and is a valid shape //
    if Diagramming && ValidShape
    {
        // keep the marker pen in bounds //
        KeepMarkerInBounds()

        // add the line to the shape path //
        ShapePath.addLine(to:CGPoint(x:MouseLocation.x, y:MouseLocation.y))

        // assign the shape path to the shape //
        Shapes[Shapes.count - 1].path = ShapePath

        // set that no longer a valid shape //
        ValidShape = false

        // return since done with mouse up portion of diagramming mode //
        return
    }

    // if mouse location is out of range and moving by selecting squares //
    if (MouseLocation.x < 30 || MouseLocation.y < 30 || MouseLocation.x > 630 || MouseLocation.y
        > 630) && (MoveBySelectingSquares && SelectedSquares == 2)
    {
        // clear the selected square if needed //
        ClearSelectedSquareIfNeeded()

        // exit function since went off the board when selecting squares //
        return
    }

    // if user not moving valid piece or moving by selecting squares and selected first square //
    if !ValidPiece || (MoveBySelectingSquares && SelectedSquares == 1)
    {
        // exit since game is over or tried to move an invalid piece //
        return
    }

    // reset whether a valid piece or not //
    ValidPiece = false

    // reset selected squares to zero //
    SelectedSquares = 0

    // keep the chess piece inside the board //
    KeepInBounds()

    // if not moving by selecting squares //
    if !MoveBySelectingSquares
    {
        // remove the animation effects //
        CATransaction.setDisableActions(true)

        // set the moving piece layer position //
        MovingPieceLayer.position = CGPoint(x:MouseLocation.x - 0.5, y:MouseLocation.y - 0.5)

        // remove the moving piece layer from the nsview //
        MovingPieceLayer.removeFromSuperlayer()
    }

    // get the ending square for the move //
    ToSquare = SquareLabel()

    // the chess move in UCI notation //
    UCIMove = FromSquare + ToSquare

    // convert the UCI move the start and end squares //
    UCIToSquares()

    // if illegal move //
    if !(BoardFindMove(Board, UCIMove) || BoardFindMove(Board, UCIMove + "q"))
    {
        // play the sound of an illegal move being made //
        PlayGameSoundThread("wrongsquare")

        // undo the illegal move //
        UndoIllegalMove()

        // exit function since had an illegal move //
        return
    }

    // add current move to the move list //
    MoveList.append(UCIMove)

    // add a placeholder annotation to the annotation list //
    AnnotationList.append("")

    // promote the pawn to another piece if had a pawn promotion //
    PromotePawnIfNeeded()

    // if not waiting for pawn promotion to complete //
    if !QueryPawnPromotion
    {
        // finish up the current move //
        FinishUpMove()
    }
}

// play a game sound thread //
func PlayGameSoundThread(_ Sound:String)
{
    // if the sound is currently turned off //
    if SoundOff
    {
        // exit function since sound was turned off by the user //
        return
    }

    // stop the current game sound if it was already playing //
    GameSound.stop()

    // wait until the current game sound thread is done executing //
    while GameSoundThread.isExecuting
    {
    }

    // if the game sound to be play is a thump sound //
    if Sound == "thump"
    {
        // set the game sound selection and volume //
        SetGameSoundSelection(0, Volume:1)
    }
    // if the game sound to play is a wrong square sound //
    else if Sound == "wrongsquare"
    {
        // set the game sound selection and volume //
        SetGameSoundSelection(1, Volume:0.25)
    }
    // if the game sound to play is a pop sound //
    else if Sound == "pop"
    {
        // set the game sound selection and volume //
        SetGameSoundSelection(2, Volume:0.4)
    }
    // if the game sound to play is a click sound //
    else if Sound == "click"
    {
        // set the game sound selection and volume //
        SetGameSoundSelection(3, Volume:0.10)
    }
    // if the game sound to play is a clear sound //
    else if Sound == "clear"
    {
        // set the game sound selection and volume //
        SetGameSoundSelection(4, Volume:0.25)
    }
    // if the game sound to play is a warning sound //
    else if Sound == "warning"
    {
        // set the game sound selection and volume //
        SetGameSoundSelection(5, Volume:0.3)
    }
    // if the game sound to play is a switch sound //
    else if Sound == "switch"
    {
        // set the game sound selection and volume //
        SetGameSoundSelection(6, Volume:0.4)
    }
    // if the game sound to play is a button click sound //
    else if Sound == "buttonclick"
    {
        // set the game sound selection and volume //
        SetGameSoundSelection(7, Volume:0.3)
    }
    // if the game sound to play is a camera taking a photo //
    else if Sound == "camera"
    {
        // set the game sound selection and volume //
        SetGameSoundSelection(8, Volume:0.7)
    }
    // if the game sound to play is a marker pen starting to write //
    else if Sound == "marker"
    {
        // set the game sound selection and volume //
        SetGameSoundSelection(9, Volume:0.7)
    }
    // if the game sound to play is a movie icon being clicked //
    else if Sound == "movie"
    {
        // set the game sound selection and volume //
        SetGameSoundSelection(10, Volume:0.3)
    }

    // create the game sound thread //
    GameSoundThread = Thread(target:self, selector:#selector(GreenChess.PlayGameSound), object:
                      nil)

    // set the game sound thread priority //
    GameSoundThread.threadPriority = 1

    // start the game sound thread //
    GameSoundThread.start()
}

// set the game sound selection and the volume //
func SetGameSoundSelection(_ Selection:Int, Volume:Float)
{
    // set the game sound selection //
    GameSoundSelection = Selection

    // set the sound volume //
    GameSoundVolume = Volume
}

// play the chosen game sound //
@objc func PlayGameSound()
{
    // get the selected game sound from the game sound list //
    GameSound = GameSounds[GameSoundSelection]

    // set the volume of the game sound //
    GameSound.volume = GameSoundVolume

    // play the game sound //
    GameSound.play()
}

// finish up the current move //
func FinishUpMove()
{
    // store the autosave data //
    StoreAutosaveData("")

    // make the move on the board //
    GCMakeMoveOnBoard(MoveList[MoveList.count - 1])

    // save the previous move list //
    PreviousMoveList = MoveList

    // save the previous annotation list //
    PreviousAnnotationList = AnnotationList

    // update the moving piece to the new location on the board //
    UpdateDestinationSquare()

    // perform an en passant move and capture if needed //
    PerformEnpassantMoveIfNeeded()

    // perform castle if needed //
    PerformCastleIfNeeded()

    // update the chessboard //
    UpdateBoardAndCapturedPieces()

    // update the pieces layer for the illustrations //
    UpdateIllustrationPiecesLayer()

    // if didn't have a promotion //
    if !QueryPawnPromotion
    {
        // play the piece thump sound thread //
        PlayGameSoundThread("thump")
    }

    // print the game information to the screen //
    PrintGameInfo()

    // if playing against a chess engine and the game is not already over //
    if !HumanOpponent && !BoardGameOver(Board)
    {
        // start an engine move //
        StartEngineMove()
    }
}

// the show the stockfish analysis button action //
@IBAction func ShowStockfishAnalysisButtonAction(_:AnyObject)
{
    // switch the stockfish analysis option //
    ShowStockfishAnalysis = !ShowStockfishAnalysis

    // play the sound of a button click //
    PlayGameSoundThread("click")

    // print game info //
    PrintGameInfo()
}

// the autoload stockfish button action //
@IBAction func AutoloadStockfishButtonAction(_:AnyObject)
{
    // switch the autoload stockfish option //
    AutoloadStockfish = !AutoloadStockfish

    // play the sound of a button click //
    PlayGameSoundThread("click")
}

// the move by selecting squares action //
@IBAction func MoveBySelectingSquaresAction(_:AnyObject)
{
    // switch the move by selecting squares mode //
    MoveBySelectingSquares = !MoveBySelectingSquares

    // play the sound of a button click //
    PlayGameSoundThread("click")
}

// the game sounds button action //
@IBAction func GameSoundsButtonAction(_:AnyObject)
{
    // switch the sound mode //
    SoundOff = !SoundOff

    // play the sound of a button click //
    PlayGameSoundThread("click")
}

// draw the layer into the image with the specified opacity //
func DrawLayerInImage(_ Image:NSImage, Opacity:CGFloat, Width:CGFloat)
{
    // draw the layer into the image //
    Image.draw(in:NSMakeRect(0, 0, Width, Width), from:NSZeroRect, operation:
               NSCompositingOperation.sourceOver, fraction:Opacity)
}

// override the draw function //
override func draw(_ Rect:NSRect)
{
    // draw the board composite into the view //
    BoardComposite.draw(in:self.bounds)

    // draw the pieces layer into the view //
    PiecesLayer.draw(in:self.bounds)
}

// take a snapshot of the current board layout //
func TakeBoardSnapshot(_ Width:CGFloat)
{
    // set the snapshot image size to the specified width //
    SnapshotImage = NSImage(size:NSMakeSize(Width, Width))

    // if diagramming or entering preferences //
    if Diagramming || EnteringPreferences
    {
        // set the snapshot bitmap to given width //
        SnapshotBitmap = self.bitmapImageRepForCachingDisplay(in:NSMakeRect(0, 0, Width, Width))!

        // set the board snapshot size //
        SnapshotBitmap.size = self.frame.size

        // draw the specified image into the bitmap representation //
        self.cacheDisplay(in:self.bounds, to:SnapshotBitmap)

        // add the board snapshot bitmap to the board snapshot image //
        SnapshotImage.addRepresentation(SnapshotBitmap)
    }
    // if not diagramming or entering preferences //
    else
    {
        // lock the snapshot image focus //
        SnapshotImage.lockFocus()

        // draw the board composite into the image //
        BoardComposite.draw(in:CGRect(x:0, y:0, width:Width, height:Width))

        // draw the pieces layer into the image //
        PiecesLayer.draw(in:CGRect(x:0, y:0, width:Width, height:Width))

        // unlock the snapshot image focus //
        SnapshotImage.unlockFocus()
    }
}

// the diagram blue color action //
@IBAction func DiagramBlueColorAction(_:AnyObject)
{
    // set the diagram color to blue //
    DiagramColor = NSColor.blue

    // play the sound of a button click //
    PlayGameSoundThread("click")

    // set diagram cursor to blue highlighter //
    SetDiagramCursor("blue")
}

// the diagram cyan color action //
@IBAction func DiagramCyanColorAction(_:AnyObject)
{
    // set the diagram color to cyan //
    DiagramColor = NSColor.cyan

    // play the sound of a button click //
    PlayGameSoundThread("click")

    // set diagram cursor to cyan highlighter //
    SetDiagramCursor("cyan")
}

// the diagram magenta color action //
@IBAction func DiagramMagentaColorAction(_:AnyObject)
{
    // set the diagram color to magenta //
    DiagramColor = NSColor.magenta

    // play the sound of a button click //
    PlayGameSoundThread("click")

    // set the diagram cursor to magenta highlighter //
    SetDiagramCursor("magenta")
}

// the diagram white color action //
@IBAction func DiagramWhiteColorAction(_:AnyObject)
{
    // set the diagram color to white //
    DiagramColor = NSColor.white

    // play the sound of a button click //
    PlayGameSoundThread("click")

    // set the diagram cursor to white highlighter //
    SetDiagramCursor("white")
}

// the diagram black color action //
@IBAction func DiagramBlackColorAction(_:AnyObject)
{
    // set the diagram color to black //
    DiagramColor = NSColor.black

    // play the sound of a button click //
    PlayGameSoundThread("click")

    // set the diagram cursor to black //
    SetDiagramCursor("black")
}

// the diagram yellow color action //
@IBAction func DiagramYellowColorAction(_:AnyObject)
{
    // set the diagram color to yellow //
    DiagramColor = NSColor.yellow

    // play the sound of a button click //
    PlayGameSoundThread("click")

    // set the diagram cursor to yellow //
    SetDiagramCursor("yellow")
}

// the diagram green color action //
@IBAction func DiagramGreenColorAction(_:AnyObject)
{
    // set the diagram color to green //
    DiagramColor = NSColor.green

    // play the sound of a button click //
    PlayGameSoundThread("click")

    // set the diagram cursor to green //
    SetDiagramCursor("green")
}

// the diagram red color action //
@IBAction func DiagramRedColorAction(_:AnyObject)
{
    // set the diagram color to red //
    DiagramColor = NSColor.red

    // play the sound of a button click //
    PlayGameSoundThread("click")

    // set the diagram cursor to red //
    SetDiagramCursor("red")
}

// the erase shapes action //
@IBAction func EraseShapes(_:AnyObject)
{
    // play the sound of the diagram being cleared //
    PlayGameSoundThread("clear")

    // erase all the gallery grid view shapes //
    GalleryGridView.EraseShapes()
}

// the undo shape action //
@IBAction func UndoShape(_:AnyObject)
{
    // play the game sound of a click //
    PlayGameSoundThread("click")

    // undo a shape on the gallery grid view //
    GalleryGridView.UndoShape()
}

// the gallery grid blue color action //
@IBAction func GalleryGridBlueColor(_:AnyObject)
{
    // play the game sound of a click //
    PlayGameSoundThread("click")

    // set the gallery grid color to blue //
    GalleryGridColor = NSColor.blue

    // set the gallery grid view diagram cursor to blue highlighter //
    GalleryGridView.SetDiagramCursor("blue")
}

// the gallery grid cyan color action //
@IBAction func GalleryGridCyanColor(_:AnyObject)
{
    // play the game sound of a click //
    PlayGameSoundThread("click")

    // set the gallery grid color to cyan //
    GalleryGridColor = NSColor.cyan

    // set the gallery grid view diagram cursor to cyan highlighter //
    GalleryGridView.SetDiagramCursor("cyan")
}

// the gallery grid yellow color action //
@IBAction func GalleryGridYellowColor(_:AnyObject)
{
    // play the game sound of a click //
    PlayGameSoundThread("click")

    // set the gallery grid color to yellow //
    GalleryGridColor = NSColor.yellow

    // set the gallery grid view diagram cursor to yellow highlighter //
    GalleryGridView.SetDiagramCursor("yellow")
}

// the gallery grid black color action //
@IBAction func GalleryGridBlackColor(_:AnyObject)
{
    // play the game sound of a click //
    PlayGameSoundThread("click")

    // set the gallery grid color to black //
    GalleryGridColor = NSColor.black

    // set the gallery grid view diagram cursor to black highlighter //
    GalleryGridView.SetDiagramCursor("black")
}

// the gallery grid white color action //
@IBAction func GalleryGridWhiteColor(_:AnyObject)
{
    // play the game sound of a click //
    PlayGameSoundThread("click")

    // set the gallery grid color to white //
    GalleryGridColor = NSColor.white

    // set the gallery grid view diagram cursor to white highlighter //
    GalleryGridView.SetDiagramCursor("white")
}

// the gallery grid magenta color action //
@IBAction func GalleryGridMagentaColor(_:AnyObject)
{
    // play the game sound of a click //
    PlayGameSoundThread("click")

    // set the gallery grid color to magenta //
    GalleryGridColor = NSColor.magenta

    // set the gallery grid view diagram cursor to magenta highlighter //
    GalleryGridView.SetDiagramCursor("magenta")
}

// the gallery grid green color action //
@IBAction func GalleryGridGreenColor(_:AnyObject)
{
    // play the game sound of a click //
    PlayGameSoundThread("click")

    // set the gallery grid color to green //
    GalleryGridColor = NSColor.green

    // set the gallery grid view diagram cursor to green highlighter //
    GalleryGridView.SetDiagramCursor("green")
}

// the gallery grid red color action //
@IBAction func GalleryGridRedColor(_:AnyObject)
{
    // play the game sound of a click //
    PlayGameSoundThread("click")

    // set the gallery grid color to red //
    GalleryGridColor = NSColor.red

    // set the gallery grid view diagram cursor to red highlighter //
    GalleryGridView.SetDiagramCursor("red")
}

// take a snapshot of the gallery grid //
@IBAction func GalleryGridSnapshot(_:AnyObject)
{
    // if currently animating //
    if GalleryGridView.Animating
```

```
   {
      // exit function since currently animating //
      return
   }

   // play the game sound of a camera //
   PlayGameSoundThread("camera")

   // close the attached sheet if have one //
   CloseAttachedSheet()

   // create the save panel //
   SavePanel = NSSavePanel()

   // set save panel to close when quitting application //
   SavePanel.preventsApplicationTerminationWhenModal = false

   // set the allowed file types to jpeg //
   SavePanel.allowedFileTypes = ["jpeg"]

   // set the snapshot file name to GreenChessSnapshot //
   SavePanel.nameFieldStringValue = "GreenChessSnapshot"

   // save the gallery grid snapshot //
   SavePanel.beginSheetModal(for:MainWindow, completionHandler:
                             {Result -> Void in self.PlayGameSoundThread("buttonclick")
                              if Result == NSApplication.ModalResponse.OK
                              {self.SaveGalleryGridSnapshot(self.SavePanel.url!.path)} else
                              {self.ReopenPreviouslyAttachedSheet()}})
}

// make a gallery grid movie //
@IBAction func GalleryGridMovie(_:AnyObject)
{
   // if currently animating //
   if GalleryGridView.Animating
   {
      // exit function since currently animating //
      return
   }

   // play the game sound of a movie button //
   PlayGameSoundThread("movie")

   // close the attached sheet if have one //
   CloseAttachedSheet()

   // create the save panel //
   SavePanel = NSSavePanel()

   // set the save panel to close when quitting application //
   SavePanel.preventsApplicationTerminationWhenModal = false

   // set the allowed file types to mov //
   SavePanel.allowedFileTypes = ["mov"]

   // set the movie file name to GreenChessMovie //
   SavePanel.nameFieldStringValue = "GreenChessMovie"

   // save the gallery grid movie //
   SavePanel.beginSheetModal(for:MainWindow, completionHandler:
                             {Result -> Void in self.PlayGameSoundThread("buttonclick")
                              if Result == NSApplication.ModalResponse.OK
                              {self.SaveGalleryGridMovie()} else
                              {self.ReopenPreviouslyAttachedSheet()}})
}

// the action to create a screen recording of a game //
@IBAction func GameExplorerMovieButtonAction(_:AnyObject)
{
   // play the sound of clicking on the game to movie icon //
   PlayGameSoundThread("movie")

   // close the attached sheet if have one //
   CloseAttachedSheet()

   // create a panel to save the game movie //
   SavePanel = NSSavePanel()

   // set the save panel to close when quitting application //
   SavePanel.preventsApplicationTerminationWhenModal = false

   // set the allowed type for a movie to be mov //
   SavePanel.allowedFileTypes = ["mov"]

   // set the default movie name //
   SavePanel.nameFieldStringValue = "GreenChessMovie"

   // start the save panel query for saving a game in movie format //
   SavePanel.beginSheetModal(for:MainWindow, completionHandler:
                             {Result -> Void in self.PlayGameSoundThread("buttonclick")
                              if Result == NSApplication.ModalResponse.OK {self.SaveGameMovie()}
                              else {self.ReopenPreviouslyAttachedSheet()}})
}

// get the pixel buffer from the cg image //
func PixelBufferFromCGImage(_ Image:CGImage) -> CVPixelBuffer
{
   // if the pixel buffer is nil //
   if PixelBuffer == nil
   {
      // create the pixel buffer //
      CVPixelBufferCreate(kCFAllocatorDefault, 2000, 2000, kCVPixelFormatType_32ARGB,
                          ["kCVPixelBufferCGImageCompatibilityKey":true,
                          "kCVPixelBufferCGBitmapContextCompatibilityKey":true] as CFDictionary?,
                          &PixelBuffer)
   }

   // lock the base address //
   CVPixelBufferLockBaseAddress(PixelBuffer!, CVPixelBufferLockFlags(rawValue:CVOptionFlags(0)))

   // draw the image with the context //
   CGContext(data:CVPixelBufferGetBaseAddress(PixelBuffer!), width:2000, height:2000,
             bitsPerComponent:8, bytesPerRow:8000, space:CGColorSpaceCreateDeviceRGB(),
             bitmapInfo:CGImageAlphaInfo.premultipliedFirst.rawValue)?.draw(Image, in:
             CGRect(x:0, y:0, width:2000, height:2000))

   // unlock the base address //
   CVPixelBufferUnlockBaseAddress(PixelBuffer!, CVPixelBufferLockFlags(rawValue:
                                  CVOptionFlags(0)))

   // return the pixel buffer //
   return PixelBuffer!
}

// take a gallery grid snapshot //
func TakeGalleryGridSnapshot(_ Frame:Int)
{
   // create the gallery grid snapshot //
   GalleryGridSnapshot = NSImage(size:NSSize(width:GalleryGridView.Width, height:
                                 GalleryGridView.Height))

   // lock the focus of the gallery grid snapshot //
   GalleryGridSnapshot.lockFocus()

   // draw the given frame into the gallery grid snapshot //
   GalleryGridImage.draw(in:NSMakeRect(CGFloat(-GalleryGridView.FrameWidth * Double(Frame)), 0,
                         CGFloat(GalleryGridImage.size.width), CGFloat(GalleryGridImage.size
                         .height)), from:NSZeroRect, operation:NSCompositingOperation.copy,
                         fraction:1)

   // unlock the gallery grid snapshot focus //
   GalleryGridSnapshot.unlockFocus()

   // convert the gallery grid snapshot into a gallery grid cgimage //
   GalleryGridCGImage = GalleryGridSnapshot.cgImage(forProposedRect:nil, context:nil, hints:nil)
}

// write the gallery grid movie //
@objc func WriteGalleryGridMovie()
{
   // set the number of gallery grid movie frames //
   GalleryGridMovieFrames = Int((Double(GalleryGridImage.size.width) - GalleryGridView.Width) /
                               GalleryGridView.FrameWidth) + 1

   // loop over all the gallery grid movie frames //
   for FrameIndex in 0 ..< GalleryGridMovieFrames
   {
      // take a gallery snapshot for the given frame //
      TakeGalleryGridSnapshot(FrameIndex)

      // append the pixel buffer with frame duration of two seconds //
      Adaptor!.append(PixelBufferFromCGImage(GalleryGridCGImage), withPresentationTime:
                      CMTimeMake(Int64(FrameIndex * 2), 1))
   }

   // mark the video input as finished //
   VidInput!.markAsFinished()

   // finish writing the video //
   VideoWriter!.finishWriting(completionHandler:{() -> Void in})

   // reset the adaptor //
   Adaptor = nil

   // run on the main thread //
   DispatchQueue.main.async
   {
      // reopen the previously attached sheet if had one //
      self.ReopenPreviouslyAttachedSheet()

      // set that done animating progress //
      self.AnimateProgress = false

      // reset the gallery grid title //
      self.MainWindow.title = self.GalleryGridTitle
   }
}

// save the gallery grid snapshot //
func SaveGalleryGridSnapshot(_ Filename:String)
{
   // save the snapshot file name to save //
   SnapshotFilename = Filename

   // add a loop to create the galley grid snapshot //
   RunLoop.current.add(Timer(timeInterval:0.01, target:self, selector:#selector(GreenChess
                       .CreateGalleryGridSnapshot), userInfo:nil, repeats:false), forMode:
                       RunLoopMode.commonModes)
}

// create the gallery grid snapshot //
@objc func CreateGalleryGridSnapshot()
{
   // take a snapshot of the gallery grid view //
   GalleryGridView.TakeSnapshot()

   // write the image to a jpeg file //
   try? NSBitmapImageRep(cgImage:GalleryGridView.SnapshotImage.cgImage(forProposedRect:nil,
                         context:nil, hints:nil)!).representation(using:NSBitmapImageRep
                         .FileType.jpeg, properties:[:])!.write(to:URL(fileURLWithPath:
                         SnapshotFilename), options:[.atomic])

   // reopen the previously attached sheet if have one //
   ReopenPreviouslyAttachedSheet()
}

// save the gallery grid movie //
func SaveGalleryGridMovie()
{
   // if in the game grid //
   if InGameGrid
   {
      // set the gallery grid image to the grid image //
      GalleryGridImage = GalleryGridView.GridImage
   }
   // if in the game gallery //
   else
   {
      // set the gallery grid image to the gallery image //
      GalleryGridImage = GalleryGridView.GalleryImage
   }

   // if the movie file already exists //
   if FileManager.default.fileExists(atPath:SavePanel.url!.path)
   {
      // remove the old movie file since av can't overwrite existing movies //
      try? FileManager.default.removeItem(atPath:SavePanel.url!.path)
   }

   // assign the video writer //
   VideoWriter = try? AVAssetWriter(outputURL:SavePanel.url!, fileType:AVFileType.mov)

   // assign the video input //
   VidInput = AVAssetWriterInput(mediaType:AVMediaType.video, outputSettings:[AVVideoCodecKey:
                                 AVVideoCodecH264, AVVideoWidthKey:NSNumber(value:
                                 GalleryGridView.Width * 2), AVVideoHeightKey:NSNumber(value:
                                 GalleryGridView.Height * 2)])

   // assign the adaptor //
   Adaptor = AVAssetWriterInputPixelBufferAdaptor(assetWriterInput:VidInput!,
             sourcePixelBufferAttributes:[kCVPixelBufferPixelFormatTypeKey as String:
             NSNumber(value:kCVPixelFormatType_32ARGB as UInt32), kCVPixelBufferWidthKey as
             String:NSNumber(value:GalleryGridView.Width * 2), kCVPixelBufferHeightKey as
             String:NSNumber(value:GalleryGridView.Height * 2)])

   // add the video input //
   VideoWriter!.add(VidInput!)

   // start writing the video //
   VideoWriter!.startWriting()

   // start the session at time zero //
   VideoWriter!.startSession(atSourceTime:kCMTimeZero)

   // set title to "saving movie" //
   MainWindow.title = "Saving Movie"

   // start animating progress //
   AnimateProgress = true

   // create the save gallery grid movie thread //
   SaveGalleryGridMovieThread = Thread(target:self, selector:#selector(GreenChess
                                .WriteGalleryGridMovie), object:nil)

   // set the save gallery grid movie thread to the highest priority //
   SaveGalleryGridMovieThread.threadPriority = 1

   // start the save gallery grid movie thread //
   SaveGalleryGridMovieThread.start()
}

// save a game to a movie recording //
func SaveGameMovie()
{
   // set that currently recording a movie of a game being played //
   RecordingMovie = true

   // if the movie file already exists //
   if FileManager.default.fileExists(atPath:SavePanel.url!.path)
   {
      // remove the old movie file since av can't overwrite existing movies //
      try? FileManager.default.removeItem(atPath:SavePanel.url!.path)
   }

   // assign the video writer //
   VideoWriter = try? AVAssetWriter(outputURL:SavePanel.url!, fileType:AVFileType.mov)

   // assign the video input //
   VidInput = AVAssetWriterInput(mediaType:AVMediaType.video, outputSettings:[AVVideoCodecKey:
                                 AVVideoCodecH264, AVVideoWidthKey:NSNumber(value:1320),
                                 AVVideoHeightKey:NSNumber(value:1320)])

   // assign the adaptor //
   Adaptor = AVAssetWriterInputPixelBufferAdaptor(assetWriterInput:VidInput!,
             sourcePixelBufferAttributes:[kCVPixelBufferPixelFormatTypeKey as
             String:NSNumber(value:kCVPixelFormatType_32ARGB as UInt32), kCVPixelBufferWidthKey
             as String:NSNumber(value:1320), kCVPixelBufferHeightKey as String:NSNumber(value:
             1320)])

   // add the video input //
   VideoWriter!.add(VidInput!)

   // start writing the video //
   VideoWriter!.startWriting()

   // start the session at time zero //
   VideoWriter!.startSession(atSourceTime:kCMTimeZero)

   // set that currently animating progress //
   AnimateProgress = true

   // set that currently saving a movie //
   MainWindow.title = "Saving Movie"

   // create the save game movie thread //
   SaveGameMovieThread = Thread(target:self, selector:#selector(GreenChess.WriteGameMovie),
                         object:nil)

   // set the save game movie thread to the highest priority //
   SaveGameMovieThread.threadPriority = 1

   // start the save game movie thread //
   SaveGameMovieThread.start()
}

// write the game movie //
@objc func WriteGameMovie()
{
   // convert the game to frames //
   ConvertGameToFrames()

   // loop over all the frames //
   for FrameIndex in 0 ..< Frames.count
   {
      // append the pixel buffer with duration of two seconds //
      Adaptor!.append(PixelBufferFromCGImage(Frames[FrameIndex].cgImage(forProposedRect:nil,
                      context:nil, hints:nil)!), withPresentationTime:
                      CMTimeMake(Int64(FrameIndex * 2), 1))
   }

   // mark the video input as finished //
   VidInput!.markAsFinished()

   // finish writing the video //
   VideoWriter!.finishWriting(completionHandler:{() -> Void in})

   // set that currently done recording a movie //
   RecordingMovie = false

   // turn off the animate progress //
   AnimateProgress = false

   // reset the adaptor //
   Adaptor = nil

   // run on the main thread //
   DispatchQueue.main.async
   {
      // reopen the previously attached sheet //
      self.ReopenPreviouslyAttachedSheet()

      // update the game explorer //
      self.UpdateGameExplorer(self)

      // set the main window title back to game explorer mode //
      self.MainWindow.title = "Green Chess (Game Explorer Mode)"
   }
}

// the take a board snapshot action //
@IBAction func BoardSnapshotAction(_:AnyObject)
{
   // close the attached sheet if have one //
   CloseAttachedSheet()

   // turn on the board snapshot menu item //
   BoardSnapshotMenuItem.state = NSControl.StateValue(rawValue:1)

   // play the game sound of a camera taking a photo //
   PlayGameSoundThread("camera")

   // take a snapshot of the current board //
   TakeBoardSnapshot(660)

   // create a save panel for the board snapshot image //
   SavePanel = NSSavePanel()

   // set save panel to close when quitting application //
   SavePanel.preventsApplicationTerminationWhenModal = false

   // set the allowed type of file format for a saved board snapshot image //
   SavePanel.allowedFileTypes = ["jpeg"]

   // set the default filename to save board snapshot as //
   SavePanel.nameFieldStringValue = "GreenChessSnapshot"

   // run the panel to save the current board snapshot as jpeg file //
   SavePanel.beginSheetModal(for:MainWindow, completionHandler:
                             {Result -> Void in self.PlayGameSoundThread("buttonclick")
                              if Result == NSApplication.ModalResponse.OK
                              {self.SaveBoardSnapshot(self.SavePanel.url!.path)} else
                              {self.FinishUpBoardSnapshot()}})
}

// save the board snapshot //
func SaveBoardSnapshot(_ Filename:String)
{
   // write the board snapshot //
   try? NSBitmapImageRep(cgImage:SnapshotImage.cgImage(forProposedRect:nil, context:nil, hints:
                         nil)!).representation(using:NSBitmapImageRep.FileType.jpeg, properties
                         :[:])!.write(to:URL(fileURLWithPath:Filename), options:[.atomic])

   // finish up the board snapshot //
   FinishUpBoardSnapshot()
}

// finish up the board snapshot //
func FinishUpBoardSnapshot()
{
   // reopen the previously attached sheet if have one //
   ReopenPreviouslyAttachedSheet()

   // turn off the board snapshot menu item //
   BoardSnapshotMenuItem.state = NSControl.StateValue(rawValue:0)
}

// rotate the board view //
@IBAction func RotateBoardMenuItemAction(_:AnyObject)
{
   // add a timer to rotate the board //
   RunLoop.current.add(Timer(timeInterval:0.01, target:self, selector:#selector(GreenChess
                       .RotateTheBoard), userInfo:nil, repeats:false), forMode:RunLoopMode
                       .commonModes)
}

// rotate the board //
@objc func RotateTheBoard()
{
   // set that board has been rotated //
   RotateBoard = !RotateBoard

   // update the board labels layer //
   UpdateBoardLabelsLayer()

   // recomposite the board for the updated board labels layer //
   RecompositeBoard(3, Span:1, Width:660)

   // replay all the moves //
   ReplayMoves(0, End:MoveList.count, Display:false)

   // update the move list //
   UpdateMoveList()

   // update the state of the rotate board menu item //
   RotateBoardMenuItem.state = NSControl.StateValue(rawValue:BoolToInt(RotateBoard))

   // update the user preference for the rotate board key //
   AppPreferences.set(RotateBoard, forKey:"RotateBoard")
}

// change to currently playing against uci chess engine //
@IBAction func UCIEngineOpponentMenuItemAction(_:AnyObject)
{
   // if the uci engine has not loaded yet //
   if !UCIEngineLoaded
   {
      // load stockfish as uci engine  //
      LoadEngine(Bundle.main.path(forResource:"stockfish-8-64", ofType:"")!)
   }
   // uci engine has already loaded //
   else
   {
      // change to currently playing against uci chess engine //
      Opponent(false, Engine:false, UCIEngine:true)
   }
}

// change to currently playing against the built-in chess engine //
@IBAction func EngineOpponentMenuItemAction(_:AnyObject)
{
   // change to currently playing against the built-in chess engine //
   Opponent(false, Engine:true, UCIEngine:false)
}

// change to currently playing against a human //
@IBAction func HumanOpponentMenuItemAction(_:AnyObject)
{
   // change to currently playing against a human //
   Opponent(true, Engine:false, UCIEngine:false)
}

// save a chess game to the given filename //
func SaveGame(_ Filename:String)
{
   // convert the game to a game data object //
   ConvertToGameData()

   // write the game in pgn notation to the chosen filename //
   DataWritePGN(GameData, Filename, Attributes.FENColor == "b")
}

// construct pgn string representing game //
func GamePGN() -> String
{
   // initialize number of move plies //
   Plies = BoolToInt(Attributes.FENColor == "b")

   // add header information to pgn string //
   PGN = "[Event \"\(Attributes.EventTitle)\"]\n[Site \"\(Attributes.SiteLocation)\"]\n" +
         "[Date \"\(Attributes.DateGamePlayed)\"]\n[Round \"\(Attributes.RoundNumber)\"]\n" +
         "[White \"\(Attributes.WhiteName)\"]\n[Black \"\(Attributes.BlackName)\"]\n" +
         "[Result \"\(Attributes.EndResult)\"]\n[TimeControl \"\(Attributes.TimeControl)\"]\n" +
         "[WhiteElo \"\(Attributes.WhiteElo)\"]\n[BlackElo \"\(Attributes.BlackElo)\"]\n" +
         "[WhiteUSCF \"\(Attributes.WhiteUSCF)\"]\n[BlackUSCF \"\(Attributes.BlackUSCF)\"]\n" +
         "[Annotator \"\(Attributes.Annotator)\"]\n[ECO \"\(Attributes.ECO)\"]\n" +
         "[Opening \"\(Attributes.Opening)\"]\n[Section \"\(Attributes.Section)\"]\n" +
         "[Board \"\(Attributes.BoardNumber)\"]\n[SetUp \"1\"]\n" +
         "[FEN \"\(Attributes.FENLayout) \(Attributes.FENColor) " +
         "\(Attributes.FENCastlingRights) \(Attributes.FENEnpassantSquare) 0 " +
         "\(Attributes.FENStartMoveNumber)\"]\n\n"

   // initialize the game information board //
   BoardInitializeAttributes(GameDataBoard, Attributes.FENLayout, Attributes.FENCastlingRights,
                             Attributes.FENColor, Attributes.FENEnpassantSquare)

   // loop over all moves in the move list //
   for Index in 0 ..< MoveList.count
   {
      // if index is outside the range of the annotation list //
      if Index > AnnotationList.count - 1
      {
         // exit loop since index is outside the range of the annotation list //
         break
      }

      // make the move on the data board //
      GCMakeMoveOnDataBoard(MoveList[Index])

      // if is the first move and black happens to make first move from fen info //
      if Plies == 1 && Attributes.FENColor == "b"
      {
         // add to pgn string that black is moving first such as from fen game //
         PGN += "\(Attributes.FENStartMoveNumber)... "
      }

      // increment plies //
      Plies += 1

      // if starting a new move number //
      if Plies % 2 == 1
      {
         // add move number to pgn string //
         PGN += "\((Plies - 1) / 2 + Int(Attributes.FENStartMoveNumber)!). "
      }

      // add move to pgn string //
      PGN += String(validatingUTF8:BoardPGNLabel(GameDataBoard))!

      // if the move annotation is currently not empty //
      if !AnnotationList[Index].isEmpty
      {
         // add the move annotation to the pgn string //
         PGN += " \(AnnotationList[Index])"
      }

      // if have added another five move sequences or at end of move list //
      if Plies % 10 == 0 || Index == MoveList.count - 1
      {
         // add end of line to pgn string //
         PGN += "\n"
      }
      // need to write a space after move //
      else
      {
         // add space to pgn string //
         PGN += " "
      }
   }

   // if the move list was empty //
   if MoveList.isEmpty
   {
      // store an * to represent no moves in game //
      PGN += "*\n"
   }

   // append result and newline to pgn string //
   PGN += "\(Attributes.EndResult)\n"

   // return the game pgn //
   return PGN
}

// convert the game to a game data object //
func ConvertToGameData()
{
   // initialize the game data //
   DataInitializeAttributes(GameData)

   // create the fen string from the individual fen components //
   FENString = "\(Attributes.FENLayout) \(Attributes.FENColor) \(Attributes.FENCastlingRights)"
               + " \(Attributes.FENEnpassantSquare) 0 \(Attributes.FENStartMoveNumber)"

   // store the game header data in the game data oject //
   DataStoreGameHeader(GameData, Attributes.WhiteName, Attributes.BlackName,
                       Attributes.SiteLocation, Attributes.EventTitle, Attributes.RoundNumber,
                       Attributes.EndResult, Attributes.DateGamePlayed, Attributes.TimeControl,
                       Attributes.WhiteElo, Attributes.BlackElo, Attributes.WhiteUSCF,
                       Attributes.BlackUSCF, Attributes.Annotator, Attributes.ECO,
                       Attributes.Opening, Attributes.Section, Attributes.BoardNumber, FENString)

   // initialize the game information board //
   BoardInitializeAttributes(GameDataBoard, Attributes.FENLayout, Attributes.FENCastlingRights,
                             Attributes.FENColor, Attributes.FENEnpassantSquare)

   // loop over all moves in the move list //
   for Index in 0 ..< MoveList.count
   {
      // if index is outside the range of the annotation list //
      if Index > AnnotationList.count - 1
      {
         // exit loop since index is outside the range of the annotation list //
         break
      }

      // make the move on the data board //
      GCMakeMoveOnDataBoard(MoveList[Index])

      // if the move exists //
      if MoveExists
      {
         // store the current move and annotation into the game data //
         DataStoreMove(GameData, BoardPGNLabel(GameDataBoard), AnnotationList[Index])
      }
   }
}

// load uci chess engine //
func LoadEngine(_ Filename:String)
{
   // set the engine task launch path //
   EngineTask.launchPath = Filename

   // set the engine task standard input //
   EngineTask.standardInput = EnginePipeWrite

   // set the engine task standard output //
   EngineTask.standardOutput = EnginePipeRead

   // set the engine task write file handle //
   EngineFileHandleWrite = EnginePipeWrite.fileHandleForWriting

   // set the engine task read file handle //
   EngineFileHandleRead = EnginePipeRead.fileHandleForReading

   // observe when the engine read file handle has data in it //
   Notify.addObserver(self, selector:#selector(GreenChess.EngineDataReady(_:)), name:FileHandle
                      .readCompletionNotification, object:EngineFileHandleRead)

   // launch the uci chess engine //
   EngineTask.launch()

   // query the chess engine to identify itself //
   WriteToEngine("uci\n")
}

// check if the uci chess engine loaded successfully //
func CheckUCIEngineLoaded()
{
   // if the uci chess engine successfully loaded //
   if UCIEngineLoaded
   {
      // if not autoloading the stockfish engine //
      if !AutoloadStockfish
      {
         // set that querying to see if uci chess engine loaded or not //
         QueryUCIEngineLoad = true

         // play the sound of a switch //
         PlayGameSoundThread("switch")

         // set the alert button to be okay //
         AlertButtons = ["Okay"]

         // alert that uci engine loaded successfully //
         MakeAlert(AlertButtons, Query:"Enabled \(EngineName)")
      }

      // add uci engine to timed game popup button //
      TimedGameOpponentButton.addItem(withTitle:"Stockfish")

      // set the opponent to the uci engine //
      Opponent(false, Engine:false, UCIEngine:true)
   }
}

// append the pgn game to a currently existing pgn game //
func AppendGame(_ Filename:String)
{
   // convert the current game to data format //
   ConvertToGameData()

   // append the game in pgn notation to the chosen filename //
   DataAppendPGN(GameData, Filename, Attributes.FENColor == "b")
}

// run the read the opening book thread //
func RunReadOpeningBookThread()
{
   // create the thread to read in the opening book //
   ReadOpeningBookThread = Thread(target:self, selector:#selector(GreenChess.ReadOpeningBook),
                                  object:nil)

   // set the read opening book thread priority to the maximum value //
   ReadOpeningBookThread.threadPriority = 1

   // start the read opening book thread //
   ReadOpeningBookThread.start()
}

// read the chess opening book //
@objc func ReadOpeningBook()
{
   // clear the opening book move list //
   OpeningBookMoveList.removeAll()

   // initialize the chess opening book //
   OpeningBookInitializeAttributes(OpeningBook)

   // read in the opening book pgn database from the main bundle //
   DatabaseReadGameCollection(OpeningBookGameList, Bundle.main.path(forResource:"eco", ofType:
                              "pgn")!)

   // loop over the entire opening book game list //
   for OpeningBookIndex in 0 ..< Int(DatabaseGetDataListSize(OpeningBookGameList))
   {
      // initialize the opening book board //
      BoardInitializeAttributes(OpeningBoard, DefaultFENLayout, "KQkq", "w", "-")

      // get the move list for the specified game in the opening book game list //
      OpeningBookMoveList = String(validatingUTF8:DatabaseGetGameDataUCI(OpeningBookGameList,
                                   CInt(OpeningBookIndex)))!.components(separatedBy:" ")

      // loop over the move list for this game in the opening book //
      for MoveListCountIndex in 0 ..< OpeningBookMoveList.count
      {
         // get the zobrist hash of the opening book board before a move is made //
         ZobristHash = BoardGetZobristHash(OpeningBoard)

         // find the move on the opening book board //
         BoardFindMove(OpeningBoard, OpeningBookMoveList[MoveListCountIndex])

         // make the move on the opening book board //
         BoardMakeMove(OpeningBoard, 0)

         // store the next move and zobrist hash into the chess opening book //
         OpeningBookStoreOpeningBookData(OpeningBook, ZobristHash, BoardUCILabel(OpeningBoard))
      }
   }
}

// open the game //
@objc func OpenGame()
{
   // create the open game thread //
   OpenGameThread = Thread(target:self, selector:#selector(GreenChess.RunOpenGameThread),
                           object:nil)

   // set the open game thread to the highest priority //
   OpenGameThread.threadPriority = 1

   // start the open game thread //
   OpenGameThread.start()
}

// run the open game thread to open a game file //
@objc func RunOpenGameThread()
{
   // initialize the game data //
   DataInitializeAttributes(GameData)

   // read in a list of pgn games //
   DatabaseReadGameCollection(GameList, GameFilename)

   // store the game list into the game list data //
   GameListData.StoreGameList(String(validatingUTF8:DatabaseGetGameDescriptions(GameList))!
                              .components(separatedBy:"**********"))

   // run on the main thread //
   DispatchQueue.main.async
   {
      // finish opening the game file //
      self.FinishOpeningGame()
   }
}

// finish opening the game //
func FinishOpeningGame()
{
   // set that doesn't allow multiple row selection //
   GameListTableView.allowsMultipleSelection = false

   // if there were valid games read in to the game database //
   if DatabaseGetDataListSize(GameList) > 0
   {
      // set that have now read in the game list //
      ReadInGameList = true
   }

   // if games exist in the game list //
   if DatabaseGetDataListSize(GameList) > 1
   {
      // initialize the previously selected row to zero //
      PreviouslySelectedRow = 0

      // search the game list //
      SearchGameListMenuItemAction(self)
   }
   // if there is only one game in the game list //
   else if DatabaseGetDataListSize(GameList) == 1
   {
      // get the first game from the pgn game list //
      GameData = DatabaseGetGameData(GameData, GameList, 0)

      // display selected game from game list //
      DisplaySelectedGameFromGameList()
   }

   // set the initial value of the database explorer slider //
   DatabaseExplorerSlider.intValue = 0

   // set the minimum value of the database explorer slider //
   DatabaseExplorerSlider.minValue = 0

   // set the database explorer slider maximum value //
   DatabaseExplorerSlider.maxValue = Double(GameListData.DatabaseSize()) - 1

   // set the minimum value of the database game explorer slider //
   DatabaseGameExplorerSlider.minValue = 0

   // clear the search string //
   SearchString = ""

   // update the search string to the game list search box //
   GameListSearchBox.stringValue = SearchString

   // set that done querying whether to open a game file or not //
   QueryOpenGameFile = false
}

// cancel loading a game from the selected game in the game list window //
@IBAction func CancelLoadGameFromListButtonAction(_:AnyObject)
{
   // close the game list window //
   CloseGameListWindow()
}

// cancel loading the game from the database explorer selection //
@IBAction func CancelLoadGameFromDatabaseExplorerButtonAction(_:AnyObject)
{
   // revert the game data to the saved game data //
   DataCopy(GameData, SavedGameData)

   // close the database explorer window //
   CloseDatabaseExplorerWindow()
}

// load the selected game from the database explorer //
@IBAction func LoadGameFromDatabaseExplorerButtonAction(_:AnyObject)
{
   // if haven't read in a database list //
   if !ReadInGameList
   {
      // cancel loading the game from the database explorer //
      CancelLoadGameFromDatabaseExplorerButtonAction(self)
   }
   // have already read in a game database //
   else
   {
      // close the database explorer window //
      CloseDatabaseExplorerWindow()

      // reset the game clock //
      ResetGameClock()
   }
}

// load the selected game from the game list window //
@IBAction func LoadGameFromListButtonAction(_:AnyObject)
{
   // close the game list window //
   CloseGameListWindow()

   // if the selected range is within the range of the game list data and read data //
   if Int(GameListTableView.selectedRow) >= 0 && Int(GameListTableView.selectedRow) <
      Int(DatabaseGetDataListSize(GameList)) && ReadInGameList
   {
      // get the selected game to display //
      GameData = DatabaseGetGameData(GameData, GameList, CInt(GameListData
                                     .GetGameIndex(GameListTableView.selectedRow)))

      // display the selected game from the game list //
      DisplaySelectedGameFromGameList()
   }
}

// display the selected game from the game list //
func DisplaySelectedGameFromGameList()
{
   // start a new game //
   StartNewGame(false)

   // extact and display the game information from the game data list //
   ExtractFromGameData()

   // store the autosave data //
   StoreAutosaveData("")

   // reset the game clock //
   ResetGameClock()
}

// process the fen string //
func ProcessFENString()
{
   // get the fen string from the game data //
   FENString = String(validatingUTF8:DataGetFENString(GameData))!

   // convert the fen string to an array //
   FENStringArray = FENString.components(separatedBy:" ")

   // if the fen string is of the correct size //
   if FENStringArray.count == 6
   {
      // get the starting fen board layout from the game data //
      Attributes.FENLayout = FENStringArray[0]

      // get the starting fen color to move first for the game //
      Attributes.FENColor = FENStringArray[1].lowercased()

      // get the fen castling rights for the game //
      Attributes.FENCastlingRights = FENStringArray[2]

      // get the fen enpassant square for the game //
      Attributes.FENEnpassantSquare = FENStringArray[3].lowercased()

      // get the fen start move number for the game //
      Attributes.FENStartMoveNumber = FENStringArray[5]
   }
   // the fen string was incomplete //
```

```
      // the fen string was incomplete //
      else
      {
         // set the default fen strings //
         SetDefaultFENStrings()
      }
   }

// update the game information extracted from the game data //
func UpdateGameInfo()
{
   // extract the name of the white player //
   Attributes.WhiteName = String(validatingUTF8:DataWhitePlayer(GameData))!

   // update the white name text field //
   WhiteNameTextField.stringValue = Attributes.WhiteName

   // extract the name of the black player //
   Attributes.BlackName = String(validatingUTF8:DataBlackPlayer(GameData))!

   // update the black name text field //
   BlackNameTextField.stringValue = Attributes.BlackName

   // extract the time control for the game //
   Attributes.TimeControl = String(validatingUTF8:DataGetTimeControl(GameData))!

   // update the time control text field //
   TimeControlTextField.stringValue = Attributes.TimeControl

   // extract the opening for the game //
   Attributes.Opening = String(validatingUTF8:DataGetOpening(GameData))!

   // update the opening text field //
   OpeningTextField.stringValue = Attributes.Opening

   // extract the section for the game //
   Attributes.Section = String(validatingUTF8:DataGetSection(GameData))!

   // update the section text field //
   SectionTextField.stringValue = Attributes.Section

   // extract the board number for the game //
   Attributes.BoardNumber = String(validatingUTF8:DataGetBoardNumber(GameData))!

   // update the board number text field //
   BoardNumberTextField.stringValue = Attributes.BoardNumber

   // extract the white elo for the game //
   Attributes.WhiteElo = String(validatingUTF8:DataGetWhiteElo(GameData))!

   // update the white elo text field //
   WhiteEloTextField.stringValue = Attributes.WhiteElo

   // extract the black elo for the game //
   Attributes.BlackElo = String(validatingUTF8:DataGetBlackElo(GameData))!

   // update the black elo text field //
   BlackEloTextField.stringValue = Attributes.BlackElo

   // extract the white uscf for the game //
   Attributes.WhiteUSCF = String(validatingUTF8:DataGetWhiteUSCF(GameData))!

   // update the white uscf text field //
   WhiteUSCFTextField.stringValue = Attributes.WhiteUSCF

   // extract the black uscf for the game //
   Attributes.BlackUSCF = String(validatingUTF8:DataGetBlackUSCF(GameData))!

   // update the black uscf text field //
   BlackUSCFTextField.stringValue = Attributes.BlackUSCF

   // extract the annotator for the game //
   Attributes.Annotator = String(validatingUTF8:DataGetAnnotator(GameData))!

   // update the annotator text field //
   AnnotatorTextField.stringValue = Attributes.Annotator

   // extract the eco for the game //
   Attributes.ECO = String(validatingUTF8:DataGetECO(GameData))!

   // update the eco text field //
   ECOTextField.stringValue = Attributes.ECO

   // extract the site location of the game //
   Attributes.SiteLocation = String(validatingUTF8:DataGetSite(GameData))!

   // update the site location text field //
   SiteLocationTextField.stringValue = Attributes.SiteLocation

   // extract the event title of the game //
   Attributes.EventTitle = String(validatingUTF8:DataGetEvent(GameData))!

   // update the event title text field //
   EventTitleTextField.stringValue = Attributes.EventTitle

   // extract the round number of the game //
   Attributes.RoundNumber = String(validatingUTF8:DataGetRound(GameData))!

   // update the round number text field //
   RoundNumberTextField.stringValue = Attributes.RoundNumber

   // extract the end result of the game //
   Attributes.EndResult = String(validatingUTF8:DataGetResult(GameData))!

   // if white won the game //
   if Attributes.EndResult == "1-0"
   {
      // turn on the white won button //
      WhiteWonButton.state = NSControl.StateValue(rawValue:1)

      // set that white won the game //
      WhiteWonGame = true
   }
   // if black won the game //
   else if Attributes.EndResult == "0-1"
   {
      // turn on the black won button //
      BlackWonButton.state = NSControl.StateValue(rawValue:1)

      // set that black won the game //
      BlackWonGame = true
   }
   // if the game ended in a draw //
   else if Attributes.EndResult == "1/2-1/2"
   {
      // turn on the draw button //
      DrawButton.state = NSControl.StateValue(rawValue:1)

      // set that the game was drawn //
      DrawnGame = true
   }

   // extract the date the game was played //
   Attributes.DatePlayed = String(validatingUTF8:DataGetDate(GameData))!

   // if the date format is valid //
   if TheDateFormatter.date(from:Attributes.DatePlayed) != nil
   {
      // update the date picker calendar //
      GameDate.dateValue = TheDateFormatter.date(from:Attributes.DatePlayed)!
   }
}

// save the current board layout for before go into the position editor //
func SaveBoardLayout()
{
   // save the current board layout //
   SavedLayout = Layout

   // save the fen layout //
   FENSavedLayout = Attributes.FENLayout

   // save the fen color //
   FENSavedColor = Attributes.FENColor

   // save the fen start move number //
   FENSavedStartMoveNumber = Attributes.FENStartMoveNumber

   // save the fen en passant square //
   FENSavedEnpassantSquare = Attributes.FENEnpassantSquare

   // save the fen castling rights //
   FENSavedCastlingRights = Attributes.FENCastlingRights
}

// reset the board layout to the original setup //
func ResetBoardLayout()
{
   // reset the board layout //
   Layout = SavedLayout

   // reset the fen layout //
   Attributes.FENLayout = FENSavedLayout

   // reset the fen color //
   Attributes.FENColor = FENSavedColor

   // reset the fen start move number //
   Attributes.FENStartMoveNumber = FENSavedStartMoveNumber

   // reset the fen en passant square //
   Attributes.FENEnpassantSquare = FENSavedEnpassantSquare

   // reset the fen castling rights //
   Attributes.FENCastlingRights = FENSavedCastlingRights

   // set moving piece to zero //
   MovingPiece = 0
}

// enter the database explorer //
@IBAction func DatabaseExplorerMenuItemAction(_:AnyObject)
{
   // add a timer to run the database explorer menu item action //
   RunLoop.current.add(Timer(timeInterval:0.01, target:self, selector:
                       #selector(GreenChess.TheDatabaseExplorerMenuItemAction), userInfo:nil,
                       repeats:false), forMode:RunLoop.commonModes)
}

// the database explorer menu item action //
@objc func TheDatabaseExplorerMenuItemAction()
{
   // set that currently in the database explorer //
   InDatabaseExplorer = true

   // convert the game data to a game data object //
   ConvertToGameData()

   // save the current game data object //
   DataCopy(SavedGameData, GameData)

   // turn on the database explorer menu item //
   DatabaseExplorerMenuItem.state = NSControl.StateValue(rawValue:1)

   // if already read in a game database //
   if ReadInGameList
   {
      // display the first game from the game database //
      UpdateDatabaseExplorer(self)
   }
   // did not read in a game database successfully //
   else
   {
      // there was no database loaded yet //
      MainWindow.title = "No Database Loaded"

      // update the move list //
      UpdateMoveList()
   }

   // open the database explorer window sheet //
   MainWindow.beginSheet(DatabaseExplorerWindow, completionHandler:nil)
}

// edit the current board position //
@IBAction func CurrentPositionEditorMenuItemAction(_:AnyObject)
{
   // set that editing nonempty board position //
   EditingNonemptyPosition = true

   // enter the position editor //
   PositionEditorMenuItemAction(self)
}

// create an initial board layout position for analysis //
@IBAction func PositionEditorMenuItemAction(_:AnyObject)
{
   // set that currently in the position editor //
   InPositionEditor = true

   // if not entering fen //
   if !EnteringFEN
   {
      // save the current board layout //
      SaveBoardLayout()

      // if starting with an empty position //
      if !EditingNonemptyPosition
      {
         // set the default fen strings //
         SetDefaultFENStrings()

         // assign a blank layout //
         Layout = BlankLayout
      }
      // starting from a current position //
      else
      {
         // clear the fen enpassant square //
         Attributes.FENEnpassantSquare = "-"

         // clear the fen castling rights //
         Attributes.FENCastlingRights = "-"

         // if black is to move next //
         if BoardIsBlack(Board)
         {
            // set the fen color to black //
            Attributes.FENColor = "b"
         }
         // white is to move next //
         else
         {
            // set the fen color to white //
            Attributes.FENColor = "w"
         }
      }
   }

   // initialize the castles //
   InitializeCastles()

   // clear the moving piece //
   MovingPiece = 0

   // hide the captured pieces white view //
   CapturedPiecesWhiteView.alphaValue = 0

   // hide the captured pieces black view //
   CapturedPiecesBlackView.alphaValue = 0

   // hide the pgn moves table view //
   PGNMovesTableView.enclosingScrollView?.alphaValue = 0

   // hide the game info table view //
   GameInfoTableView.enclosingScrollView?.alphaValue = 0

   // clear the tables //
   ClearTables()

   // save that not yet decided whether will save edited position or not //
   WillSaveEditedPosition = false

   // set the position cursor to a white pawn //
   SetPosCursor(1)

   // turn on the white pawn item //
   MainToolbar.selectedItemIdentifier = NSToolbarItem.Identifier(rawValue:"204")

   // set that not currently on the second position editor menu //
   SecondPositionMenu = false

   // turn on the position editor menu item //
   PositionEditorMenuItem.state = NSControl.StateValue(rawValue:1)

   // set the main window title to the position editor mode //
   MainWindow.title = "GreenChess (Position Editor Mode)"

   // update the board //
   UpdateBoardAndCapturedPieces()

   // add the position editor piece items to the toolbar //
   AddPositionEditorPieceItems()

   // update the position editor toolbar buttons //
   UpdatePositionEditorToolbarButtons()
}

// create the castling rights for the fen notation string //
func CreateCastlingRights()
{
   // initialize the fen notation string castling rights //
   Attributes.FENCastlingRights = ""

   // if white given right to kingside castle and correct king and rook positions //
   if WhiteKingsideCastle && Layout[4][0] == 6 && Layout[7][0] == 4
   {
      // append to the fen castling rights that white able to castle kingside //
      Attributes.FENCastlingRights += "K"
   }

   // if white given right to queenside castle and correct king and rook positions //
   if WhiteQueensideCastle && Layout[4][0] == 6 && Layout[0][0] == 4
   {
      // append to the fen castling rights that white able to castle queenside //
      Attributes.FENCastlingRights += "Q"
   }

   // if black given right to kingside castle and correct king and rook positions //
   if BlackKingsideCastle && Layout[4][7] == -6 && Layout[7][7] == -4
   {
      // append to the fen castling rights that black able to castle kingside //
      Attributes.FENCastlingRights += "k"
   }

   // if black given right to queenside castle and correct king and rook positions //
   if BlackQueensideCastle && Layout[4][7] == -6 && Layout[0][7] == -4
   {
      // append to the fen castling rights that black able to castle queenside //
      Attributes.FENCastlingRights += "q"
   }

   // if ended up with no fen castling rights //
   if Attributes.FENCastlingRights.isEmpty
   {
      // set that will have no possible castling rights //
      Attributes.FENCastlingRights = "-"
   }
}

// exit the position editor //
func ExitPositionEditor()
{
   // reset to the arrow cursor //
   NSCursor.arrow.set()

   // turn off the position editor menu item //
   PositionEditorMenuItem.state = NSControl.StateValue(rawValue:0)

   // if will be able to save the edited position //
   if WillSaveEditedPosition
   {
      // save the fen layout from the edited board position //
      ConvertLayoutToFEN()

      // save whether was currently analyzing the game or not //
      WasAnalyzingGame = AnalyzingGame

      // save the edited board layout //
      SaveBoardLayout()

      // start a new game //
      StartNewGame(false)

      // reset the board layout to saved layout //
      ResetBoardLayout()

      // reset the game clock //
      ResetGameClock()

      // if was currently analyzing a game //
      if WasAnalyzingGame
      {
         // set that in game analysis mode //
         AnalyzingGame = true

         // set that in game explorer //
         ExploringGame = true

         // set game explorer menu item to show exit game explorer analysis mode //
         GameExplorerMenuItem.title = "Exit Game Explorer Analysis Mode"
      }

      // store the autosave data //
      StoreAutosaveData("")
   }
   // not able to save the edited board position //
   else
   {
      // reset board to the original layout //
      ResetBoardLayout()
   }

   // set that no longer in the position editor //
   InPositionEditor = false

   // replay all the moves //
   ReplayMoves(0, End:MoveList.count, Display:false)

   // unhide the captured pieces white view //
   CapturedPiecesWhiteView.alphaValue = 1

   // unhide the captured pieces black view //
   CapturedPiecesBlackView.alphaValue = 1

   // unhide the pgn moves table view //
   PGNMovesTableView.enclosingScrollView?.alphaValue = 1

   // unhide the game info table view //
   GameInfoTableView.enclosingScrollView?.alphaValue = 1

   // set that no longer saving the position editor board layout //
   WillSaveEditedPosition = false

   // set that not editing nonempty board position anymore //
   EditingNonemptyPosition = false

   // reset entering fen to convert to false //
   EnteringFEN = false

   // if was analyzing a game //
   if AnalyzingGame
   {
      // set title to show that in the game explorer analysis mode //
      MainWindow.title = "GreenChess (Game Explorer Analysis Mode)"
   }
   // wasn't analyzing game //
   else
   {
      // reset the main window title //
      MainWindow.title = "GreenChess"
   }

   // add the default items to the toolbar //
   AddDefaultItems()
}

// the append game to existing pgn file menu item action //
@IBAction func AppendGameMenuItemAction(_:AnyObject)
{
   // turn on the append game menu item //
   AppendGameMenuItem.state = NSControl.StateValue(rawValue:1)

   // close the attached sheet if had one //
   CloseAttachedSheet()

   // create the append panel //
   AppendPanel = NSOpenPanel()

   // set append panel to close if quitting application //
   AppendPanel.preventsApplicationTerminationWhenModal = false

   // set the allowed file types that can open for the append panel //
   AppendPanel.allowedFileTypes = ["pgn"]

   // run the append panel as a sheet on the main window //
   AppendPanel.beginSheetModal(for:MainWindow, completionHandler:
                               {Result -> Void in self.PlayGameSoundThread("buttonclick")
                               if Result == NSApplication.ModalResponse.OK
                               {self.AppendGame(self.AppendPanel.url!.path)}
                               self.ReopenPreviouslyAttachedSheet()
                               self.AppendGameMenuItem.state = NSControl.StateValue(rawValue:0)})
}

// the open game menu item action //
@IBAction func OpenGameMenuItemAction(_:AnyObject)
{
   // turn on the open game menu item //
   OpenGameMenuItem.state = NSControl.StateValue(rawValue:1)

   // close the previously attached sheet if had one //
   CloseAttachedSheet()

   // create an open panel //
   OpenPanel = NSOpenPanel()

   // set the open panel to close if quitting application //
   OpenPanel.preventsApplicationTerminationWhenModal = false

   // set allowed file types to only pgn format //
   OpenPanel.allowedFileTypes = ["pgn"]

   // run the open panel as a sheet on the main window //
   OpenPanel.beginSheetModal(for:MainWindow, completionHandler:
                             {Result -> Void in self.PlayGameSoundThread("buttonclick")
                             if Result == NSApplication.ModalResponse.OK
                             {self.GameFilename = self.OpenPanel.url!.path
                             RunLoop.current.add(Timer(timeInterval:0.01, target:self,
                             selector:#selector(GreenChess.OpenGame), userInfo:nil, repeats:
                             false), forMode:RunLoop.commonModes)}
                             self.ReopenPreviouslyAttachedSheet()
                             self.OpenGameMenuItem.state = NSControl.StateValue(rawValue:0)})
}

// save the current game menu item action //
@IBAction func SaveGameMenuItemAction(_:AnyObject)
{
   // turn on the save game menu item //
   SaveGameMenuItem.state = NSControl.StateValue(rawValue:1)

   // close the previously attached sheet if had one //
   CloseAttachedSheet()

   // create a save panel //
   SavePanel = NSSavePanel()

   // set the save panel to close if quitting application //
   SavePanel.preventsApplicationTerminationWhenModal = false

   // set the allowed type of file format for a saved game //
   SavePanel.allowedFileTypes = ["pgn"]

   // set the default filename to save game as //
   SavePanel.nameFieldStringValue = "GreenChessGame"

   // run panel to save the current game in pgn format //
   SavePanel.beginSheetModal(for:MainWindow, completionHandler:
                             {Result -> Void in self.PlayGameSoundThread("buttonclick")
                             if Result == NSApplication.ModalResponse.OK
                             {self.SaveGame(self.SavePanel.url!.path)}
                             self.ReopenPreviouslyAttachedSheet()
                             self.SaveGameMenuItem.state = NSControl.StateValue(rawValue:0)})
}

// exit the diagram mode //
func ExitDiagramMode()
{
   // add the default items to the toolbar //
   AddDefaultItems()

   // loop over all the shapes //
   for Index in 0 ..< Shapes.count
   {
      // remove the shape from the super layer //
      Shapes[Index].removeFromSuperlayer()
   }

   // remove all the shapes from the shape list //
   Shapes.removeAll()

   // reset the game data //
   ResetGameData()
}

// reset the game data //
func ResetGameData()
{
   // start a new game //
   StartNewGame(false)

   // recover the saved game data //
   DataCopy(GameData, SavedGameData)

   // extract and display information from the saved game data object //
   ExtractFromGameData()
}

// enter the diagramming mode //
@IBAction func EnterDiagramMode(_:AnyObject)
{
   // set that the game explorer window is currently closed //
   GameExplorerWindowOpen = false

   // add the diagram items to the toolbar //
   AddDiagramItems()

   // set the main toolbar selected item to 305 for blue color //
   MainToolbar.selectedItemIdentifier = NSToolbarItem.Identifier(rawValue:"305")

   // set the diagram color to blue //
   DiagramColor = NSColor.blue

   // set the diagram cursor to blue highlighter //
   SetDiagramCursor("blue")

   // set the main window title to game explorer diagram mode //
   MainWindow.title = "GreenChess (Game Explorer Diagram Mode)"

   // set the game explorer menu item title to exit game explorer diagram mode //
   GameExplorerMenuItem.title = "Exit Game Explorer Diagram Mode"

   // turn off the game explorer menu item //
   GameExplorerMenuItem.state = NSControl.StateValue(rawValue:0)

   // end the game explorer window //
   MainWindow.endSheet(GameExplorerWindow)

   // close the game explorer window //
   GameExplorerWindow.close()

   // play the sound of a button click //
   PlayGameSoundThread("buttonclick")

   // set that currently diagramming //
   Diagramming = true

   // save the game subset //
   SaveGameSubset()
}

// enter the position editor directly from the game explorer //
@IBAction func ExplorerPositionEditorMenuItemAction(_:AnyObject)
{
   // enter the game explorer analysis mode //
   EnterTheAnalysisMode()

   // edit the current board position //
   CurrentPositionEditorMenuItemAction(self)
}

// enter the game analysis mode //
@IBAction func EnterAnalysisMode(_:AnyObject)
{
   // set the main window title to Game Explorer Analysis Mode //
   MainWindow.title = "GreenChess (Game Explorer Analysis Mode)"

   // enter the analysis mode //
   EnterTheAnalysisMode()
}

// enter the analysis mode //
func EnterTheAnalysisMode()
{
   // set that the game explorer window is currently closed //
   GameExplorerWindowOpen = false

   // set the game explorer menu item title to exit mode //
   GameExplorerMenuItem.title = "Exit Game Explorer Analysis Mode"

   // set the game explorer menu item state to off //
   GameExplorerMenuItem.state = NSControl.StateValue(rawValue:0)

   // end the game explorer window sheet //
   MainWindow.endSheet(GameExplorerWindow)

   // close the game explorer window sheet //
   GameExplorerWindow.close()

   // play the game sound of a button click //
   PlayGameSoundThread("buttonclick")

   // set that currently analyzing the current game position //
   AnalyzingGame = true

   // save the game subset //
   SaveGameSubset()
}

// save the game subset //
func SaveGameSubset()
{
   // convert the game to a game data object //
   ConvertToGameData()

   // save the game data //
   DataCopy(SavedGameData, GameData)

   // create subarray from the move list from zero to the current move location //
   MoveList = Array(MoveList[0 ..< Int(GameExplorerSlider.intValue)])

   // create subarray from annotation list from zero to the current move location //
   AnnotationList = Array(AnnotationList[0 ..< Int(GameExplorerSlider.intValue)])

   // set the previous move list to the move list //
   PreviousMoveList = MoveList

   // set the previous annotation list to the annotation list //
   PreviousAnnotationList = AnnotationList

   // replay the move list to the current position in the game explorer board //
   ReplayMoves(0, End:MoveList.count, Display:false)
}

// exit the game explorer //
@IBAction func ExitGameExplorer(_:AnyObject)
{
   // play the game sound of a button click //
   PlayGameSoundThread("buttonclick")

   // end the game explorer window sheet //
   MainWindow.endSheet(GameExplorerWindow)

   // close the game explorer window sheet //
   GameExplorerWindow.close()

   // set that the game explorer window is currently closed //
   GameExplorerWindowOpen = false

   // turn the game explorer menu item off //
   GameExplorerMenuItem.state = NSControl.StateValue(rawValue:0)

   // set that no longer exploring the game //
   ExploringGame = false

   // set main window title to greenchess //
   MainWindow.title = "GreenChess"

   // replay the moves to the current game position //
   ReplayMoves(0, End:MoveList.count, Display:false)
}

// close the database explorer window //
func CloseDatabaseExplorerWindow()
{
   // play the game sound of a button click //
   PlayGameSoundThread("buttonclick")

   // end the database explorer window sheet //
   MainWindow.endSheet(DatabaseExplorerWindow)

   // close the database explorer window //
   DatabaseExplorerWindow.close()

   // set the database explorer menu item to off //
   DatabaseExplorerMenuItem.state = NSControl.StateValue(rawValue:0)

   // set that no longer in the database explorer //
   InDatabaseExplorer = false

   // start a new game //
   StartNewGame(false)

   // extract and display information from the game data object //
   ExtractFromGameData()

   // store the autosave data //
   StoreAutosaveData("")
}

// extract and display information from the game data object //
func ExtractFromGameData()
{
   // process the fen string from the game data //
   ProcessFENString()

   // extract the move list from the game data //
   MoveList = String(validatingUTF8:DataGetGameDataUCI(GameData))!.components(separatedBy:" ")

   // extract the annotation list from the game data //
   AnnotationList = String(validatingUTF8:DataGetGameDataAnnotations(GameData))!
                    .components(separatedBy:"*****")

   // clear the move and annotation lists if they were empty //
   ClearEmptyMoveList()

   // set the previous move list to the move list //
   PreviousMoveList = MoveList

   // set the previous annotation list to the annotation list //
   PreviousAnnotationList = AnnotationList

   // update the game information from the game data object //
   UpdateGameInfo()

   // set the maximum value of the database game explorer slider //
   DatabaseGameExplorerSlider.maxValue = Double(MoveList.count)

   // set the current value of the database game explorer slider //
   DatabaseGameExplorerSlider.intValue = Int32(MoveList.count)

   // replay all the moves for this current game //
   ReplayMoves(0, End:MoveList.count, Display:false)

   // clear the temporary move list //
   TempMoveList.removeAll()

   // set that initially haven't found any invalid moves //
   FoundInvalidMoves = false

   // loop over the entire move list //
   for Index in 0 ..< MoveList.count
   {
      // if the move is not an invalid move //
      if MoveList[Index] != "Invalid Move"
      {
         // add the move to the temporary move list //
         TempMoveList.append(MoveList[Index])

         // set that found invalid moves //
         FoundInvalidMoves = true
      }
   }

   // copy the temporary move list back into the move list //
   MoveList = TempMoveList

   // if found invalid moves //
   if FoundInvalidMoves
   {
      // set the maximum value of the database game explorer slider //
      DatabaseGameExplorerSlider.maxValue = Double(MoveList.count)

      // set the current value of the database game explorer slider //
      DatabaseGameExplorerSlider.intValue = Int32(MoveList.count)

      // replay all the moves for this current game //
      ReplayMoves(0, End:MoveList.count, Display:false)
   }
}

// close the game list window //
func CloseGameListWindow()
{
   // play the game sound of a button click //
   PlayGameSoundThread("buttonclick")

   // end the game list window sheet //
   MainWindow.endSheet(GameListWindow)

   // close the game list window //
   GameListWindow.close()

   // set the search game list menu item to off //
   SearchGameListMenuItem.state = NSControl.StateValue(rawValue:0)

   // set that no longer searching the game list //
   SearchingGameList = false

   // reset the main window title //
   MainWindow.title = "GreenChess"
}

// exit the gallery grid //
func ExitGalleryGrid()
{
   // play the sound of a button click //
   PlayGameSoundThread("buttonclick")

   // remove the tracking rectangle from the gallery grid view //
   GalleryGridView.removeTrackingRect(GalleryGridView.TrackingRectangle)

   // stop the gallery grid view animation //
   GalleryGridView.StopAnimation()

   // end the gallery grid window //
   MainWindow.endSheet(GalleryGridWindow)

   // close the gallery grid window //
   GalleryGridWindow.close()

   // reset the previous window title //
   MainWindow.title = PreviousMainWindowTitle

   // erase all the gallery grid view shapes //
   GalleryGridView.EraseShapes()
}

// exit the game grid //
@IBAction func ExitGameGrid(_:AnyObject)
{
   // exit the gallery grid //
   ExitGalleryGrid()

   // turn off the game grid menu item //
   GameGridMenuItem.state = NSControl.StateValue(rawValue:0)

   // set that no longer in the game grid //
   InGameGrid = false

   // replay all the moves //
   ReplayMoves(0, End:MoveList.count, Display:false)
}

// exit the game gallery //
@IBAction func ExitGameGallery(_:AnyObject)
{
   // exit the gallery grid //
   ExitGalleryGrid()

   // turn off the game gallery menu item //
   GameGalleryMenuItem.state = NSControl.StateValue(rawValue:0)

   // set that no longer in the game gallery //
   InGameGallery = false

   // replay all the moves //
   ReplayMoves(0, End:MoveList.count, Display:false)
}

// exit the preferences window //
@IBAction func ExitPreferencesWindow(_:AnyObject)
{
   // play the game sound of a button click //
   PlayGameSoundThread("buttonclick")

   // turn off the preferences menu item //
   PreferencesMenuItem.state = NSControl.StateValue(rawValue:0)

   // end the preferences sheet //
   MainWindow.endSheet(PreferencesWindow)

   // close the preferences window //
   PreferencesWindow.close()

   // save the board labels font type to preferences //
   AppPreferences.set(BoardLabelsFontType, forKey:"BoardLabelsFontType")

   // save the board labels bold setting to preferences //
   AppPreferences.set(BoardLabelsBold, forKey:"BoardLabelsBold")

   // save the board labels font sizes setting to preferences //
   AppPreferences.set(BoardLabelsSizes, forKey:"BoardLabelsSizes")

   // save the move list font type to preferences //
   AppPreferences.set(MoveListFontType, forKey:"MoveListFontType")

   // save the move list bold setting to preferences //
   AppPreferences.set(MoveListBold, forKey:"MoveListBold")

   // save the move list font sizes setting to preferences //
   AppPreferences.set(MoveListSizes, forKey:"MoveListSizes")

   // save the game info font type to preferences //
   AppPreferences.set(GameInfoFontType, forKey:"GameInfoFontType")

   // save the game info bold setting to preferences //
   AppPreferences.set(GameInfoBold, forKey:"GameInfoBold")

   // save the game info font sizes setting to preferences //
   AppPreferences.set(GameInfoSizes, forKey:"GameInfoSizes")
```

```
      // save the game list font type to preferences //
      AppPreferences.set(GameListFontType, forKey:"GameListFontType")

      // save the game list bold setting to preferences //
      AppPreferences.set(GameListBold, forKey:"GameListBold")

      // save the game list font sizes setting to preferences //
      AppPreferences.set(GameListSizes, forKey:"GameListSizes")

      // save the chosen label color //
      AppPreferences.set(LabelColor, forKey:"LabelColor")

      // save the chosen info color //
      AppPreferences.set(InfoColor, forKey:"InfoColor")

      // save the chosen move list color //
      AppPreferences.set(MoveListColor, forKey:"MoveListColor")

      // save the chosen game list color //
      AppPreferences.set(GameListColor, forKey:"GameListColor")

      // save the chosen rainbow background color //
      AppPreferences.set(RainbowBackgroundColor, forKey:"RainbowBackgroundColor")

      // save the chosen blue background color //
      AppPreferences.set(BlueBackgroundColor, forKey:"BlueBackgroundColor")

      // save the chosen brown background color //
      AppPreferences.set(BrownBackgroundColor, forKey:"BrownBackgroundColor")

      // save the chosen green background color //
      AppPreferences.set(GreenBackgroundColor, forKey:"GreenBackgroundColor")

      // save the chosen teal background color //
      AppPreferences.set(TealBackgroundColor, forKey:"TealBackgroundColor")

      // save the chosen game clock border color //
      AppPreferences.set(GameClockColor, forKey:"GameClockColor")

      // save the game clock background color //
      AppPreferences.set(GameClockBackgroundColor, forKey:"GameClockBackgroundColor")

      // save the clock background visibility //
      AppPreferences.set(ClockBackgroundVisibility, forKey:"ClockBackgroundVisibility")

      // save the chosen yellow background color //
      AppPreferences.set(YellowBackgroundColor, forKey:"YellowBackgroundColor")

      // save the chosen gray background color //
      AppPreferences.set(GrayBackgroundColor, forKey:"GrayBackgroundColor")

      // save the chosen background color set //
      AppPreferences.set(BackgroundColorSet, forKey:"BackgroundColorSet")

      // save the user preference for the board type //
      AppPreferences.set(BoardType, forKey:"BoardType")

      // save the user preference for the piece set //
      AppPreferences.set(ChessPieceSetType, forKey:"ChessPieceSetType")

      // save the user preference for the border texture //
      AppPreferences.set(BorderTexture, forKey:"BorderTexture")

      // save the user preference for the border visibility //
      AppPreferences.set(BorderVisibility, forKey:"BorderVisibility")

      // save the user preference for the board tint visibility //
      AppPreferences.set(BoardTintVisibility, forKey:"BoardTintVisibility")

      // save the window texture opacity //
      AppPreferences.set(WindowTextureOpacity, forKey:"WindowTextureOpacity")

      // save the user preference for the board roughness //
      AppPreferences.set(BoardRoughness, forKey:"BoardRoughness")

      // save the user preference for the square border //
      AppPreferences.set(SquareBorder, forKey:"SquareBorder")

      // save the user preference for the board tint color //
      AppPreferences.set(BoardTintColor, forKey:"BoardTintColor")

      // save the user preference for the highlight last move made option //
      AppPreferences.set(HighlightLastMoveMade, forKey:"HighlightLastMoveMade")

      // save the user preference for the highlight threatened pieces option //
      AppPreferences.set(HighlightThreatenedPieces, forKey:"HighlightThreatenedPieces")

      // save the user preference for the highlight captured pieces option //
      AppPreferences.set(HighlightCapturedPieces, forKey:"HighlightCapturedPieces")

      // save the user preference for the multithread uci engine option //
      AppPreferences.set(MultithreadUCIEngine, forKey:"MultithreadUCIEngine")

      // save the autosave game preference //
      AppPreferences.set(AutosaveGame, forKey:"AutosaveGame")

      // update the user preference for the show stockfish analysis option //
      AppPreferences.set(ShowStockfishAnalysis, forKey:"ShowStockfishAnalysis")

      // save the user preference for the board animation rate //
      AppPreferences.set(AnimationRate, forKey:"AnimationRate")

      // update the user preference for the autoload stockfish option //
      AppPreferences.set(AutoloadStockfish, forKey:"AutoloadStockfish")

      // save the user preference for the gallery and grid animation rate //
      AppPreferences.set(GalleryGridAnimationRate, forKey:"GalleryGridAnimationRate")

      // save the user preference for the uci engine difficulty //
      AppPreferences.set(UCIEngineDifficulty, forKey:"UCIEngineDifficulty")

      // save the user preference for the uci engine hash //
      AppPreferences.set(UCIEngineHash, forKey:"UCIEngineHash")

      // save the user preference for the uci engine riskiness //
      AppPreferences.set(UCIEngineRiskiness, forKey:"UCIEngineRiskiness")

      // set the user preference for the search time //
      AppPreferences.set(SearchTime, forKey:"SearchTime")

      // save the user preference for the use eco book option //
      AppPreferences.set(UseEcoBook, forKey:"UseEcoBook")

      // get the current preferences selection //
      PreferencesSelection = Int(PreferencesToolbar.selectedItemIdentifier!.rawValue)!

      // save the user preference for the preferences selection //
      AppPreferences.set(PreferencesSelection, forKey:"PreferencesSelection")

      // update the user preference for the sound off key //
      AppPreferences.set(SoundOff, forKey:"SoundOff")

      // update the user preference for move by selecting squares //
      AppPreferences.set(MoveBySelectingSquares, forKey:"MoveBySelectingSquares")

      // set that no longer entering in preferences //
      EnteringPreferences = false

      // replay all the moves //
      ReplayMoves(0, End:MoveList.count, Display:false)
   }

   // get the string from the text field //
   func StringFromTextField(_ TextField:String) -> String
   {
      // return string from the text field with white space, newlines, and quotes trimmed off //
      return TextField.replacingOccurrences(of:"\"", with:"").replacingOccurrences(of:"\n",
                        with:" ").trimmingCharacters(in:CharacterSet.whitespacesAndNewlines)
   }

   // exit the about window //
   @IBAction func ExitAboutWindow(_:AnyObject)
   {
      // play the game sound of a button click //
      PlayGameSoundThread("buttonclick")

      // turn off the about menu item //
      AboutMenuItem.state = NSControl.StateValue(rawValue:0)

      // end the about window sheet //
      MainWindow.endSheet(AboutWindow)

      // close the about window //
      AboutWindow.close()

      // reopen the previously attached sheet //
      ReopenPreviouslyAttachedSheet()
   }

   // exit the enter game info window //
   @IBAction func ExitEnterGameInfoWindow(_:AnyObject)
   {
      // play the game sound of a button click //
      PlayGameSoundThread("buttonclick")

      // turn the enter game info menu item off //
      EnterGameInfoMenuItem.state = NSControl.StateValue(rawValue:0)

      // close the enter game info sheet in the application //
      MainWindow.endSheet(EnterGameInfoWindow)

      // close the enter game info window //
      EnterGameInfoWindow.close()

      // get the name of the person playing as white if was entered //
      Attributes.WhiteName = StringFromTextField(WhiteNameTextField.stringValue)

      // get the name of the person playing as black if was entered //
      Attributes.BlackName = StringFromTextField(BlackNameTextField.stringValue)

      // get the site location of the game if was entered //
      Attributes.SiteLocation = StringFromTextField(SiteLocationTextField.stringValue)

      // get the title of the tournament event if was entered //
      Attributes.EventTitle = StringFromTextField(EventTitleTextField.stringValue)

      // get the round number of the game if was entered //
      Attributes.RoundNumber = StringFromTextField(RoundNumberTextField.stringValue)

      // get the time control of the game if was entered //
      Attributes.TimeControl = StringFromTextField(TimeControlTextField.stringValue)

      // get the opening of the game if was entered //
      Attributes.Opening = StringFromTextField(OpeningTextField.stringValue)

      // get the section of the game if was entered //
      Attributes.Section = StringFromTextField(SectionTextField.stringValue)

      // get the board number of the game if was entered //
      Attributes.BoardNumber = StringFromTextField(BoardNumberTextField.stringValue)

      // get the white elo of the game if was entered //
      Attributes.WhiteElo = StringFromTextField(WhiteEloTextField.stringValue)

      // get the black elo of the game if was entered //
      Attributes.BlackElo = StringFromTextField(BlackEloTextField.stringValue)

      // get the white uscf of the game if was entered //
      Attributes.WhiteUSCF = StringFromTextField(WhiteUSCFTextField.stringValue)

      // get the black uscf of the game if was entered //
      Attributes.BlackUSCF = StringFromTextField(BlackUSCFTextField.stringValue)

      // get the annotator of the game if was entered //
      Attributes.Annotator = StringFromTextField(AnnotatorTextField.stringValue)

      // get the eco of the game if was entered //
      Attributes.ECO = StringFromTextField(ECOTextField.stringValue)

      // if white won the game //
      if WhiteWonGame
      {
         // set that white won the game //
         Attributes.EndResult = "1-0"
      }
      // if black won the game //
      else if BlackWonGame
      {
         // set that black won the game //
         Attributes.EndResult = "0-1"
      }
      // if the game was drawn //
      else if DrawnGame
      {
         // set that game ended in a draw //
         Attributes.EndResult = "1/2-1/2"
      }
      // the game has not ended //
      else
      {
         // set the game result to none //
         Attributes.EndResult = ""
      }

      // get the date that the game was played on //
      Attributes.DateGamePlayed = TheDateFormatter.string(from:GameDate.dateValue)

      // print the current game information //
      PrintGameInfo()

      // set that no longer entering game info //
      EnteringGameInfo = false

      // store the autosave data //
      StoreAutosaveData("")
   }

   // update the game gallery or grid view //
   @IBAction func UpdateGalleryGridView(_:AnyObject)
   {
      // update the game gallery or grid view //
      GalleryGridView.UpdateGalleryGridView(GalleryGridSlider.doubleValue)
   }

   // the game grid menu item action //
   @IBAction func GameGridMenuItemAction(_:AnyObject)
   {
      // set that in the game grid //
      InGameGrid = true

      // save the previous main window title //
      PreviousMainWindowTitle = MainWindow.title

      // turn on the game grid menu item //
      GameGridMenuItem.state = NSControl.StateValue(rawValue:1)

      // set the game grid slider minimum value //
      GameGridSlider.minValue = 0

      // set the game grid slider maximum value //
      GameGridSlider.maxValue = 1

      // set the game grid slider current value to zero //
      GameGridSlider.doubleValue = 0

      // set the active gallery grid view //
      GalleryGridView = GameGridView

      // set the active gallery grid review button //
      GalleryGridReview = GameGridReviewButton

      // set the active gallery grid window //
      GalleryGridWindow = GameGridWindow

      // set the active gallery grid toolbar //
      GalleryGridToolbar = GridToolbar

      // set the active gallery grid slider //
      GalleryGridSlider = GameGridSlider

      // set the current gallery grid title //
      GalleryGridTitle = "Gallery"

      // deselect the gallery grid diagram color button //
      GalleryGridToolbar.selectedItemIdentifier = nil

      // create the gallery grid thread //
      GalleryGridThread = Thread(target:self, selector:#selector(GreenChess.GetGalleryGridImages),
                                 object:nil)

      // set the gallery grid thread to the highest priority //
      GalleryGridThread.threadPriority = 1

      // start the gallery grid thread //
      GalleryGridThread.start()
   }

   // the game gallery menu item action //
   @IBAction func GameGalleryMenuItemAction(_:AnyObject)
   {
      // set that currently in the game gallery //
      InGameGallery = true

      // save the main window title //
      PreviousMainWindowTitle = MainWindow.title

      // turn on the game gallery menu item //
      GameGalleryMenuItem.state = NSControl.StateValue(rawValue:1)

      // set the game gallery slider minimum value to zero //
      GameGallerySlider.minValue = 0

      // set the game gallery slider maximum value to one //
      GameGallerySlider.maxValue = 1

      // set the game gallery slider initial value to zero //
      GameGallerySlider.doubleValue = 0

      // set the active gallery grid view //
      GalleryGridView = GameGalleryView

      // set the active gallery grid review button //
      GalleryGridReview = GameGalleryReviewButton

      // set the active gallery grid window //
      GalleryGridWindow = GameGalleryWindow

      // set the active gallery grid toolbar //
      GalleryGridToolbar = GalleryToolbar

      // set the active gallery grid slider //
      GalleryGridSlider = GameGallerySlider

      // set the gallery grid title //
      GalleryGridTitle = "Gallery"

      // deselect the gallery grid diagram color button //
      GalleryGridToolbar.selectedItemIdentifier = nil

      // create the gallery grid thread //
      GalleryGridThread = Thread(target:self, selector:#selector(GreenChess.GetGalleryGridImages),
                                 object:nil)

      // set the gallery grid thread to the highest priority //
      GalleryGridThread.threadPriority = 1

      // start the gallery grid thread //
      GalleryGridThread.start()
   }

   // set the board label fonts //
   @IBAction func SetBoardLabelsFont(_:AnyObject)
   {
      // get the board labels font type from the board labels font button //
      BoardLabelsFontType = BoardLabelsFontButton.indexOfSelectedItem

      // update the board labels layer //
      UpdateBoardLabelsLayer()

      // recomposite the board for the updated board labels layer //
      RecompositeBoard(3, Span:1, Width:660)
   }

   // set the move list font //
   @IBAction func SetMoveListFont(_:AnyObject)
   {
      // get the move list font type from the move list font button //
      MoveListFontType = MoveListFontButton.indexOfSelectedItem

      // update the move list //
      UpdateMoveList()
   }

   // set the game info font //
   @IBAction func SetGameInfoFont(_:AnyObject)
   {
      // get the game info font type from the game info font button //
      GameInfoFontType = GameInfoFontButton.indexOfSelectedItem

      // update the game info list //
      UpdateGameInfoList()
   }

   // set the chess piece set type //
   @IBAction func SetPieceSetType(_:AnyObject)
   {
      // get the chess piece set type from the piece set button //
      ChessPieceSetType = PieceSetButton.indexOfSelectedItem

      // update the board and captured pieces //
      UpdateBoardAndCapturedPieces()

      // update the position editor toolbar buttons //
      UpdatePositionEditorToolbarButtons()
   }

   // update the board type //
   @IBAction func UpdateBoardTypeLayer(_:AnyObject)
   {
      // get the board type from the board textures button //
      BoardType = BoardTexturesButton.indexOfSelectedItem

      // update the chess board layer //
      BoardLayers[4] = ChessBoardImages[BoardType]

      // recomposite the board for the updated board type layer //
      RecompositeBoard(4, Span:1, Width:660)
   }

   // set the game list font //
   @IBAction func SetGameListFont(_:AnyObject)
   {
      // get the game list font type from the game list button //
      GameListFontType = GameListFontButton.indexOfSelectedItem
   }

   // set the move list bold type //
   @IBAction func SetMoveListBold(_:AnyObject)
   {
      // get the move list bold type from the move list bold button //
      MoveListBold = IntToBool(MoveListBoldButton.indexOfSelectedItem)

      // update the move list //
      UpdateMoveList()
   }

   // set the game info bold type //
   @IBAction func SetGameInfoBold(_:AnyObject)
   {
      // get the game info bold type from the game info bold button //
      GameInfoBold = IntToBool(GameInfoBoldButton.indexOfSelectedItem)

      // update the game info list //
      UpdateGameInfoList()
   }

   // set the game list bold type //
   @IBAction func SetGameListBold(_:AnyObject)
   {
      // get the game list bold type from the game list bold button //
      GameListBold = IntToBool(GameListBoldButton.indexOfSelectedItem)
   }

   // set the board labels bold setting //
   @IBAction func SetBoardLabelsBold(_:AnyObject)
   {
      // get the board labels bold setting from the board labels bold button //
      BoardLabelsBold = IntToBool(BoardLabelsBoldButton.indexOfSelectedItem)

      // update the board labels layer //
      UpdateBoardLabelsLayer()

      // recomposite the board for the updated board labels layer //
      RecompositeBoard(3, Span:1, Width:660)
   }

   // set the move list font sizes setting //
   @IBAction func SetMoveListSizes(_:AnyObject)
   {
      // get the move list font sizes setting from the move list font sizes button //
      MoveListSizes = MoveListSizesButton.indexOfSelectedItem

      // update the move list //
      UpdateMoveList()
   }

   // set the game info sizes setting //
   @IBAction func SetGameInfoSizes(_:AnyObject)
   {
      // get the game info sizes setting from the game info fonts sizes button //
      GameInfoSizes = GameInfoSizesButton.indexOfSelectedItem

      // update the game info list //
      UpdateGameInfoList()
   }

   // set the game list sizes setting //
   @IBAction func SetGameListSizes(_:AnyObject)
   {
      // get the game list sizes setting from the game info fonts sizes button //
      GameListSizes = GameListSizesButton.indexOfSelectedItem
   }

   // set the board labels font sizes setting //
   @IBAction func SetBoardLabelsSizes(_:AnyObject)
   {
      // get board labels font sizes setting from the board labels font sizes button //
      BoardLabelsSizes = BoardLabelsSizesButton.indexOfSelectedItem

      // update the board labels layer //
      UpdateBoardLabelsLayer()

      // recomposite the board for the updated board labels layer //
      RecompositeBoard(3, Span:1, Width:660)
   }

   // set the current background color //
   func SetBackgroundColor(_ ColorSet:Int)
   {
      // set the current background color set //
      BackgroundColorSet = ColorSet

      // set the rainbow background view border to false //
      RainbowBackgroundView.Bordered = false

      // set the blue background view border to false //
      BlueBackgroundView.Bordered = false

      // set the brown background view border to false //
      BrownBackgroundView.Bordered = false

      // set the green background view border to false //
      GreenBackgroundView.Bordered = false

      // set the teal background view border to false //
      TealBackgroundView.Bordered = false

      // set the yellow background view border to false //
      YellowBackgroundView.Bordered = false

      // set the gray background view border to false //
      GrayBackgroundView.Bordered = false

      // if color set is for rainbow colors //
      if BackgroundColorSet == 1
      {
         // get the current rainbow background color //
         RainbowBackgroundColor = RainbowBackgroundView.CurrentColor

         // set the background color //
         SetWindowBackgroundColor(RainbowBackgroundColors[RainbowBackgroundColor])

         // set that the rainbow background color view is bordered //
         RainbowBackgroundView.Bordered = true
      }
      // if color set is for blue colors //
      else if BackgroundColorSet == 2
      {
         // get the current blue background color //
         BlueBackgroundColor = BlueBackgroundView.CurrentColor

         // set the background color //
         SetWindowBackgroundColor(BlueBackgroundColors[BlueBackgroundColor])

         // set that the blue background color view is bordered //
         BlueBackgroundView.Bordered = true
      }
      // if color set is for brown colors //
      else if BackgroundColorSet == 3
      {
         // get the current brown background color //
         BrownBackgroundColor = BrownBackgroundView.CurrentColor

         // set the background color //
         SetWindowBackgroundColor(BrownBackgroundColors[BrownBackgroundColor])

         // set that the brown background color view is bordered //
         BrownBackgroundView.Bordered = true
      }
      // if color set is for green colors //
      else if BackgroundColorSet == 4
      {
         // get the current green background color //
         GreenBackgroundColor = GreenBackgroundView.CurrentColor

         // set the background color //
         SetWindowBackgroundColor(GreenBackgroundColors[GreenBackgroundColor])

         // set that the green background color view is bordered //
         GreenBackgroundView.Bordered = true
      }
      // if color set is for teal colors //
      else if BackgroundColorSet == 5
      {
         // get the current teal background color //
         TealBackgroundColor = TealBackgroundView.CurrentColor

         // set the background color //
         SetWindowBackgroundColor(TealBackgroundColors[TealBackgroundColor])

         // set that the teal background color view is bordered //
         TealBackgroundView.Bordered = true
      }
      // if color set is for yellow colors //
      else if BackgroundColorSet == 6
      {
         // get the current yellow background color //
         YellowBackgroundColor = YellowBackgroundView.CurrentColor

         // set the background color //
         SetWindowBackgroundColor(YellowBackgroundColors[YellowBackgroundColor])

         // set that the yellow background color view is bordered //
         YellowBackgroundView.Bordered = true
      }
      // if color set is for gray colors //
      else if BackgroundColorSet == 7
      {
         // get the current gray background color //
         GrayBackgroundColor = GrayBackgroundView.CurrentColor

         // set the background color //
         SetWindowBackgroundColor(GrayBackgroundColors[GrayBackgroundColor])

         // set that the gray background color view is bordered //
         GrayBackgroundView.Bordered = true
      }

      // refresh the rainbow background color view //
      RainbowBackgroundView.needsDisplay = true

      // refresh the blue background color view //
      BlueBackgroundView.needsDisplay = true

      // refresh the brown background color view //
      BrownBackgroundView.needsDisplay = true

      // refresh the green background color view //
      GreenBackgroundView.needsDisplay = true

      // refresh the teal background color view //
      TealBackgroundView.needsDisplay = true

      // refresh the yellow background color view //
      YellowBackgroundView.needsDisplay = true

      // refresh the gray background color view //
      GrayBackgroundView.needsDisplay = true
   }

   // set the board labels font color //
   func SetBoardLabelsFontColor()
   {
      // get the label color from the current color //
      LabelColor = BoardLabelsColorView.CurrentColor

      // refresh the board labels color view //
      BoardLabelsColorView.needsDisplay = true

      // update the board labels layer //
      UpdateBoardLabelsLayer()

      // update the board composite for the updated board labels layer //
      RecompositeBoard(3, Span:1, Width:660)
   }

   // set the game info font color //
   func SetInfoFontColor()
   {
      // get the game info color from the current color //
      InfoColor = GameInfoColorView.CurrentColor

      // refresh the game info color view //
      GameInfoColorView.needsDisplay = true

      // update the game info text //
      UpdateGameInfoList()
   }

   // set the move list font color //
   func SetMoveListFontColor()
   {
      // get the move list color from the current color //
      MoveListColor = MoveListColorView.CurrentColor

      // refresh the move list color view //
      MoveListColorView.needsDisplay = true

      // update the move list //
      UpdateMoveList()
   }

   // set the game clock background color //
   func SetGameClockBackgroundColor()
   {
      // set the game clock background color from the clock background color view //
      GameClockBackgroundColor = ClockBackgroundColorView.CurrentColor

      // update the clock background color view //
      ClockBackgroundColorView.needsDisplay = true

      // update the game clock background color //
      GameClock.UpdateClockBackgroundColor(GameClockColors[GameClockBackgroundColor])

      // update the game clock preview background color //
      GameClockPreview.UpdateClockBackgroundColor(GameClockColors[GameClockBackgroundColor])

      // update the game clock preview //
      GameClockPreview.UpdateClock(0, ElapsedBlackTime:0)
   }

   // set the game clock color //
   func SetGameClockColor()
   {
      // set the game clock color from the clock color view //
      GameClockColor = ClockColorView.CurrentColor

      // update the clock color view //
      ClockColorView.needsDisplay = true

      // update the game clock color //
      GameClock.UpdateClockColor(GameClockColors[GameClockColor])

      // update the game clock border preview color //
      GameClockPreview.UpdateClockColor(GameClockColors[GameClockColor])

      // update the game clock preview //
      GameClockPreview.UpdateClock(0, ElapsedBlackTime:0)
   }

   // set the game list font color //
   func SetGameListFontColor()
   {
      // get game list color from the current color //
      GameListColor = GameListColorView.CurrentColor

      // refresh the game list color view //
      GameListColorView.needsDisplay = true
   }

   // update the board preferences to the given state //
   func BoardPreferencesToState(_ State:Bool)
   {
      // update the board textures field state //
      BoardTexturesField.isHidden = State

      // update the board textures button state //
      BoardTexturesButton.isHidden = State

      // update the piece set field state //
      PieceSetField.isHidden = State

      // update the piece set button state //
      PieceSetButton.isHidden = State

      // update the border visibility slider state //
      BorderVisibilitySlider.isHidden = State

      // update the board tint visibility slider state //
      BoardTintVisibilitySlider.isHidden = State

      // update the board roughness slider state //
      BoardRoughnessSlider.isHidden = State

      // update the square border slider state //
      SquareBorderSlider.isHidden = State

      // update the board tint color slider state //
      BoardTintColorSlider.isHidden = State

      // update the border visibility view state //
      BorderVisibilityView.isHidden = State

      // update the board tint visibility view state //
      BoardTintVisibilityView.isHidden = State

      // update the board roughness view state //
      BoardRoughnessView.isHidden = State

      // update the square border view state //
      SquareBorderView.isHidden = State

      // update the board tint color view state //
      BoardTintColorView.isHidden = State

      // update the border texture slider state //
      BorderTextureSlider.isHidden = State

      // update the border texture view state //
      BorderTextureView.isHidden = State

      // update the board preview image state //
      BoardPreview.isHidden = State

      // udate the autosave game button state //
      AutosaveGameButton.isHidden = State

      // update the game sound button state //
      GameSoundsButton.isHidden = State

      // update the move by selecting squares button state //
      MoveBySelectingSquaresButton.isHidden = State

      // update the opening book button state //
      EcoBookButton.isHidden = State

      // update the highlight last move made button state //
      HighlightLastMoveMadeButton.isHidden = State

      // update the highlight captured pieces button state //
      HighlightCapturedPiecesButton.isHidden = State

      // update the highlight threatened pieces button state //
      HighlightThreatenedPiecesButton.isHidden = State
   }

   // set the background preferences to the given state //
   func BackgroundPreferencesToState(_ State:Bool)
   {
      // update the window texture opacity slider state //
      WindowTextureOpacitySlider.isHidden = State

      // update the window texture opacity view state //
      WindowTextureOpacityView.isHidden = State

      // update the window color preview state //
      WindowColorPreview.isHidden = State

      // update the rainbow background text field state //
      RainbowBackgroundField.isHidden = State

      // update the rainbow background view state //
      RainbowBackgroundView.isHidden = State

      // update the blue background text field state //
      BlueBackgroundField.isHidden = State

      // update the blue background view state //
      BlueBackgroundView.isHidden = State

      // update the brown background text field state //
      BrownBackgroundField.isHidden = State

      // update the brown background view state //
      BrownBackgroundView.isHidden = State

      // update the green background text field state //
      GreenBackgroundField.isHidden = State

      // update the green background view state //
      GreenBackgroundView.isHidden = State

      // update the teal background text field state //
      TealBackgroundField.isHidden = State

      // update the teal background view state //
      TealBackgroundView.isHidden = State

      // update the yellow background text field state //
      YellowBackgroundField.isHidden = State

      // update the yellow background view state //
      YellowBackgroundView.isHidden = State

      // update the gray background text field state //
      GrayBackgroundField.isHidden = State

      // update the gray background view state //
      GrayBackgroundView.isHidden = State
   }

   // update the font preference states //
   func FontPreferencesToState(_ State:Bool)
   {
      // update the board labels font button state //
      BoardLabelsFontButton.isHidden = State

      // update the board labels bold button state //
      BoardLabelsBoldButton.isHidden = State

      // update the board labels font sizes button state //
      BoardLabelsSizesButton.isHidden = State

      // update the move list font button state //
      MoveListFontButton.isHidden = State

      // update the move list bold button state //
      MoveListBoldButton.isHidden = State

      // update the move list sizes button state //
      MoveListSizesButton.isHidden = State

      // update the game info font button state //
      GameInfoFontButton.isHidden = State

      // update the game info bold button state //
      GameInfoBoldButton.isHidden = State

      // update the game info sizes button state //
      GameInfoSizesButton.isHidden = State

      // update the game list font button state //
      GameListFontButton.isHidden = State

      // update the game list bold button state //
      GameListBoldButton.isHidden = State

      // update the game list sizes button state //
      GameListSizesButton.isHidden = State

      // update the board labels field state //
      BoardLabelsField.isHidden = State

      // update the move list field state //
      MoveListField.isHidden = State

      // update the game info list field state //
      GameInfoField.isHidden = State

      // update the game list field state //
      GameListField.isHidden = State

      // update the board labels color view state //
      BoardLabelsColorView.isHidden = State

      // update the move list color view state //
      MoveListColorView.isHidden = State

      // update the game info color view state //
      GameInfoColorView.isHidden = State

      // update the game list color view state //
      GameListColorView.isHidden = State
   }

   // update the stockfish preference states //
   func StockfishPreferencesToState(_ State:Bool)
   {
      // update the show the stockfish analysis button state //
      ShowStockfishAnalysisButton.isHidden = State

      // update the autoload stockfish button state //
      AutoloadStockfishButton.isHidden = State

      // update the multithread stockfish button state //
      MultithreadUCIEngineButton.isHidden = State

      // update the uci engine difficulty slider state //
      UCIEngineDifficultySlider.isHidden = State
```

```
   // update the uci engine difficulty view state //
   UCIEngineDifficultyView.isHidden = State

   // update the uci engine hash slider state //
   UCIEngineHashSlider.isHidden = State

   // update the uci engine hash view state //
   UCIEngineHashView.isHidden = State

   // update the uci engine riskiness slider state //
   UCIEngineRiskinessSlider.isHidden = State

   // update the uci engine riskiness view state //
   UCIEngineRiskinessView.isHidden = State
}

// set the board preferences layout //
func BoardPreferencesLayout()
{
   // update the board textures field frame //
   BoardTexturesField.frame = NSMakeRect(20, 585, 200, 20)

   // update the board textures button frame //
   BoardTexturesButton.frame = NSMakeRect(45, 562, 150, 25)

   // update the piece set field frame //
   PieceSetField.frame = NSMakeRect(241, 585, 200, 20)

   // update the piece set button frame //
   PieceSetButton.frame = NSMakeRect(266, 562, 150, 25)

   // update the board preview frame //
   BoardPreview.frame = NSMakeRect(30, 160, 400, 400)

   // update the autosave game button frame //
   AutosaveGameButton.frame = NSMakeRect(445, 525, 190, 20)

   // update the game sounds button frame //
   GameSoundsButton.frame = NSMakeRect(445, 475, 190, 20)

   // update the opening book button frame //
   EcoBookButton.frame = NSMakeRect(445, 425, 190, 20)

   // update the highlight last move made button frame //
   HighlightLastMoveMadeButton.frame = NSMakeRect(445, 375, 190, 20)

   // update the highlight captured pieces button frame //
   HighlightCapturedPiecesButton.frame = NSMakeRect(445, 325, 190, 20)

   // update the highlight threatened pieces button frame //
   HighlightThreatenedPiecesButton.frame = NSMakeRect(445, 275, 190, 20)

   // update the move by selecting squares button frame //
   MoveBySelectingSquaresButton.frame = NSMakeRect(445, 225, 190, 20)

   // update the border texture view frame //
   BorderTextureView.frame = NSMakeRect(30, 130, 210, 20)

   // update the border texture slider frame //
   BorderTextureSlider.frame = NSMakeRect(240, 134, 390, 13)

   // update the border visibility view frame //
   BorderVisibilityView.frame = NSMakeRect(30, 107, 210, 20)

   // update the border visibility slider frame //
   BorderVisibilitySlider.frame = NSMakeRect(240, 111, 390, 13)

   // update the square border view frame //
   SquareBorderView.frame = NSMakeRect(30, 84, 210, 20)

   // update the square border slider frame //
   SquareBorderSlider.frame = NSMakeRect(240, 88, 390, 13)

   // update the board roughness view frame //
   BoardRoughnessView.frame = NSMakeRect(30, 61, 210, 20)

   // update the board roughness slider frame //
   BoardRoughnessSlider.frame = NSMakeRect(240, 65, 390, 13)

   // update the board tint color view frame //
   BoardTintColorView.frame = NSMakeRect(30, 38, 210, 20)

   // update the board tint color slider frame //
   BoardTintColorSlider.frame = NSMakeRect(240, 42, 390, 13)

   // update the board tint visibility view frame //
   BoardTintVisibilityView.frame = NSMakeRect(30, 15, 210, 20)

   // update the board tint visibility slider frame //
   BoardTintVisibilitySlider.frame = NSMakeRect(240, 19, 390, 13)
}

// update the background preferences layout //
func BackgroundPreferencesLayout()
{
   // update the window color preview frame //
   WindowColorPreview.frame = NSMakeRect(30, 576, 40, 40)

   // update the window texture opacity slider frame //
   WindowTextureOpacitySlider.frame = NSMakeRect(30, 555, 600, 13)

   // update the window texture opacity view frame //
   WindowTextureOpacityView.frame = NSMakeRect(30, 537, 600, 20)

   // update the rainbow background text field frame //
   RainbowBackgroundField.frame = NSMakeRect(230, 510, 200, 20)

   // update the rainbow background view frame //
   RainbowBackgroundView.frame = NSMakeRect(30, 470, 600, 40)

   // update the blue background text field frame //
   BlueBackgroundField.frame = NSMakeRect(230, 435, 200, 20)

   // update the blue background view frame //
   BlueBackgroundView.frame = NSMakeRect(30, 395, 600, 40)

   // update the brown background text field frame //
   BrownBackgroundField.frame = NSMakeRect(230, 360, 200, 20)

   // update the brown background view frame //
   BrownBackgroundView.frame = NSMakeRect(30, 320, 600, 40)

   // update the green background text field frame //
   GreenBackgroundField.frame = NSMakeRect(230, 285, 200, 20)

   // update the green background view frame //
   GreenBackgroundView.frame = NSMakeRect(30, 245, 600, 40)

   // update the teal background text field frame //
   TealBackgroundField.frame = NSMakeRect(230, 210, 200, 20)

   // update the teal background view frame //
   TealBackgroundView.frame = NSMakeRect(30, 170, 600, 40)

   // update the yellow background text field frame //
   YellowBackgroundField.frame = NSMakeRect(230, 135, 200, 20)

   // update the yellow background view frame //
   YellowBackgroundView.frame = NSMakeRect(30, 95, 600, 40)

   // update the gray background text field frame //
   GrayBackgroundField.frame = NSMakeRect(230, 60, 200, 20)

   // update the gray background view frame //
   GrayBackgroundView.frame = NSMakeRect(30, 20, 600, 40)
}

// update the font preferences layout //
func FontPreferencesLayout()
{
   // update the board labels field frame //
   BoardLabelsField.frame = NSMakeRect(230, 585, 200, 20)

   // update the board labels font button frame //
   BoardLabelsFontButton.frame = NSMakeRect(230, 560, 200, 25)

   // update the board labels bold button frame //
   BoardLabelsBoldButton.frame = NSMakeRect(230, 535, 200, 25)

   // update the board labels font sizes button frame //
   BoardLabelsSizesButton.frame = NSMakeRect(230, 510, 200, 25)

   // update the board labels color view frame //
   BoardLabelsColorView.frame = NSMakeRect(30, 470, 600, 40)

   // update the game info field frame //
   GameInfoField.frame = NSMakeRect(230, 435, 200, 20)

   // update the game info font button frame //
   GameInfoFontButton.frame = NSMakeRect(230, 410, 200, 25)

   // update the game info bold button frame //
   GameInfoBoldButton.frame = NSMakeRect(230, 385, 200, 25)

   // update the game info sizes button frame //
   GameInfoSizesButton.frame = NSMakeRect(230, 360, 200, 25)

   // update the game info color view frame //
   GameInfoColorView.frame = NSMakeRect(30, 320, 600, 40)

   // update the move list field frame //
   MoveListField.frame = NSMakeRect(230, 285, 200, 20)

   // update the move list font button frame //
   MoveListFontButton.frame = NSMakeRect(230, 260, 200, 25)

   // update the move list bold button frame //
   MoveListBoldButton.frame = NSMakeRect(230, 235, 200, 25)

   // update the move list font sizes button frame //
   MoveListSizesButton.frame = NSMakeRect(230, 210, 200, 25)

   // update the move list color view frame //
   MoveListColorView.frame = NSMakeRect(30, 170, 600, 40)

   // update the game list field frame //
   GameListField.frame = NSMakeRect(230, 135, 200, 20)

   // update the game list font button frame //
   GameListFontButton.frame = NSMakeRect(230, 110, 200, 25)

   // update the game list bold button frame //
   GameListBoldButton.frame = NSMakeRect(230, 85, 200, 25)

   // update the game list font sizes button frame //
   GameListSizesButton.frame = NSMakeRect(230, 60, 200, 25)

   // update the game list color view frame //
   GameListColorView.frame = NSMakeRect(30, 20, 600, 40)
}

// update the stockfish preferences layout //
func StockfishPreferencesLayout()
{
   // update the autoload stockfish button frame //
   AutoloadStockfishButton.frame = NSMakeRect(30, 580, 190, 20)

   // update the multithread uci engine button frame //
   MultithreadUCIEngineButton.frame = NSMakeRect(30, 530, 190, 20)

   // update the show stockfish analysis button frame //
   ShowStockfishAnalysisButton.frame = NSMakeRect(30, 480, 190, 20)

   // update the uci engine difficulty slider frame //
   UCIEngineDifficultySlider.frame = NSMakeRect(30, 435, 600, 13)

   // update the uci engine difficulty view frame //
   UCIEngineDifficultyView.frame = NSMakeRect(30, 417, 600, 20)

   // update the uci engine hash slider frame //
   UCIEngineHashSlider.frame = NSMakeRect(30, 385, 600, 13)

   // update the uci engine hash view frame //
   UCIEngineHashView.frame = NSMakeRect(30, 367, 600, 20)

   // update the uci engine riskiness slider frame //
   UCIEngineRiskinessSlider.frame = NSMakeRect(30, 335, 600, 13)

   // update the uci engine riskiness view frame //
   UCIEngineRiskinessView.frame = NSMakeRect(30, 317, 600, 20)
}

// update the clock preferences layout //
func ClockPreferencesLayout()
{
   // update the game clock color field frame //
   GameClockColorField.frame = NSMakeRect(230, 585, 200, 20)

   // update the clock color view frame //
   ClockColorView.frame = NSMakeRect(30, 545, 600, 40)

   // update the game clock background color field frame //
   GameClockBackgroundColorField.frame = NSMakeRect(230, 500, 200, 20)

   // update the clock background color view frame //
   ClockBackgroundColorView.frame = NSMakeRect(30, 460, 600, 40)

   // update the clock background visibility slider //
   ClockBackgroundVisibilitySlider.frame = NSMakeRect(30, 430, 600, 13)

   // update the clock background opacity view //
   ClockBackgroundOpacityView.frame = NSMakeRect(30, 412, 600, 20)

   // update the game clock preview //
   GameClockPreview.frame = NSMakeRect(175, 230, 310, 130)
}

// update the speed preferences layout //
func SpeedPreferencesLayout()
{
   // update the animation rate slider frame //
   AnimationRateSlider.frame = NSMakeRect(30, 585, 600, 13)

   // update the animation rate view frame //
   AnimationRateView.frame = NSMakeRect(30, 567, 600, 20)

   // update the gallery and grid animation rate slider frame //
   GalleryGridAnimationRateSlider.frame = NSMakeRect(30, 535, 600, 13)

   // update the gallery and grid animation rate view frame //
   GalleryGridAnimationRateView.frame = NSMakeRect(30, 517, 600, 20)

   // update the search time slider frame //
   SearchTimeSlider.frame = NSMakeRect(30, 485, 600, 13)

   // update the search time view frame //
   SearchTimeView.frame = NSMakeRect(30, 467, 600, 20)
}

// update the clock preference state //
func ClockPreferencesToState(_ State:Bool)
{
   // update the game clock color field state //
   GameClockColorField.isHidden = State

   // update the clock color view state //
   ClockColorView.isHidden = State

   // update the game clock background color field state //
   GameClockBackgroundColorField.isHidden = State

   // update the clock background color view state //
   ClockBackgroundColorView.isHidden = State

   // update the game clock preview state //
   GameClockPreview.isHidden = State

   // update the clock background visibility slider state //
   ClockBackgroundVisibilitySlider.isHidden = State

   // update the clock background opacity view //
   ClockBackgroundOpacityView.isHidden = State
}

// update the speed preferences state //
func SpeedPreferencesToState(_ State:Bool)
{
   // update the animation rate slider state //
   AnimationRateSlider.isHidden = State

   // update the animation rate view state //
   AnimationRateView.isHidden = State

   // update the gallery and grid animation rate slider state //
   GalleryGridAnimationRateSlider.isHidden = State

   // update the gallery and grid animation rate view state //
   GalleryGridAnimationRateView.isHidden = State

   // update the search time slider state //
   SearchTimeSlider.isHidden = State

   // update the search time view state //
   SearchTimeView.isHidden = State
}

// the clock preferences action //
@IBAction func ClockPreferencesAction(_:AnyObject)
{
   // update the game clock preview //
   GameClockPreview.UpdateClock(0, ElapsedBlackTime:0)

   // update the clock preferences state //
   ClockPreferencesToState(false)

   // update the speed preferences state //
   SpeedPreferencesToState(true)

   // update the stockfish preferences state //
   StockfishPreferencesToState(true)

   // update the font preferences state //
   FontPreferencesToState(true)

   // update the background preferences state //
   BackgroundPreferencesToState(true)

   // update the board preferences state //
   BoardPreferencesToState(true)

   // update the clock preferences layout //
   ClockPreferencesLayout()
}

// the speed preferences action //
@IBAction func SpeedPreferencesAction(_:AnyObject)
{
   // update the clock preferences state //
   ClockPreferencesToState(true)

   // update the speed preferences state //
   SpeedPreferencesToState(false)

   // update the stockfish preferences state //
   StockfishPreferencesToState(true)

   // update the font preferences state //
   FontPreferencesToState(true)

   // update the background preferences state //
   BackgroundPreferencesToState(true)

   // update the board preferences state //
   BoardPreferencesToState(true)

   // update the speed preferences layout //
   SpeedPreferencesLayout()
}

// the stockfish preferences action //
@IBAction func StockfishPreferencesAction(_:AnyObject)
{
   // update the clock preferences state //
   ClockPreferencesToState(true)

   // update the speed preferences state //
   SpeedPreferencesToState(true)

   // update the stockfish preferences state //
   StockfishPreferencesToState(false)

   // update the font preferences state //
   FontPreferencesToState(true)

   // update the background preferences state //
   BackgroundPreferencesToState(true)

   // update the board preferences state //
   BoardPreferencesToState(true)

   // update the stockfish prferences layout //
   StockfishPreferencesLayout()
}

// the board preferences action //
@IBAction func BoardPreferencesAction(_:AnyObject)
{
   // update the clock preferences state //
   ClockPreferencesToState(true)

   // update the speed preferences state //
   SpeedPreferencesToState(true)

   // update the stockfish preferences state //
   StockfishPreferencesToState(true)

   // update the font preferences state //
   FontPreferencesToState(true)

   // update the background preferences state //
   BackgroundPreferencesToState(true)

   // update the board preferences state //
   BoardPreferencesToState(false)

   // update the board preferences layout //
   BoardPreferencesLayout()
}

// the background preferences state action //
@IBAction func BackgroundPreferencesAction(_:AnyObject)
{
   // update the clock preferences state //
   ClockPreferencesToState(true)

   // update the speed preferences state //
   SpeedPreferencesToState(true)

   // update the stockfish preferences state //
   StockfishPreferencesToState(true)

   // update the font preferences state //
   FontPreferencesToState(true)

   // update the background preferences state //
   BackgroundPreferencesToState(false)

   // update the board preferences state //
   BoardPreferencesToState(true)

   // update the background preferences layout //
   BackgroundPreferencesLayout()
}

// the font preferences action //
@IBAction func FontPreferencesAction(_:AnyObject)
{
   // update the clock preferences state //
   ClockPreferencesToState(true)

   // update the speed preferences state //
   SpeedPreferencesToState(true)

   // update the stockfish preferences state //
   StockfishPreferencesToState(true)

   // update the font preferences state //
   FontPreferencesToState(false)

   // update the background preferences state //
   BackgroundPreferencesToState(true)

   // update the board preferences state //
   BoardPreferencesToState(true)

   // update the font preferences layout //
   FontPreferencesLayout()
}

// the preferences menu item action //
@IBAction func PreferencesMenuItemAction(_:AnyObject)
{
   // add a timer to run the preferences menu item action //
   RunLoop.current.add(Timer(timeInterval:0.01, target:self, selector:
                  #selector(GreenChess.ThePreferencesMenuItemAction), userInfo:nil,
                  repeats:false), forMode:RunLoopMode.commonModes)
}

// the preferences menu item action //
@objc func ThePreferencesMenuItemAction()
{
   // set that currently entering in preferences //
   EnteringPreferences = true

   // update the move list //
   UpdateMoveList()

   // update the board preview //
   UpdateBoardPreview()

   // begin the preferences sheet //
   MainWindow.beginSheet(PreferencesWindow, completionHandler:nil)

   // turn the preferences menu item on //
   PreferencesMenuItem.state = NSControl.StateValue(rawValue:1)
}

// search the game list menu item action //
@IBAction func SearchGameListMenuItemAction(_:AnyObject)
{
   // set that currently searching the game list window //
   SearchingGameList = true

   // update the game list data //
   UpdateTheGameList()

   // reload the data rows //
   ReloadDataRows()

   // display the current game count //
   DisplayGameCount()

   // open the game list window sheet //
   MainWindow.beginSheet(GameListWindow, completionHandler:nil)

   // set the search game list menu item to on //
   SearchGameListMenuItem.state = NSControl.StateValue(rawValue:1)

   // update the game list search box to the current search string value //
   GameListSearchBox.stringValue = SearchString
}

// remove toolbar items //
func RemoveToolbarItems(_ Toolbar:NSToolbar)
{
   // while there are toolbar items left //
   while Toolbar.items.count > 0
   {
      // enable the toolbar menu item //
      Toolbar.items[Toolbar.items.count - 1].isEnabled = true

      // remove the specified toolbar item //
      Toolbar.removeItem(at:Toolbar.items.count - 1)
   }
}

// add the default toolbar items //
func AddDefaultItems()
{
   // remove all the current toolbar items //
   RemoveToolbarItems(MainToolbar)

   // add a flexible space to the toolbar //
   MainToolbar.insertItem(withItemIdentifier:NSToolbarItem.Identifier(rawValue:
                     "NSToolbarFlexibleSpaceItem"), at:MainToolbar.items.count)

   // add the opponent icon to the toolbar //
   MainToolbar.insertItem(withItemIdentifier:NSToolbarItem.Identifier(rawValue:"100"), at:
                     MainToolbar.items.count)

   // add a flexible space to the toolbar //
   MainToolbar.insertItem(withItemIdentifier:NSToolbarItem.Identifier(rawValue:
                     "NSToolbarFlexibleSpaceItem"), at:MainToolbar.items.count)
}

// add the diagram items to the toolbar //
func AddDiagramItems()
{
   // remove all the current toolbar items //
   RemoveToolbarItems(MainToolbar)

   // loop over the toolbar items //
   for Index in 300 ... 311
   {
      // add a toolbar item //
      MainToolbar.insertItem(withItemIdentifier:NSToolbarItem.Identifier(rawValue:"\(Index)"),
                     at:MainToolbar.items.count)

      // if the index is not 311 //
      if Index != 311
      {
         // add a flexible space to the toolbar //
         MainToolbar.insertItem(withItemIdentifier:NSToolbarItem.Identifier(rawValue:
                        "NSToolbarFlexibleSpaceItem"), at:MainToolbar.items.count)
      }
   }
}

// add the position editor piece toolbar items //
func AddPositionEditorPieceItems()
{
   // remove the current toolbar items //
   RemoveToolbarItems(MainToolbar)

   // loop over toolbar items //
   for Index in 200 ... 215
   {
      // add the toolbar item at the specified index //
      MainToolbar.insertItem(withItemIdentifier:NSToolbarItem.Identifier(rawValue:"\(Index)"),
                     at:MainToolbar.items.count)

      // if the index is not 215 //
      if Index != 215
      {
         // add a flexible space to the toolbar //
         MainToolbar.insertItem(withItemIdentifier:NSToolbarItem.Identifier(rawValue:
                        "NSToolbarFlexibleSpaceItem"), at:MainToolbar.items.count)
      }
   }

   // set the position editor second menu button image to the going right icon //
   PositionEditorSecondMenu.image = NSImage(named:NSImage.Name(rawValue:"right_icon"))
}

// add the position editor castling toolbar items //
func AddPositionEditorCastlingItems()
{
   // remove the current toolbar items //
   RemoveToolbarItems(MainToolbar)

   // loop over load and left or right toolbar items //
   for Index in 200 ... 201
   {
      // add the toolbar items at the selected index //
      MainToolbar.insertItem(withItemIdentifier:NSToolbarItem.Identifier(rawValue:"\(Index)"),
                     at:MainToolbar.items.count)

      // add a flexible space to the toolbar //
      MainToolbar.insertItem(withItemIdentifier:NSToolbarItem.Identifier(rawValue:
                     "NSToolbarFlexibleSpaceItem"), at:MainToolbar.items.count)
   }

   // loop over toolbar items //
   for Index in 216 ... 220
   {
      // add the toolbar item at the specified index //
      MainToolbar.insertItem(withItemIdentifier:NSToolbarItem.Identifier(rawValue:"\(Index)"),
                     at:MainToolbar.items.count)

      // if the index is not 220 //
      if Index != 220
      {
         // add a flexible space to the toolbar //
         MainToolbar.insertItem(withItemIdentifier:NSToolbarItem.Identifier(rawValue:
                        "NSToolbarFlexibleSpaceItem"), at:MainToolbar.items.count)
      }
   }

   // set the position editor second menu button image to the going left icon //
   PositionEditorSecondMenu.image = NSImage(named:NSImage.Name(rawValue:"left_icon"))
}

// go left in the list of saved positions //
@IBAction func PositionButtonLeftAction(_:AnyObject)
{
   // if there is only one saved position //
   if SavedPositionData.count == 1
   {
      // return since there is only one saved position //
      return
   }

   // if at the beginning of the saved position list //
   if LoadPositionIndex == 0
   {
      // set the load position index to the end of the list //
      LoadPositionIndex = SavedPositionData.count - 1
   }
   // not at the beginning of the saved position list //
   else
   {
      // decrement the load position index //
      LoadPositionIndex -= 1
   }

   // animate the position image changing //
   AnimatePositionChanging()
}

// go right button in list of saved positions //
@IBAction func PositionButtonRightAction(_:AnyObject)
{
   // if there is only one saved position //
   if SavedPositionData.count == 1
   {
      // return since there is only one saved position //
      return
   }

   // if the load position index is at the end of the list //
   if LoadPositionIndex == SavedPositionData.count - 1
   {
      // set the load position index to the beginning //
      LoadPositionIndex = 0
   }
   // the load position index is not at the end of the list //
   else
   {
      // increment the load poation index //
      LoadPositionIndex += 1
   }

   // animate the position image changing //
   AnimatePositionChanging()
}

// exit the load position window //
@IBAction func ExitLoadPositionWindow(_:AnyObject)
{
   // play the game sound of a button click //
   PlayGameSoundThread("buttonclick")

   // remove all the animations from position layer one //
   PositionLayers[0].removeAllAnimations()

   // remove all the animations from position layer two //
   PositionLayers[1].removeAllAnimations()

   // turn off the load position menu item //
   LoadPositionMenuItem.state = NSControl.StateValue(rawValue:0)

   // end the load position window //
   MainWindow.endSheet(LoadPositionWindow)

   // close the load position window //
   LoadPositionWindow.close()

   // set loading saved position to false //
   LoadingSavedPosition = false
}

// the load position menu item action //
@IBAction func LoadPositionMenuItemAction(_:AnyObject)
{
   // turn on the load position menu item //
   LoadPositionMenuItem.state = NSControl.StateValue(rawValue:1)

   // if the saved position data is empty //
   if SavedPositionData.isEmpty
   {
      // set query load position to true //
      QueryLoadPosition = true

      // play the sound of a warning message //
      PlayGameSoundThread("warning")

      // set the alert button to be okay //
      AlertButtons = ["Okay"]

      // let user know that there are no saved position //
      MakeAlert(AlertButtons, Query:"No Saved Boards")

      // exit the function //
      return
   }

   // set that currently trying to load a saved position //
   LoadingSavedPosition = true

   // copy the current saved position into position layer one //
   PositionLayers[0].contents = SavedPositionImages[LoadPositionIndex]

   // set position layer two opacity to zero //
   PositionLayers[1].opacity = 0

   // set the position layer one position //
   PositionLayers[0].position = CGPoint(x:250, y:250)

   // begin the load position window sheet //
   MainWindow.beginSheet(LoadPositionWindow, completionHandler:nil)
}

// animate a saved position layer changing //
func AnimatePositionChanging()
{
   // remove all the animations from position layer one //
   PositionLayers[0].removeAllAnimations()

   // remove all the animations from position layer two //
   PositionLayers[1].removeAllAnimations()

   // remove position later two from the super layer //
   PositionLayers[1].removeFromSuperlayer()

   // initialize position layer two //
   PositionLayers[1] = CALayer()

   // add the position layer two to the load position view //
   LoadPositionView.layer?.addSublayer(PositionLayers[1])

   // set the position layer two border color to white //
   PositionLayers[1].borderColor = NSColor.white.cgColor

   // set the position layer two border width to 1 //
   PositionLayers[1].borderWidth = 1

   // set the position layer two bounds //
   PositionLayers[1].bounds = CGRect(x:0, y:0, width:500, height:500)

   // set the position layer two position //
   PositionLayers[1].position = CGPoint(x:250, y:250)

   // set the position layer two contents to position layer one contents //
   PositionLayers[1].contents = PositionLayers[0].contents

   // set the position layer one contents to the new saved position image //
   PositionLayers[0].contents = SavedPositionImages[LoadPositionIndex]

   // set animation opacity to start at 1 //
   PositionAnimation.fromValue = 1

   // set animation opacity to end at 0 //
   PositionAnimation.toValue = 0

   // set ending position layer two opacity to zero //
   PositionLayers[1].opacity = 0

   // add an animation to position layer two //
   PositionLayers[1].add(PositionAnimation, forKey:"opacity")
}

// the save position menu item action //
@IBAction func SavePositionMenuItemAction(_:AnyObject)
{
   // set query save position to true //
   QuerySavePosition = true

   // turn on the save position menu item state //
   SavePositionMenuItem.state = NSControl.StateValue(rawValue:1)

   // take a snapshot of the current board //
   TakeBoardSnapshot(660)

   // save the snapshot image to game attributes //
   Attributes.PositionImage = SnapshotImage

   // convert the current game to game data //
   ConvertToGameData()

   // create a new element at the end of the saved positions data array //
   SavedPositionData.append(DataNew())

   // copy the game data to the end of the saved positions data array //
   DataCopy(SavedPositionData[SavedPositionData.count - 1], GameData)

   // append the snapshot image to saved position images //
   SavedPositionImages.append(SnapshotImage)

   // store the autosave data to the saved position slot //
   StoreAutosaveData("\(SavedPositionSlot)")

   // if autosaving game data //
   if AutosaveGame
   {
      // increment the saved position slot index //
      SavedPositionSlot = SavedPositionSlot + 1

      // if the saved position slot is greater than 10 //
      if SavedPositionSlot > 10
      {
         // reset the saved position slot to 1 //
         SavedPositionSlot = 1
      }

      // save the saved position slot index to preferences //
      AppPreferences.set(SavedPositionSlot, forKey:"SavedPositionSlot")
   }

   // set the alert button to be okay //
   AlertButtons = ["Okay"]

   // make an alert that the game data was just saved //
   MakeAlert(AlertButtons, Query:"Saved Board")
}

// the load position button action //
@IBAction func LoadPositionButtonAction(_:AnyObject)
{
   // copy the saved position data into the game data //
   DataCopy(GameData, SavedPositionData[LoadPositionIndex])

   // save whether was currently analyzing the game or not //
   WasAnalyzingGame = AnalyzingGame

   // start a new game //
   StartNewGame(false)

   // extact and display the game information from the game data list //
   ExtractFromGameData()

   // reset the game clock //
   ResetGameClock()

   // if was currently analyzing a game //
   if WasAnalyzingGame
   {
      // set that in game analysis mode //
      AnalyzingGame = true

      // set that in game explorer //
      ExploringGame = true

      // set game explorer menu item to show exit game explorer analysis mode //
      GameExplorerMenuItem.title = "Exit Game Explorer Analysis Mode"

      // set title to show that in the game explorer analysis mode //
      MainWindow.title = "GreenChess (Game Explorer Analysis Mode)"
   }

   // store the autosave data //
   StoreAutosaveData("")

   // exit the load position window //
   ExitLoadPositionWindow(self)
}

// the about window menu item action //
@IBAction func AboutMenuItemAction(_:AnyObject)
{
   // set that currently in the about window //
   InAboutWindow = true

   // close the attached sheet //
   CloseAttachedSheet()

   // begin the about window //
   MainWindow.beginSheet(AboutWindow, completionHandler:nil)

   // turn on the about menu item //
   AboutMenuItem.state = NSControl.StateValue(rawValue:1)
}

// the enter game info menu item action //
@IBAction func EnterGameInfoMenuItemAction(_:AnyObject)
{
   // set that currently entering in game info //
   EnteringGameInfo = true

   // open the enter game info window as a sheet on the main window //
   MainWindow.beginSheet(EnterGameInfoWindow, completionHandler:nil)

   // turn the enter game info menu item on //
   EnterGameInfoMenuItem.state = NSControl.StateValue(rawValue:1)
}

// start a timed game //
@IBAction func StartTimedGame(_:AnyObject)
{
   // start a new game //
```

```
      StartNewGame(true)

      // set that currently playing a timed game //
      PlayingTimedGame = true

      // set that want to start the clock //
      StartClock = true

      // set main window to show that currently in timed game mode //
      MainWindow.title = "GreenChess (Timed Game Mode)"

      // save the original animation rate //
      OriginalAnimationRate = AnimationRate

      // set the animation rate to the maximum value //
      AnimationRate = 100

      // save whether using the opening book or not //
      OriginalUseEcoBook = UseEcoBook

      // set that currently using opening book //
      UseEcoBook = true

      // get the index of the timed game opponent //
      TimedGameOpponent = TimedGameOpponentButton.indexOfSelectedItem

      // get the index of the timed game color button //
      TimedGameColor = TimedGameColorButton.indexOfSelectedItem

      // get the index of the timed game time control button //
      TimedGameTime = TimedGameTimeButton.indexOfSelectedItem

      // if user chose to play against the greenchess engine //
      if TimedGameOpponent == 0
      {
         // set opponent to be the greenchess engine //
         Opponent(false, Engine:true, UCIEngine:false)
      }
      // user chose uci engine //
      else
      {
         // set the opponent to be uci engine //
         Opponent(false, Engine:false, UCIEngine:true)
      }

      // calculate the total game time //
      TotalGameTime = 9000 * (TimedGameTime + 1)

      // calculate the timed game offset from the six o'clock position //
      TimedGameOffset = 216000 - TotalGameTime

      // set the total amount of time remaining for white //
      RemainingWhiteTime = TotalGameTime

      // set the total amount of time remaining for black //
      RemainingBlackTime = TotalGameTime

      // reset the game clock //
      ResetGameClock()

      // update the timed game menu item to show exit timed game mode //
      TimedGameMenuItem.title = "Exit Timed Game Mode"

      // close the timed game window //
      CloseTimedGameWindow()

      // if user is playing as black against greenchess or a uci engine //
      if TimedGameColor == 1
      {
         // start an engine move //
         StartEngineMove()
      }
   }

   // remove the timed game offset //
   func RemoveTimedGameOffset()
   {
      // remove the timed game offset from the elapsed white time //
      ElapsedWhiteTime -= TimedGameOffset

      // remove the timed game offset from the elapsed black time //
      ElapsedBlackTime -= TimedGameOffset

      // reset the timed game offset to zero //
      TimedGameOffset = 0

      // reset the animation rate to the original value //
      AnimationRate = OriginalAnimationRate

      // reset the eco book status to the original value //
      UseEcoBook = OriginalUseEcoBook

      // reset game menu item to play timed game //
      TimedGameMenuItem.title = "Timed Game"

      // reset the mainwindow title to greenchess //
      MainWindow.title = "GreenChess"

      // set that no longer querying if timed game is over //
      QueryTimedGameOver = false

      // set that no longer playing a timed game //
      PlayingTimedGame = false
   }

   // the enter fen menu item action //
   @IBAction func EnterFENMenuItemAction(_:AnyObject)
   {
      // set that currently in the enter fen window //
      InEnterFENWindow = true

      // set that currently entering fen to convert //
      EnteringFEN = true

      // turn on the enter fen menu item state //
      EnterFENMenuItem.state = NSControl.StateValue(rawValue:1)

      // save the previous main window title //
      PreviousMainWindowTitle = MainWindow.title

      // set the main window title to fen converter //
      MainWindow.title = "FEN Converter"

      // start the fen window //
      MainWindow.beginSheet(FENWindow, completionHandler:nil)
   }

   // the timed game menu item action //
   @IBAction func TimedGameMenuItemAction(_:AnyObject)
   {
      // if currently playing a timed game //
      if PlayingTimedGame
      {
         // remove the timed game offset //
         RemoveTimedGameOffset()

         // exit function since done //
         return
      }

      // set that currently entering timed game info //
      EnteringTimedGameInfo = true

      // turn on the timed game menu item //
      TimedGameMenuItem.state = NSControl.StateValue(rawValue:1)

      // start the timed game window //
      MainWindow.beginSheet(TimedGameWindow, completionHandler:nil)
   }

   // function to run the game annotator menu item action //
   @IBAction func GameAnnotatorMenuItemAction(_:AnyObject)
   {
      // add a timer to run the game annotator menu item action //
      RunLoop.current.add(Timer(timeInterval:0.01, target:self, selector:#selector(GreenChess
                          .TheGameAnnotatorMenuItemAction), userInfo:nil, repeats:false), forMode:
                          RunLoopMode.commonModes)
   }

   // the game annotator menu item action //
   @objc func TheGameAnnotatorMenuItemAction()
   {
      // set that currently annotating a game //
      AnnotatingGame = true

      // convert the game to frames //
      ConvertGameToFrames()

      // set the annotation frames //
      AnnotationFrames = Frames

      // set the main window title to be in the game annotator mode //
      MainWindow.title = "GreenChess (Game Annotator Mode)"

      // turn on the game annotator menu item //
      GameAnnotatorMenuItem.state = NSControl.StateValue(rawValue:1)

      // set the minimum value of the game annotator slider to be one //
      GameAnnotatorSlider.minValue = 1

      // set the initial value of the game annotator slider to be one //
      GameAnnotatorSlider.intValue = 1

      // remove the animation effects //
      CATransaction.setDisableActions(true)

      // set the annotation board layer left contents to the board composite //
      AnnotationBoardLayerLeft.contents = BoardComposite

      // remove the animation effects //
      CATransaction.setDisableActions(true)

      // set the annotation board layer middle contents to the board composite //
      AnnotationBoardLayerMiddle.contents = BoardComposite

      // remove the animation effects //
      CATransaction.setDisableActions(true)

      // set the annotation board layer right contents to the board composite //
      AnnotationBoardLayerRight.contents = BoardComposite

      // remove the animation effects //
      CATransaction.setDisableActions(true)

      // set the annotation pieces layer left contents to an empty image //
      AnnotationPiecesLayerLeft.contents = NSImage(size:NSSize(width:250, height:250))

      // remove the animation effects //
      CATransaction.setDisableActions(true)

      // set the annotation pieces layer middle contents to an empty image //
      AnnotationPiecesLayerMiddle.contents = NSImage(size:NSSize(width:250, height:250))

      // remove the animation effects //
      CATransaction.setDisableActions(true)

      // set the annotation pieces layer right contents to an empty image //
      AnnotationPiecesLayerRight.contents = NSImage(size:NSSize(width:250, height:250))

      // remove the animation effects //
      CATransaction.setDisableActions(true)

      // if there are no moves in the game //
      if MoveList.isEmpty
      {
         // set pgn moves view to be empty //
         PGNMoveView.stringValue = ""

         // print that there are currently no moves to annotate //
         GameAnnotationTextField.string = "There are currently no moves to annotate."

         // make the game annotation text field editing temporarily unavailable //
         GameAnnotationTextField.isEditable = false

         // make the game annotation text field selection temporarily unavailable //
         GameAnnotationTextField.isSelectable = false

         // set the maxmium value of the game annotator slider to one //
         GameAnnotatorSlider.maxValue = 1
      }
      // there are currently moves in the game //
      else
      {
         // make the game annotation text field editable //
         GameAnnotationTextField.isEditable = true

         // make the game annotation text field selectable //
         GameAnnotationTextField.isSelectable = true

         // set the maximum value of the game annotator slider to number of moves //
         GameAnnotatorSlider.maxValue = Double(MoveList.count)

         // update the game annotator to display the first move //
         UpdateGameAnnotator(1)

         // set the game annotator move location to one //
         GameAnnotatorMoveLocation = 1
      }

      // open the game annotator window //
      MainWindow.beginSheet(GameAnnotatorWindow, completionHandler:nil)
   }

   // load the entered fen to convert //
   @IBAction func LoadEnteredFEN(_:AnyObject)
   {
      // get the entered fen from the enter fen text field //
      EnteredFENString = StringFromTextField(EnterFENTextField.string)

      // if the entered fen string is empty //
      if EnteredFENString.isEmpty
      {
         // close the enter fen window //
         CloseFENWindow()

         // exit function //
         return
      }

      // save the current board layout //
      SaveBoardLayout()

      // parse the entered fen string to convert //
      RawEnteredFENStringArray = EnteredFENString.components(separatedBy:" ")

      // clear the entered fen string array //
      EnteredFENStringArray.removeAll()

      // loop over all the items in the raw entered fen string array //
      for Index in 0 ..< RawEnteredFENStringArray.count
      {
         // if the raw entered string array element is not empty //
         if RawEnteredFENStringArray[Index] != ""
         {
            // copy over the nonempty element to the entered fen string array //
            EnteredFENStringArray.append(RawEnteredFENStringArray[Index])
         }
      }

      // if the entered fen string array only has one element //
      if EnteredFENStringArray.count == 1
      {
         // append the default white color //
         EnteredFENStringArray.append("w")
      }

      // if the entered fen string array only has two elements //
      if EnteredFENStringArray.count == 2
      {
         // append the default castling rights //
         EnteredFENStringArray.append("KQkq")
      }

      // if the entered fen string array only has three elements //
      if EnteredFENStringArray.count == 3
      {
         // append the default enpassant square //
         EnteredFENStringArray.append("-")
      }

      // if the entered fen string only has four elements //
      if EnteredFENStringArray.count == 4
      {
         // append the default fifty move rule value //
         EnteredFENStringArray.append("0")
      }

      // if the entered fen string array only has five elements //
      if EnteredFENStringArray.count == 5
      {
         // append the default start move number //
         EnteredFENStringArray.append("1")
      }

      // set the fen layout from the entered fen string //
      Attributes.FENLayout = EnteredFENStringArray[0]

      // set the fen color from the entered fen string //
      Attributes.FENColor = EnteredFENStringArray[1].lowercased()

      // set the fen castling rights from the entered fen string //
      Attributes.FENCastlingRights = EnteredFENStringArray[2]

      // set the fen enpassant square from the entered fen string //
      Attributes.FENEnpassantSquare = EnteredFENStringArray[3].lowercased()

      // set the fen start move number from the entered fen string //
      Attributes.FENStartMoveNumber = EnteredFENStringArray[5]

      // close the enter fen window //
      CloseFENWindow()

      // set the initial layout //
      SetInitialLayout()

      // send the fen to convert to the position editor //
      PositionEditorMenuItemAction(self)
   }

   // close the enter fen window //
   func CloseFENWindow()
   {
      // turn off the enter fen menu item //
      EnterFENMenuItem.state = NSControl.StateValue(rawValue:0)

      // play the sound of a button click //
      PlayGameSoundThread("buttonclick")

      // end the enter fen window //
      MainWindow.endSheet(FENWindow)

      // close the enter fen window //
      FENWindow.close()

      // reset the main window title //
      MainWindow.title = PreviousMainWindowTitle

      // set that no longer in the enter fen window //
      InEnterFENWindow = false
   }

   // exit the enter fen window //
   @IBAction func ExitFENWindow(_:AnyObject)
   {
      // set that no longer entering fen to convert //
      EnteringFEN = false

      // close the fen window //
      CloseFENWindow()
   }

   // exit timed game window //
   @IBAction func ExitTimedGameWindow(_:AnyObject)
   {
      // close the timed game window //
      CloseTimedGameWindow()
   }

   // close the timed game window //
   func CloseTimedGameWindow()
   {
      // set the timed game menu item state to off //
      TimedGameMenuItem.state = NSControl.StateValue(rawValue:0)

      // play the sound of a button click //
      PlayGameSoundThread("buttonclick")

      // end the timed game window sheet //
      MainWindow.endSheet(TimedGameWindow)

      // close the timed game window //
      TimedGameWindow.close()

      // set that no longer entering timed game info //
      EnteringTimedGameInfo = false
   }

   // exit the game annotator //
   @IBAction func ExitGameAnnotator(_:AnyObject)
   {
      // turn off the game annotator menu item //
      GameAnnotatorMenuItem.state = NSControl.StateValue(rawValue:0)

      // play the sound of a button being clicked //
      PlayGameSoundThread("buttonclick")

      // end the game annotator window //
      MainWindow.endSheet(GameAnnotatorWindow)

      // close the game annotator window //
      GameAnnotatorWindow.close()

      // if there were moves in the game //
      if MoveList.count > 0
      {
         // get the current game annotator move location //
         GameAnnotatorMoveLocation = Int(GameAnnotatorSlider.intValue)

         // process the entered move annotation //
         ProcessEnteredAnnotation()
      }

      // reset the main title to be greenchess //
      MainWindow.title = "GreenChess"

      // set that no longer annotating the game //
      AnnotatingGame = false

      // replay all the moves //
      ReplayMoves(0, End:MoveList.count, Display:false)

      // store the autosave data //
      StoreAutosaveData("")
   }

   // exit the diagram mode //
   @IBAction func ExitDiagramMode(_:AnyObject)
   {
      // play the sound of a button click //
      PlayGameSoundThread("buttonclick")

      // the game explorer menu item action //
      GameExplorerMenuItemAction(self)
   }

   // the game explorer menu item action //
   @IBAction func GameExplorerMenuItemAction(_:AnyObject)
   {
      // add a timer to run the game explorer menu item action //
      RunLoop.current.add(Timer(timeInterval:0.01, target:self, selector:
                          #selector(GreenChess.TheGameExplorerMenuItemAction), userInfo:nil,
                          repeats:false), forMode:RunLoopMode.commonModes)
   }

   // the game explorer menu item action //
   @objc func TheGameExplorerMenuItemAction()
   {
      // if not analyzing and not diagramming game //
      if !AnalyzingGame && !Diagramming
      {
         // set the game explorer move location to zero //
         GameExplorerSlider.intValue = 0
      }

      // set that the game explorer window is currently open //
      GameExplorerWindowOpen = true

      // if already currently analyzing the game //
      if AnalyzingGame
      {
         // set analyzing game to be false //
         AnalyzingGame = false

         // reset the game data //
         ResetGameData()
      }
      // if currently diagramming //
      else if Diagramming
      {
         // set that done with diagramming mode //
         Diagramming = false

         // exit the diagram mode //
         ExitDiagramMode()
      }

      // set main window title to game explorer mode //
      MainWindow.title = "GreenChess (Game Explorer Mode)"

      // set the game explorer menu item title //
      GameExplorerMenuItem.title = "Explorer"

      // turn on the game explorer menu item //
      GameExplorerMenuItem.state = NSControl.StateValue(rawValue:1)

      // turn on the game explorer buttons //
      SetGameExplorerButtons(true)

      // set that currently exploring the game //
      ExploringGame = true

      // set the minimum value of the slider //
      GameExplorerSlider.minValue = 0

      // set the maximum value of the slider //
      GameExplorerSlider.maxValue = Double(MoveList.count)

      // set the number of tick marks for the game explorer slider //
      GameExplorerSlider.numberOfTickMarks = MoveList.count + 1

      // update the game explorer board to the current board position //
      UpdateGameExplorer(self)

      // initialize the transposition table //
      EngineInitializeTransTable(Engine)

      // open the game explorer window as a sheet on the main window //
      MainWindow.beginSheet(GameExplorerWindow, completionHandler:nil)
   }

   // convert the numeric annotation glyphs in an annotation //
   func ConvertNumericAnnotationGlyphs()
   {
      // convert the numeric annotation glyph //
      AnnotationString = AnnotationString.replacingOccurrences(of:"$133", with:"â‡†")

      // convert the numeric annotation glyph //
      AnnotationString = AnnotationString.replacingOccurrences(of:"$132", with:"â‡†")

      // convert the numeric annotation glyph //
      AnnotationString = AnnotationString.replacingOccurrences(of:"$41", with:"â†‘")

      // convert the numeric annotation glyph //
      AnnotationString = AnnotationString.replacingOccurrences(of:"$40", with:"â†‘")

      // convert the numeric annotation glyph //
      AnnotationString = AnnotationString.replacingOccurrences(of:"$37", with:"â†’")

      // convert the numeric annotation glyph //
      AnnotationString = AnnotationString.replacingOccurrences(of:"$36", with:"â†’")

      // convert the numeric annotation glyph //
      AnnotationString = AnnotationString.replacingOccurrences(of:"$33", with:"âŸ³")

      // convert the numeric annotation glyph //
      AnnotationString = AnnotationString.replacingOccurrences(of:"$32", with:"âŸ³")

      // convert the numeric annotation glyph //
      AnnotationString = AnnotationString.replacingOccurrences(of:"$23", with:"â¨€")

      // convert the numeric annotation glyph //
      AnnotationString = AnnotationString.replacingOccurrences(of:"$22", with:"â¨€")

      // convert the numeric annotation glyph //
      AnnotationString = AnnotationString.replacingOccurrences(of:"$19", with:"âˆ’ +")

      // convert the numeric annotation glyph //
      AnnotationString = AnnotationString.replacingOccurrences(of:"$18", with:"+ âˆ’")

      // convert the numeric annotation glyph //
      AnnotationString = AnnotationString.replacingOccurrences(of:"$17", with:"âˆ“")

      // convert the numeric annotation glyph //
      AnnotationString = AnnotationString.replacingOccurrences(of:"$16", with:"Â±")

      // convert the numeric annotation glyph //
      AnnotationString = AnnotationString.replacingOccurrences(of:"$15", with:"â©±")

      // convert the numeric annotation glyph //
      AnnotationString = AnnotationString.replacingOccurrences(of:"$14", with:"â©²")

      // convert the numeric annotation glyph //
      AnnotationString = AnnotationString.replacingOccurrences(of:"$13", with:"âˆž")

      // convert the numeric annotation glyph //
      AnnotationString = AnnotationString.replacingOccurrences(of:"$10", with:"=")

      // convert the numeric annotation glyph //
      AnnotationString = AnnotationString.replacingOccurrences(of:"$7", with:"â–¡")

      // convert the numeric annotation glyph //
      AnnotationString = AnnotationString.replacingOccurrences(of:"$6", with:"â¨")

      // convert the numeric annotation glyph //
      AnnotationString = AnnotationString.replacingOccurrences(of:"$5", with:"â‰")

      // convert the numeric annotation glyph //
      AnnotationString = AnnotationString.replacingOccurrences(of:"$4", with:"â‡")

      // convert the numeric annotation glyph //
      AnnotationString = AnnotationString.replacingOccurrences(of:"$3", with:"â€¼")

      // convert the numeric annotation glyph //
      AnnotationString = AnnotationString.replacingOccurrences(of:"$2", with:"?")

      // convert the numeric annotation glyph //
      AnnotationString = AnnotationString.replacingOccurrences(of:"$1", with:"!")
   }

   // check to make sure have matching number of left and right parentheses //
   func CheckParenthesesBrackets()
   {
      // if the annotation string is empty //
      if AnnotationString.isEmpty
      {
         // exit the function since the annotation string is empty //
         return
      }

      // initialize number of left parentheses to zero //
      LeftParentheses = 0

      // initialize number of right parentheses to zero //
      RightParentheses = 0

      // loop over all the characters in the annotation string //
      for Index in 0 ..< AnnotationString.count
      {
         // if have encountered a left parentheses //
         if AnnotationString.prefix(Index + 1).last! == "("
         {
            // increment the number of left parentheses encountered so far //
            LeftParentheses += 1
         }
         // if have encountered a right parentheses //
         else if AnnotationString.prefix(Index + 1).last! == ")"
         {
            // increment the number of right parentheses encountered so far //
            RightParentheses += 1
         }
      }

      // if the number of left parentheses doesn't equal number or right parentheses //
      if LeftParentheses != RightParentheses
      {
         // remove all the left parentheses from the move annotation //
         AnnotationString = AnnotationString.replacingOccurrences(of:"(", with:"")

         // remove all the right parentheses from the move annotation //
         AnnotationString = AnnotationString.replacingOccurrences(of:")", with:"")
      }

      // remove all the left brackets //
      AnnotationString = AnnotationString.replacingOccurrences(of:"{", with:"")

      // remove all the right brackets //
      AnnotationString = AnnotationString.replacingOccurrences(of:"}", with:"")

      // add brackets to encompass the entire move annotation string //
      AnnotationString = "{\(AnnotationString)}"
   }

   // update the game annotator from the current slider value //
   @IBAction func UpdateGameAnnotatorFromSlider(_:AnyObject)
   {
      // if there were no moves in the game //
      if MoveList.isEmpty
      {
         // exit function since there were no moves in the game //
         return
      }

      // process the entered move annotation //
      ProcessEnteredAnnotation()

      // save the previous move location from the game annotator slider //
      GameAnnotatorMoveLocation = Int(GameAnnotatorSlider.intValue)

      // update the game annotator for the current move location of the slider //
      UpdateGameAnnotator(Int(GameAnnotatorSlider.intValue))
   }

   // update the database explorer board from the database game explorer slider //
   @IBAction func UpdateDatabaseGameExplorerFromSlider(_:AnyObject)
   {
      // if read in the game database already //
      if ReadInGameList
      {
         // replay moves from the beginning to the chosen move location from slider //
         ReplayMoves(0, End:Int(DatabaseGameExplorerSlider.intValue), Display:false)
      }
   }

   // update the database explorer board //
   @IBAction func UpdateDatabaseExplorer(_:AnyObject)
   {
      // if have not read in a game database yet //
      if !ReadInGameList
      {
         // exit function since have not read in a game database yet //
         return
      }

      // get the selected game to display //
      GameData = DatabaseGetGameData(GameData, GameList, CInt(DatabaseExplorerSlider.intValue))

      // extract and display the game information from the database game object //
      ExtractFromGameData()

      // display the current game description to the main window //
      MainWindow.title = GameListData.GameDescription(Int(DatabaseExplorerSlider.intValue))
   }

   // process an entered move annotation from the game annotator //
   func ProcessEnteredAnnotation()
   {
      // get the current annotation string from the game annotation text field //
      AnnotationString = StringFromTextField(GameAnnotationTextField.string)

      // if the annotation string is not empty //
      if !AnnotationString.isEmpty
      {
         // check to make sure have equal number of left and right parentheses //
         CheckParenthesesBrackets()
      }

      // update the annotation in the annotation list with the new annotation //
      AnnotationList[GameAnnotatorMoveLocation - 1] = AnnotationString
   }

   // update the game annotator for the current move location in the game //
   func UpdateGameAnnotator(_ MoveLocation:Int)
   {
      // in there are no moves in the game //
      if MoveList.isEmpty
      {
         // exit function since there are no moves in the game //
         return
      }

      // update the game annotator slider to the new move location //
      GameAnnotatorSlider.intValue = Int32(MoveLocation)

      // get the annotation string to display //
      AnnotationString = AnnotationList[MoveLocation - 1]

      // check the parentheses and brackets in the annotation and correct if needed //
      CheckParenthesesBrackets()

      // send the current move annotation to the game annotator text field //
      GameAnnotationTextField.string = AnnotationString

      // if the pgn move string index is within range //
      if MoveLocation - 1 < PGNMovesListForAnnotator.count
      {
         // set the pgn move string value to the pgn move //
         PGNMoveView.stringValue = PGNMovesListForAnnotator[MoveLocation - 1]
      }
      // the pgn move string index is out of bounds //
      else
      {
         // set the pgn move string value to be empty //
         PGNMoveView.stringValue = ""
      }

      // if have at least one move //
      if MoveLocation > 0
      {
         // disable the animation effects //
         CATransaction.setDisableActions(true)

         // set the annotation pieces layer left contents to the annotation frame to the left //
         AnnotationPiecesLayerLeft.contents = AnnotationFrames[MoveLocation - 1]
      }

      // disable the animation effects //
      CATransaction.setDisableActions(true)

      // set the annotation pieces layer middle contents to the current annotation frame //
      AnnotationPiecesLayerMiddle.contents = AnnotationFrames[MoveLocation]

      // disable the animation effect //
      CATransaction.setDisableActions(true)

      // if the annotation frame to the right is within range //
      if MoveLocation + 1 < AnnotationFrames.count
      {
         // set the annotation pieces layer right contents to the annotation frame to the right //
         AnnotationPiecesLayerRight.contents = AnnotationFrames[MoveLocation + 1]
      }
      // annotation frame index is out of range //
      else
      {
         // set the annotation pieces layer right contents to an empty image //
         AnnotationPiecesLayerRight.contents = NSImage(size:NSSize(width:250, height:250))
      }
   }

   // update the game explorer board for the current move location //
   @IBAction func UpdateGameExplorer(_:AnyObject)
   {
      // replay the moves from the beginning to the current move location //
      ReplayMoves(0, End:Int(GameExplorerSlider.intValue), Display:false)
   }

   // update the border layer //
   @IBAction func UpdateBorderLayer(_:AnyObject)
   {
      // get the current border texture from the border texture slider //
      BorderTexture = Int(BorderTextureSlider.intValue)

      // display the border texture for the chess board //
      BorderTextureView.stringValue = "Border texture is \(BorderTexture)"

      // initialize the border image //
      BoardLayers[1] = NSImage(size:NSSize(width:660, height:660))

      // lock the chess board border image focus //
      BoardLayers[1].lockFocus()

      // set the chess board color fill //
      BorderColors[BorderTexture].setFill()

      // draw the rectangle //
      NSBezierPath(rect:self.bounds).fill()

      // unlock the chess board border image focus //
      BoardLayers[1].unlockFocus()

      // recomposite the board for the updated border layer //
      RecompositeBoard(1, Span:1, Width:660)
   }

   // update the clock background visibility //
   @IBAction func UpdateClockBackgroundVisibility(_:AnyObject)
   {
      // update the clock background visibility //
      ClockBackgroundVisibility = Int(ClockBackgroundVisibilitySlider.intValue)

      // display the clock background visibility //
      DisplayClockBackgroundVisibility()

      // update the game clock background visibility //
      GameClock.UpdateClockBackgroundVisibility(ClockBackgroundVisibility)

      // update the game clock preview background visibility //
      GameClockPreview.UpdateClockBackgroundVisibility(ClockBackgroundVisibility)

      // update the game clock preview //
      GameClockPreview.UpdateClock(0, ElapsedBlackTime:0)
   }

   // update the board roughness //
   @IBAction func UpdateBoardRoughness(_:AnyObject)
   {
      // get the current board roughness from the board roughness slider //
      BoardRoughness = Int(BoardRoughnessSlider.intValue)

      // update the board roughness layer opacity //
      BoardLayerOpacities[7] = 0.15 * Float(BoardRoughness) / 100.0

      // update the board roughness view //
      BoardRoughnessView.stringValue = "Board roughness opacity is " +
                                       "\(Float(BoardRoughness) / 100.0)"

      // recomposite the board for the updated board roughness layer //
      RecompositeBoard(7, Span:1, Width:660)
   }

   // update the board tint layer //
   @IBAction func UpdateBoardTintLayer(_:AnyObject)
   {
      // get the board tint color from the board tint color slider //
      BoardTintColor = Int(BoardTintColorSlider.intValue)

      // create the board tint layer image //
      BoardLayers[5] = NSImage(size:NSSize(width:660, height:660))

      // lock the board tint color image focus //
      BoardLayers[5].lockFocus()

      // set the board tint color //
      BoardTintColors[BoardTintColor].setFill()

      // fill the rectangle with the background color //
      NSBezierPath(rect:NSMakeRect(30, 30, 600, 600)).fill()

      // unlock the board tint color image focus //
      BoardLayers[5].unlockFocus()

      // update the board tint color //
      BoardTintColorView.stringValue = "Board tint color is \(BoardTintColor)"

      // recomposite the board for the updated board tint layer //
      RecompositeBoard(5, Span:1, Width:660)
   }

   // update the square border //
   @IBAction func UpdateSquareBorder(_:AnyObject)
   {
      // get the current square border from the square border slider //
      SquareBorder = Int(SquareBorderSlider.intValue)

      // update the square border layer opacity //
      BoardLayerOpacities[8] = Float(SquareBorder) / 200.0

      // update the square border view //
      SquareBorderView.stringValue = "Square border opacity is \(Float(SquareBorder) / 100.0)"

      // recomposite the board for the updated square border layer //
      RecompositeBoard(8, Span:1, Width:660)
   }

   // update the board tint visibility //
   @IBAction func UpdateBoardTintVisibility(_:AnyObject)
   {
      // get the board tint visibility from the board tint visibility slider //
      BoardTintVisibility = Int(BoardTintVisibilitySlider.intValue)

      // update the board tint layer opacity //
      BoardLayerOpacities[5] = Float(BoardTintVisibility) / 100.0

      // update the board tint overlay opacity //
      BoardLayerOpacities[6] = BoardLayerOpacities[5] / 2.0

      // update the board tint visibility view //
      BoardTintVisibilityView.stringValue = "Board tint opacity is \(BoardLayerOpacities[5])"

      // recomposite the board for the updated board tint and tint overlay layers //
      RecompositeBoard(5, Span:2, Width:660)
   }

   // update the border visibility //
   @IBAction func UpdateBorderVisibility(_:AnyObject)
   {
      // get the current border visibility from the border visibility slider //
      BorderVisibility = Int(BorderVisibilitySlider.intValue)

      // update the border layer opacity //
      BoardLayerOpacities[1] = Float(BorderVisibility) / 100.0

      // update the border overlay opacity //
      BoardLayerOpacities[2] = BoardLayerOpacities[1]

      // display the border visibility for the chess board //
      BorderVisibilityView.stringValue = "Border opacity is \(BoardLayerOpacities[1])"

      // recomposite the board for the updated border and border overlay layers //
      RecompositeBoard(1, Span:2, Width:660)
   }
```

```
    // update the window texture opacity //
    @IBAction func UpdateWindowTextureOpacity(_:AnyObject)
    {
        // get the current window texture opacity from the window texture opacity slider //
        WindowTextureOpacity = Int(WindowTextureOpacitySlider.intValue)

        // display the window texture opacity //
        DisplayWindowTextureOpacity()

        // set the window background color from the current window color //
        SetWindowBackgroundColor(CurrentWindowColor)

        // recomposite the board for the updated color layer //
        RecompositeBoard(0, Span:1, Width:660)
    }

    // update the board animation rate //
    @IBAction func UpdateAnimationRate(_:AnyObject)
    {
        // get the current board animation rate from the board animation rate slider //
        AnimationRate = Int(AnimationRateSlider.intValue)

        // display the board animation rate //
        DisplayAnimationRate()
    }

    // update the gallery and grid animation rate //
    @IBAction func UpdateGalleryGridAnimationRate(_:AnyObject)
    {
        // get the current gallery and grid animation rate from the animation rate slider //
        GalleryGridAnimationRate = Int(GalleryGridAnimationRateSlider.intValue)

        // display the gallery and grid animation rate //
        DisplayGalleryGridAnimationRate()

        // set the game gallery view animation rate //
        GameGalleryView.SetAnimationRate(GalleryGridAnimationRate)

        // set the game grid view animation rate //
        GameGridView.SetAnimationRate(GalleryGridAnimationRate)
    }

    // update the uci engine difficulty //
    @IBAction func UpdateUCIEngineDifficulty(_:AnyObject)
    {
        // get the current uci engine difficulty from the uci engine difficulty slider //
        UCIEngineDifficulty = Int(UCIEngineDifficultySlider.intValue)

        // display the uci engine difficulty //
        DisplayUCIEngineDifficulty()
    }

    // update the uci engine hash //
    @IBAction func UpdateUCIEngineHash(_:AnyObject)
    {
        // get the current uci engine hash from the uci engine hash slider //
        UCIEngineHash = Int(UCIEngineHashSlider.intValue)

        // display the uci engine hash //
        DisplayUCIEngineHash()
    }

    // update the uci engine riskiness //
    @IBAction func UpdateUCIEngineRiskiness(_:AnyObject)
    {
        // get the current uci engine riskiness from the uci engine riskiness slider //
        UCIEngineRiskiness = Int(UCIEngineRiskinessSlider.intValue)

        // display the uci engine riskiness //
        DisplayUCIEngineRiskiness()
    }

    // update the game tree search time //
    @IBAction func UpdateSearchTime(_:AnyObject)
    {
        // get the current game tree search time from the search time slider //
        SearchTime = Int(SearchTimeSlider.intValue)

        // display the current search time //
        DisplaySearchTime()
    }

    // display the current search time to search down the game tree for //
    func DisplaySearchTime()
    {
        // display the current search time to search down the game tree for //
        SearchTimeView.stringValue = "Engine search time is \(SearchTime) seconds"
    }

    // display the window texture opacity //
    func DisplayWindowTextureOpacity()
    {
        // display the current window texture opacity //
        WindowTextureOpacityView.stringValue = "Window texture opacity is " +
                                               "\(Float(WindowTextureOpacity) * 0.01)"
    }

    // display the clock background visibility //
    func DisplayClockBackgroundVisibility()
    {
        // display the clock background visibility //
        ClockBackgroundOpacityView.stringValue = "Clock background opacity is " +
                                                 "\(Float(ClockBackgroundVisibility) * 0.01)"
    }

    // display the animation rate for the pieces moving //
    func DisplayAnimationRate()
    {
        // display the animation rate for the pieces moving //
        AnimationRateView.stringValue = "Board animation speed is \(AnimationRate)"
    }

    // display the gallery and grid animation rate for the frames moving //
    func DisplayGalleryGridAnimationRate()
    {
        // display the gallery and grid animation rate for the frames moving //
        GalleryGridAnimationRateView.stringValue = "Gallery and grid animation speed is " +
                                                   "\(GalleryGridAnimationRate)"
    }

    // display the uci engine difficulty //
    func DisplayUCIEngineDifficulty()
    {
        // display the uci engine difficulty //
        UCIEngineDifficultyView.stringValue = "Difficulty is \(UCIEngineDifficulty)"
    }

    // display the uci engine hash //
    func DisplayUCIEngineHash()
    {
        // display the uci engine hash //
        UCIEngineHashView.stringValue = "Hash is \(UCIEngineHash) MB"
    }

    // display the uci engine riskiness //
    func DisplayUCIEngineRiskiness()
    {
        // display the uci engine riskiness //
        UCIEngineRiskinessView.stringValue = "Riskiness is \(UCIEngineRiskiness)"
    }

    // make the computer move or stop button //
    @IBAction func GoStopMenuItemAction(_:AnyObject)
    {
        // set that want to start the clock //
        StartClock = true

        // if playing an engine match //
        if PlayingEngineMatch
        {
            // turn off the engine match menu item //
            EngineMatchMenuItem.state = NSControl.StateValue(rawValue:0)

            // set that no longer playing an engine match //
            PlayingEngineMatch = false
        }

        // if the engine is currently thinking //
        if EngineThinking
        {
            // if playing against uci engine //
            if UCIEngineOpponent
            {
                // tell uci engine to stop //
                WriteToEngine("stop\n")
            }
            // not using the uci chess engine mode //
            else
            {
                // make the chess engine stop //
                EngineMakeEngineStop(Engine)
            }
        }
        // if the engine is currently not thinking //
        else
        {
            // start an engine move //
            StartEngineMove()
        }
    }

    // get the corresponding index value of a rank or file coordinate string //
    func RankOrFileIndex(_ Coordinate:Character) -> Int
    {
        // if coordinate is the first file or first rank //
        if Coordinate == "a" || Coordinate  == "1"
        {
            // return an index value of zero //
            return 0
        }
        // if coordinate is the second file or second rank //
        else if Coordinate == "b" || Coordinate == "2"
        {
            // return an index value of one //
            return 1
        }
        // if coordinate is the third file or third rank //
        else if Coordinate == "c" || Coordinate == "3"
        {
            // return an index value of two //
            return 2
        }
        // if coordinate is the fourth file or fourth rank //
        else if Coordinate == "d" || Coordinate == "4"
        {
            // return an index value of three //
            return 3
        }
        // if coordinate is the fifth file or fifth rank //
        else if Coordinate == "e" || Coordinate == "5"
        {
            // return an index value of four //
            return 4
        }
        // if coordinate is the sixth file or sixth rank //
        else if Coordinate == "f" || Coordinate == "6"
        {
            // return an index value of five //
            return 5
        }
        // if coordinate is the seventh file or seventh rank //
        else if Coordinate == "g" || Coordinate == "7"
        {
            // return an index value of six //
            return 6
        }
        // if coordinate is the eighth file or eighth rank //
        else if Coordinate == "h" || Coordinate == "8"
        {
            // return an index value of seven //
            return 7
        }
        // default value //
        else
        {
            // return a default index of -1 //
            return -1
        }
    }

    // perform a move of a piece on the board //
    func PerformEngineMove()
    {
        // stop the progress animation //
        AnimateProgress = false

        // set that there is no moving piece to currently animate //
        MovingPiece = 0

        // if in the uci chess engine mode //
        if UCIEngineOpponent
        {
            // get the best move from engine in uci notation //
            UCIMove = EngineMessageTokens[1]

            // add the current move to the move list //
            MoveList.append(UCIMove)

            // add a placeholder annotation to the annotation list //
            AnnotationList.append("")

            // store the autosave data //
            StoreAutosaveData("")
        }

        // convert the uci move to start and end squares //
        UCIToSquares()

        // save the previous move list //
        PreviousMoveList = MoveList

        // save the previous annotation list //
        PreviousAnnotationList = AnnotationList

        // animate the piece moving from the start square to the end square //
        RunCoreAnimation()
    }

    // get the image associated with this piece //
    func GetPieceImage(_ Piece:Int) -> NSImage
    {
        // if piece is white //
        if Piece > 0
        {
            // set the chess piece image index corresponding to piece and chess piece set //
            ChessPieceImageIndex = Piece - 1 + ChessPieceSetType * 12
        }
        // piece is black //
        else
        {
            // set the chess piece image index corresponding to piece and chess piece set //
            ChessPieceImageIndex = abs(Piece) + 5 + ChessPieceSetType * 12
        }

        // return the correct image for the specified chess piece //
        return ChessPieceImages[ChessPieceImageIndex]
    }

    // keep the mouse location within the bounds of the chessboard //
    func KeepInBounds()
    {
        // if mouse is to the left of the board //
        if Int(MouseLocation.x) < SquareOffset
        {
            // reset x coordinate of mouse location //
            MouseLocation.x = CGFloat(SquareOffset)
        }

        // if mouse is below the board //
        if Int(MouseLocation.y) < SquareOffset
        {
            // reset y coordinate of mouse location //
            MouseLocation.y = CGFloat(SquareOffset)
        }

        // if mouse is to the right of the board //
        if Int(MouseLocation.x) > 593
        {
            // reset x coordinate of mouse location //
            MouseLocation.x = 593
        }

        // if mouse is above the board //
        if Int(MouseLocation.y) > 593
        {
            // reset y coordinate of mouse location //
            MouseLocation.y = 593
        }
    }

    // see if there is engine data ready to process //
    @objc func EngineDataReady(_ TheNotification:Notification)
    {
        // convert the data read to string format //
        EngineRawData = String(data:(TheNotification.userInfo! as NSDictionary).value(forKey:
                               "NSFileHandleNotificationDataItem") as! Data, encoding:String
                               .Encoding(rawValue:String.Encoding.utf8.rawValue))!

        // get the engine messages from the raw engine data //
        EngineMessages = EngineRawData.components(separatedBy:"\n")

        // loop over all lines in the engine messages //
        for Line in 0 ..< EngineMessages.count
        {
            // get one line of data sent from the uci chess engine //
            EngMsg = EngineMessages[Line]

            // if there are at least six characters in the engine message line //
            if EngMsg.count >= 6
            {
                // if the engine message line doesn't contain general information //
                if EngMsg.prefix(4) != "info"
                {
                    // if the engine message line contains the name of the chess engine //
                    if EngMsg.prefix(7) == "id name"
                    {
                        // set the name of the chess engine //
                        EngineName = String(EngMsg.suffix(EngMsg.count - 8))

                        // set that the uci chess engine loaded successfully //
                        UCIEngineLoaded = true

                        // tell uci engine to start a new game //
                        WriteToEngine("ucinewgame\n")

                        // update menu item name for uci engine //
                        UCIEngineOpponentMenuItem.title = EngineName
                    }
                    // if the engine message line contains the author of the chess engine //
                    else if EngMsg.prefix(9) == "id author"
                    {
                        // add the uci chess engine author name to the uci engine engine name //
                        EngineName += " by " + EngMsg.suffix(EngMsg.count - 10)

                        // update the menu item name for engine //
                        UCIEngineOpponentMenuItem.title = EngineName

                        // check if the uci engine loaded //
                        CheckUCIEngineLoaded()
                    }
                    // message line contains relevant data to be processed //
                    else
                    {
                        // tokenize the engine message from one of the engine messages lines //
                        EngineMessageTokens = EngMsg.components(separatedBy:" ")

                        // if have the bestmove engine message from engine message tokens //
                        if EngineMessageTokens[0] == "bestmove"
                        {
                            // perform an engine move on the main thread //
                            DispatchQueue.main.async
                            {
                                // perform the engine move //
                                self.PerformEngineMove()
                            }
                        }
                    }
                }
                // if have incoming info and want to show the stockfish analysis //
                else if EngMsg.prefix(4) == "info" && ShowStockfishAnalysis
                {
                    // if the principal variation line is in this string //
                    if EngMsg.range(of:" pv ") != nil
                    {
                        // get the stockfish principal variation //
                        StockfishPV = String(EngMsg.suffix(from:EngMsg.range(of:" pv ")?.upperBound)))
                    }
                }
            }
        }

        // notify when the chess engine has data to send to standard output //
        EngineFileHandleRead.readInBackgroundAndNotify()
    }

    // set whether opponent is a human, greenchess engine, or uci chess engine //
    func Opponent(_ Human:Bool, Engine:Bool, UCIEngine:Bool)
    {
        // if playing against a human //
        if Human
        {
            // set opponent icon to be none //
            OpponentToolbarItem.image = HumanOpponentIcon
        }
        // if playing against the greenchess engine //
        else if Engine
        {
            // set opponent icon to be the greenchess engine //
            OpponentToolbarItem.image = GreenChessOpponentIcon
        }
        // if playing against a uci chess engine //
        else
        {
            // set opponent icon to be a uci chess engine //
            OpponentToolbarItem.image = UCIEngineOpponentIcon
        }

        // set whether playing against a human opponent or not //
        HumanOpponent = Human

        // set whether playing against the greenchess engine or not //
        EngineOpponent = Engine

        // set whether playing against uci chess engine or not //
        UCIEngineOpponent = UCIEngine

        // update the state of the human opponent menu item //
        HumanOpponentMenuItem.state = NSControl.StateValue(rawValue:BoolToInt(HumanOpponent))

        // update the state of the greenchess engine menu item //
        EngineOpponentMenuItem.state = NSControl.StateValue(rawValue:BoolToInt(EngineOpponent))

        // update the state of the uci chess engine menu item //
        UCIEngineOpponentMenuItem.state = NSControl.StateValue(rawValue:BoolToInt(UCIEngineOpponent))

        // hide the engine match menu item if don't have uci engine loaded //
        EngineMatchMenuItem.isHidden = !UCIEngineLoaded
    }

    // update the pgn label pieces to symbol representations //
    func UpdatePGNPiecesToSymbols(_ MoveIndex:Int)
    {
        // if the pgn label is empty //
        if PGNLabel.isEmpty
        {
            // exit function since can't modify an empty string //
            return
        }

        // if currently a white move //
        if (MoveIndex + MoveOffset) % 2 == 0
        {
            // replace white king with symbol representation for white king //
            PGNLabel = PGNLabel.replacingOccurrences(of:"K", with:"♔")

            // replace white queen with symbol representation for white queen //
            PGNLabel = PGNLabel.replacingOccurrences(of:"Q", with:"♕")

            // replace white rook with symbol representation for white rook //
            PGNLabel = PGNLabel.replacingOccurrences(of:"R", with:"♖")

            // replace white bishop with symbol representation for white bishop //
            PGNLabel = PGNLabel.replacingOccurrences(of:"B", with:"♗")

            // replace white knight with symbol representation for white knight //
            PGNLabel = PGNLabel.replacingOccurrences(of:"N", with:"♘")

            // if first letter of pgn label is a file and is a pawn move //
            if PGNLabel.first! >= "a" && PGNLabel.first! <= "h"
            {
                // put a white pawn symbol at the front of the pgn label //
                PGNLabel = "♙\(PGNLabel)"
            }
        }
        // currently a black move //
        else
        {
            // replace black king with symbol representation for black king //
            PGNLabel = PGNLabel.replacingOccurrences(of:"K", with:"♚")

            // replace black queen with symbol representation for black queen //
            PGNLabel = PGNLabel.replacingOccurrences(of:"Q", with:"♛")

            // replace black rook with symbol representation for black rook //
            PGNLabel = PGNLabel.replacingOccurrences(of:"R", with:"♜")

            // replace black bishop with symbol representation for black bishop //
            PGNLabel = PGNLabel.replacingOccurrences(of:"B", with:"♝")

            // replace black knight with symbol representation for black knight //
            PGNLabel = PGNLabel.replacingOccurrences(of:"N", with:"♞")

            // if first letter of pgn label is a file and is a pawn move //
            if PGNLabel.first! >= "a" && PGNLabel.first! <= "h"
            {
                // put a black pawn symbol at the front of the pgn label //
                PGNLabel = "♟\(PGNLabel)"
            }
        }
    }

    // append a tag to the game info list //
    func AppendTag(_ Description:String, Tag:String)
    {
        // if the tag is not empty //
        if !Description.isEmpty
        {
            // append the tag to the game info list //
            GameInfoList.append("\(Tag):  \(Description)")
        }
    }

    // print the game information to the gui //
    func PrintGameInfo()
    {
        // if in game or database explorer //
        if GameExplorerWindowOpen || InDatabaseExplorer
        {
            // let the end of the pgn list be the current ply count //
            ListEnd = PlyCount
        }
        // if not currently reviewing a game or in database explorer //
        else
        {
            // let the end of the pgn list be the total move count //
            ListEnd = MoveList.count
        }

        // initialize stockfish analysis to display to false //
        StockfishAnalysisToDisplay = false

        // initialize the pgn moves list array //
        PGNMovesList.removeAll()

        // initialize the pgn moves list array for the annotations //
        PGNMovesListForAnnotator.removeAll()

        // clear the game info list //
        GameInfoList.removeAll()

        // if from the fen notation, black is to move first //
        if Attributes.FENColor == "b"
        {
            // set the move offset to one //
            MoveOffset = 1
        }
        // from the fen notation, white is to move first //
        else
        {
            // set the move offset to zero //
            MoveOffset = 0
        }

        // if have checkmate and white to move next //
        if BoardIsCheckMate(Board) && BoardIsWhite(Board)
        {
            // set current game state to show that black won //
            CurrentGameState = 1

            // append that game ended in checkmate and that black won //
            GameInfoList.append("Checkmate and Black wins")
        }
        // if have checkmate and black to move next //
        else if BoardIsCheckMate(Board) && BoardIsBlack(Board)
        {
            // set current game state to show that white won //
            CurrentGameState = 2

            // append that game ended in checkmate and that white won //
            GameInfoList.append("Checkmate and White wins")
        }
        // if have stalemate //
        else if BoardIsStalemate(Board)
        {
            // set current game state to show that had a stalemate //
            CurrentGameState = 3

            // append that game ended in stalemate and that game is over //
            GameInfoList.append("Stalemate")
        }
        // if have threefold repetition //
        else if BoardIsThreefoldRepetition(Board)
        {
            // set current game state to show that detected draw by threefold repetition //
            CurrentGameState = 4

            // append that draw detected by threefold repetition //
            GameInfoList.append("Draw detected by threefold repetition")
        }
        // if have insufficient material to checkmate with //
        else if BoardInsufficientMaterialToMate(Board)
        {
            // set current game state to show that detected draw by insufficient material //
            CurrentGameState = 7

            // append that draw detected by insufficient material to checkmate //
            GameInfoList.append("Draw detected by insufficient material for mate")
        }
        // game has not ended yet //
        else
        {
            // set the current game state to show that game not over yet //
            CurrentGameState = 0

            // append the current move number to the game info //
            GameInfoList.append(String(format:"Move %d", (ListEnd + MoveOffset) / 2 +
                                Int(Attributes.FENStartMoveNumber)!))

            // if white is currently to move //
            if BoardIsWhite(Board)
            {
                // append that white is to move //
                GameInfoList[0] += " and White to move"
            }
            // if black is currently to move //
            else
            {
                // append that black is to move //
                GameInfoList[0] += " and Black to move"
            }

            // if white is in check and to move or if black in check and to move //
            if BoardIsInCheck(Board)
            {
                // append that the color to move is currently in check //
                GameInfoList[0] += " out of check"
            }
        }

        // if user entered in the event title //
        AppendTag(Attributes.EventTitle, Tag:"Event")

        // if user entered in the game site location //
        AppendTag(Attributes.SiteLocation, Tag:"Site")

        // append the game date to the game info list //
        AppendTag(Attributes.DateGamePlayed, Tag:"Date")

        // if user entered in the game round number //
        AppendTag(Attributes.RoundNumber, Tag:"Round")

        // if user entered in name of white player //
        AppendTag(Attributes.WhiteName, Tag:"White")

        // if user entered in name of black player //
        AppendTag(Attributes.BlackName, Tag:"Black")

        // if user entered in the end result of the game //
        AppendTag(Attributes.EndResult, Tag:"Result")

        // if user entered in the time control //
        AppendTag(Attributes.TimeControl, Tag:"Time Control")

        // if user entered in the white elo //
        AppendTag(Attributes.WhiteElo, Tag:"White Elo")

        // if user entered in the black elo //
        AppendTag(Attributes.BlackElo, Tag:"Black Elo")

        // if user entered in the white uscf //
        AppendTag(Attributes.WhiteUSCF, Tag:"White USCF")

        // if user entered in the black uscf //
        AppendTag(Attributes.BlackUSCF, Tag:"Black USCF")

        // if user entered in the annotator //
        AppendTag(Attributes.Annotator, Tag:"Annotator")

        // if user entered in the eco //
        AppendTag(Attributes.ECO, Tag:"ECO")

        // if user entered in the opening //
        AppendTag(Attributes.Opening, Tag:"Opening")

        // if user entered in the section //
        AppendTag(Attributes.Section, Tag:"Section")

        // if user entered in the board number //
        AppendTag(Attributes.BoardNumber, Tag:"Board")

        // display game information to gui //
        UpdateGameInfoList()

        // initialize the game data board //
        BoardInitializeAttributes(GameDataBoard, Attributes.FENLayout, Attributes.FENCastlingRights,
                                  Attributes.FENColor, Attributes.FENEnpassantSquare)

        // if want to show the stockfish analysis //
        if ShowStockfishAnalysis
        {
            // parse the stockfish pv line //
            RawStockfishMoveList = StockfishPV.components(separatedBy:" ")

            // remove empty stockfish analysis move list elements //
            RemoveEmptyStockfishMoveListElements()
        }

        // loop over move list //
        for MoveIndex in 0 ..< ListEnd + StockfishMoveList.count
        {
            // if in the stockfish analysis move list //
            if MoveIndex >= ListEnd
            {
                // make the stockfish pv move on the data board //
                GCMakeMoveOnDataBoard(StockfishMoveList[MoveIndex - ListEnd])
            }
            // not in the stockfish analysis move list //
            else
            {
                // make the move on the data board //
                GCMakeMoveOnDataBoard(MoveList[MoveIndex])
            }

            // if the move exists and was a valid move //
            if MoveExists
            {
                // get the pgn label for the move //
                PGNLabel = String(validatingUTF8:BoardPGNLabel(GameDataBoard))!

                // update the pgn pieces to piece symbols //
                UpdatePGNPiecesToSymbols(MoveIndex)

                // set the pgn move number //
                PGNMove = String(format:"%d.", (MoveIndex + MoveOffset) / 2 + Int(Attributes
                                 .FENStartMoveNumber)!)

                // if black is moving //
                if (MoveIndex + MoveOffset) % 2 == 1
                {
                    // add .. to pgn move for black //
                    PGNMove += ".."
                }

                // add the pgn label to the pgn move //
                PGNMove += " \(PGNLabel)"

                // if not in the stockfish analysis move list //
                if MoveIndex < ListEnd
                {
                    // get the current move annotation string //
                    AnnotationString = AnnotationList[MoveIndex]

                    // convert all the numeric annotation glyphs //
                    ConvertNumericAnnotationGlyphs()

                    // append the pgn move to the pgn moves list for the annotator //
                    PGNMovesListForAnnotator.append(PGNMove)

                    // add the corresponding annotation to the pgn move string //
                    PGNMove += " " + AnnotationString

                    // add the pgn move string to the pgn move list array //
                    PGNMovesList.append(PGNMove)
                }
                // if not at the beginning of a new white/black move sequence or at end //
                else
                {
                    // add the pgn move to the stockfish analysis string //
                    StockfishAnalysisString += "(" + PGNMove + ") "
                }
            }
            // the move does not exist //
            else
            {
                // if in the stockfish analysis move list portion //
                if MoveIndex >= ListEnd
                {
                    // exit the loop since the pv line is missing a move //
                    break
                }

                // set the pgn move string to invalid move //
                PGNMove = "Invalid Move"

                // add the pgn move string to the pgn move list array //
                PGNMovesList.append(PGNMove)
            }
        }

        // if have stockfish analysis and not playing an opening move in timed game mode //
        if !StockfishAnalysisString.isEmpty && !(PlayingTimedGame && StockfishAnalysisString ==
           "Opening Book Move") && ShowStockfishAnalysis
        {
            // set stockfish analysis to display to true //
            StockfishAnalysisToDisplay = true

            // append the stockfish analysis string to the pgn moves list //
            PGNMovesList.append(StockfishAnalysisString)
        }

        // clear the stockfish pv line //
        StockfishPV = ""

        // clear the stockfish analysis string //
        StockfishAnalysisString = ""

        // clear the raw stockfish analysis move list //
        RawStockfishMoveList.removeAll()

        // clear the stockfish analysis move list //
        StockfishMoveList.removeAll()

        // update the move list //
        UpdateMoveList()
    }

    // update the game list //
    func UpdateTheGameList()
    {
        // refresh the game list //
        GameListData.RefreshGameList(FontColors(GameListColor), TheFont:GetFont(GameListFontType,
                                     Size:CGFloat(14 + GameListSizes), Bold:GameListBold))

        // reload the game list data //
        GameListTableView.reloadData()

        // select the previously selected row //
        GameListTableView.selectRowIndexes(IndexSet(integer:PreviouslySelectedRow),
                                           byExtendingSelection:false)

        // scroll to the previously selected row //
        GameListTableView.scrollRowToVisible(PreviouslySelectedRow)

        // store the selected row to the game data //
        GameListData.StoreSelectedRow(self.GameListTableView.selectedRow)
    }

    // update the game info list //
    func UpdateGameInfoList()
    {
        // refresh the game info list //
        GameInfoData.RefreshGameInfoList(GameInfoList, Color:FontColors(InfoColor), Font:
                                         GetFont(GameInfoFontType, Size:CGFloat(14 + GameInfoSizes),
                                         Bold:GameInfoBold))

        // reload the game info to the game info table view //
        GameInfoTableView.reloadData()

        // if game info list not empty //
        if !GameInfoList.isEmpty
        {
            // scroll the game info table view to the first move made //
            GameInfoTableView.scrollRowToVisible(0)
        }
    }

    // update the move list //
    func UpdateMoveList()
    {
        // refresh the pgn move list //
        PGNMovesData.RefreshPGNMoveList(PGNMovesList, Color:FontColors(MoveListColor), Font:
                                        GetFont(MoveListFontType, Size:CGFloat(14 + MoveListSizes),
                                        Bold:MoveListBold))

        // if the game explorer, database explorer, or preferences window is attached //
        if GameExplorerWindowOpen || InDatabaseExplorer || EnteringPreferences
        {
            // set the max pgn moves data rows to be 6 //
            PGNMovesData.MaxRows = 6

            // hide the vertical scroller in the game info table view //
            GameInfoTableView.enclosingScrollView?.hasVerticalScroller = false

            // hide the vertical scroller in the pgn moves table view //
            PGNMovesTableView.enclosingScrollView?.hasVerticalScroller = false
        }
        // no attached game explorer, database explorer, or preferences window //
        else
        {
            // set the max pgn moves data rows to be 10000 //
            PGNMovesData.MaxRows = 10000

            // unhide the vertical scroller in the game info table view //
            GameInfoTableView.enclosingScrollView?.hasVerticalScroller = true

            // unhide the vertical scroller in the pgn moves table view //
            PGNMovesTableView.enclosingScrollView?.hasVerticalScroller = true
        }

        // reload the game data to the pgn moves table view //
        PGNMovesTableView.reloadData()

        // if move list not empty //
        if !PGNMovesList.isEmpty
        {
            // scroll the pgn moves table view to the last move made //
            PGNMovesTableView.scrollRowToVisible(min(PGNMovesList.count * 2, PGNMovesData.MaxRows) -
                                                 1)
        }
    }

    // write command to uci chess engine //
    func WriteToEngine(_ Command:String)
    {
        // format data to write to chess engine //
        EngineDataToWrite = Command.data(using:String.Encoding.ascii)!

        // write the data to chess engine //
        EngineFileHandleWrite.write(EngineDataToWrite)

        // notify when chess engine has data to send to standard output //
        EngineFileHandleRead.readInBackgroundAndNotify()
    }

    // if have pawn promotion, extract the piece type the pawn is promoting to //
    func ExtractPromotionTypeIfNeeded()
    {
        // if the UCI move consists of a pawn promotion //
        if UCIMove.count == 5
        {
            // if the promotion portion of the UCI move is a queen //
            if UCIMove.last! == "q"
            {
                // set that pawn is promoting to a queen //
                PromotionType = 5
            }
            // if the promotion portion of the UCI move is a rook //
            else if UCIMove.last! == "r"
            {
                // set that pawn is promoting to a rook //
                PromotionType = 4
            }
            // if the promotion portion of the UCI move is a bishop //
            else if UCIMove.last! == "b"
            {
                // set that pawn is promoting to a bishop //
                PromotionType = 3
            }
            // if the promotion portion of the UCI move is a knight //
            else if UCIMove.last! == "n"
            {
                // set that pawn is promoting to a knight //
                PromotionType = 2
            }
        }
        // UCI move does not consist of a pawn promotion //
        else
        {
            // set promotion type to a null piece //
```

```
        PromotionType = 0
    }

    // if black is currently to move //
    if BoardIsWhite(Board)
    {
        // negate the type of the piece moving to represent a black piece //
        PromotionType = -PromotionType
    }
}

// convert a UCI move to start and end squares //
func UCIToSquares()
{
    // the starting file of the piece that moves //
    StartSquare.x = CGFloat(RankOrFileIndex(UCIMove.prefix(1).last!))

    // the starting rank of the piece that moves //
    StartSquare.y = CGFloat(RankOrFileIndex(UCIMove.prefix(2).last!))

    // the ending file of the piece that moves //
    EndSquare.x = CGFloat(RankOrFileIndex(UCIMove.prefix(3).last!))

    // the ending rank of the piece that moves //
    EndSquare.y = CGFloat(RankOrFileIndex(UCIMove.prefix(4).last!))
}

// convert the UCI move to origin and destination squares //
func ProcessUCIMove()
{
    // if have pawn promotion, extract the piece type the pawn is promoting to //
    ExtractPromotionTypeIfNeeded()

    // convert the UCI move to start and end squares //
    UCIToSquares()

    // get the piece that is going to be animated //
    MovingPiece = Layout[Int(StartSquare.x)][Int(StartSquare.y)]

    // remove the piece from the original square //
    Layout[Int(StartSquare.x)][Int(StartSquare.y)] = 0
}

// swap the color to move next //
@IBAction func SwapColorToMoveButtonAction(_:AnyObject)
{
    // play the game sound of a click //
    PlayGameSoundThread("click")

    // if the fen color is currently white //
    if Attributes.FENColor == "w"
    {
        // set the fen color to black //
        Attributes.FENColor = "b"
    }
    // the fen color is currently black //
    else
    {
        // set the fen color to white //
        Attributes.FENColor = "w"
    }

    // update the position editor toolbar items //
    UpdatePositionEditorToolbarButtons()
}

// convert an integer to a boolean //
func IntToBool(_ IntVar:Int) -> Bool
{
    // if integer is zero //
    if IntVar == 0
    {
        // return false //
        return false
    }
    // integer is not zero //
    else
    {
        // return true //
        return true
    }
}

// convert a boolean to an integer //
func BoolToInt(_ BoolVar:Bool) -> Int
{
    // if boolean is true //
    if BoolVar
    {
        // return one //
        return 1
    }
    // boolean is false //
    else
    {
        // return zero //
        return 0
    }
}

// set the diagram cursor //
func SetDiagramCursor(_ MarkerColor:String)
{
    // set the diagram cursor //
    DiagramCursor = NSCursor.init(image:NSImage(named:NSImage.Name(rawValue:
                              "marker_\(MarkerColor)_icon"))!, hotSpot:NSPoint(x:2, y:30))
}

// set the position editor cursor //
func SetPosCursor(_ Type:Int)
{
    // set the position editor moving piece to chosen type //
    PositionMovingPiece = Type

    // if plan to erase a piece //
    if Type == 0
    {
        // set the position cursor to the eraser icon //
        PosCursor = NSCursor.init(image:Eraser, hotSpot:NSPoint(x:37.5, y:37.5))
    }
    // if plan on adding an enpassant square //
    else if Type == 10
    {
        // set the position cursor to the en passant icon //
        PosCursor = NSCursor.init(image:EnpassantHighlight, hotSpot:NSPoint(x:37.5, y:37.5))
    }
    // if plan on adding a piece to the board //
    else
    {
        // set the position cursor to the chosen piece type //
        PosCursor = NSCursor.init(image:GetPieceImage(Type), hotSpot:NSPoint(x:37.5, y:37.5))
    }
}

// the position editor eraser action //
@IBAction func PositionEditorEraserAction(_:AnyObject)
{
    // play the game sound of a click //
    PlayGameSoundThread("click")

    // set the cursor image to an eraser //
    SetPosCursor(0)
}

// the position editor en passant square action //
@IBAction func PositionEditorEPAction(_:AnyObject)
{
    // play the game sound of a click //
    PlayGameSoundThread("click")

    // set the cursor image to an en passant square //
    SetPosCursor(10)
}

// the position editor second menu action //
@IBAction func PositionEditorSecondMenuAction(_:AnyObject)
{
    // play the game sound of a click //
    PlayGameSoundThread("click")

    // if in the second position editor menu //
    if SecondPositionMenu
    {
        // set that no longer in the second position editor menu //
        SecondPositionMenu = false

        // add the position editor piece items //
        AddPositionEditorPieceItems()
    }
    // in the first position editor menu //
    else
    {
        // set that in the second position editor menu //
        SecondPositionMenu = true

        // add the position editor castling items //
        AddPositionEditorCastlingItems()
    }
}

// the position editor white pawn action //
@IBAction func PositionEditorWhitePawnAction(_:AnyObject)
{
    // play the game sound of a click //
    PlayGameSoundThread("click")

    // set the cursor image to a white pawn //
    SetPosCursor(1)
}

// the position editor white knight action //
@IBAction func PositionEditorWhiteKnightAction(_:AnyObject)
{
    // play the game sound of a click //
    PlayGameSoundThread("click")

    // set the cursor image to a white knight //
    SetPosCursor(2)
}

// the position editor white bishop action //
@IBAction func PositionEditorWhiteBishopAction(_:AnyObject)
{
    // play the game sound of a click //
    PlayGameSoundThread("click")

    // set the cursor image to a white bishop //
    SetPosCursor(3)
}

// the position editor white rook action //
@IBAction func PositionEditorWhiteRookAction(_:AnyObject)
{
    // play the game sound of a click //
    PlayGameSoundThread("click")

    // set the cursor image to a white rook //
    SetPosCursor(4)
}

// the position editor white queen action //
@IBAction func PositionEditorWhiteQueenAction(_:AnyObject)
{
    // play the game sound of a click //
    PlayGameSoundThread("click")

    // set the cursor image to a white queen //
    SetPosCursor(5)
}

// the position editor white king action //
@IBAction func PositionEditorWhiteKingAction(_:AnyObject)
{
    // play the game sound of a click //
    PlayGameSoundThread("click")

    // set the cursor image to a white king //
    SetPosCursor(6)
}

// the position editor black pawn action //
@IBAction func PositionEditorBlackPawnAction(_:AnyObject)
{
    // play the game sound of a click //
    PlayGameSoundThread("click")

    // set the cursor image to a black pawn //
    SetPosCursor(-1)
}

// the position editor black knight action //
@IBAction func PositionEditorBlackKnightAction(_:AnyObject)
{
    // play the game sound of a click //
    PlayGameSoundThread("click")

    // set the cursor image to a black knight //
    SetPosCursor(-2)
}

// the position editor black bishop action //
@IBAction func PositionEditorBlackBishopAction(_:AnyObject)
{
    // play the game sound of a click //
    PlayGameSoundThread("click")

    // set the cursor image to a black bishop //
    SetPosCursor(-3)
}

// the position editor black rook action //
@IBAction func PositionEditorBlackRookAction(_:AnyObject)
{
    // play the game sound of a click //
    PlayGameSoundThread("click")

    // set the cursor image to a black rook //
    SetPosCursor(-4)
}

// the position editor black queen action //
@IBAction func PositionEditorBlackQueenAction(_:AnyObject)
{
    // play the game sound of a click //
    PlayGameSoundThread("click")

    // set the cursor image to a black queen //
    SetPosCursor(-5)
}

// the position editor black king action //
@IBAction func PositionEditorBlackKingAction(_:AnyObject)
{
    // play the game sound of a click //
    PlayGameSoundThread("click")

    // set the cursor image to a black king //
    SetPosCursor(-6)
}

// give or take away the right to kingside castle for white //
@IBAction func WhiteKingsideCastleButtonAction(_:AnyObject)
{
    // play the game sound of a click //
    PlayGameSoundThread("click")

    // change the kingside castling right for white //
    WhiteKingsideCastle = !WhiteKingsideCastle

    // update the position editor toolbar buttons //
    UpdatePositionEditorToolbarButtons()
}

// give or take away the right to queenside castle for white //
@IBAction func WhiteQueensideCastleButtonAction(_:AnyObject)
{
    // play the game sound of a click //
    PlayGameSoundThread("click")

    // change the queenside castling right for white //
    WhiteQueensideCastle = !WhiteQueensideCastle

    // update the position editor toolbar buttons //
    UpdatePositionEditorToolbarButtons()
}

// give or take away the right to kingside castle for black //
@IBAction func BlackKingsideCastleButtonAction(_:AnyObject)
{
    // play the game sound of a click //
    PlayGameSoundThread("click")

    // change the kingside castling right for black //
    BlackKingsideCastle = !BlackKingsideCastle

    // update the position editor toolbar buttons //
    UpdatePositionEditorToolbarButtons()
}

// give or take away the right to queenside castle for black //
@IBAction func BlackQueensideCastleButtonAction(_:AnyObject)
{
    // play the game sound of a click //
    PlayGameSoundThread("click")

    // change the queenside castling right for black //
    BlackQueensideCastle = !BlackQueensideCastle

    // update the position editor toolbar buttons //
    UpdatePositionEditorToolbarButtons()
}

// change state of whether white won the game or not //
@IBAction func WhiteWonButtonAction(_:AnyObject)
{
    // play the game sound of a click //
    PlayGameSoundThread("click")

    // turn the black won button off //
    BlackWonButton.state = NSControl.StateValue(rawValue:0)

    // turn the draw button off //
    DrawButton.state = NSControl.StateValue(rawValue:0)

    // change the state of whether white won the game or not //
    WhiteWonGame = !WhiteWonGame

    // set that black didn't win the game //
    BlackWonGame = false

    // set that game was not drawn //
    DrawnGame = false
}

// change state of whether black won the game or not //
@IBAction func BlackWonButtonAction(_:AnyObject)
{
    // play the game sound of a click //
    PlayGameSoundThread("click")

    // turn the white won button off //
    WhiteWonButton.state = NSControl.StateValue(rawValue:0)

    // turn the draw button off //
    DrawButton.state = NSControl.StateValue(rawValue:0)

    // set that white did not win the game //
    WhiteWonGame = false

    // change the state of whether black won the game or not //
    BlackWonGame = !BlackWonGame

    // set that the game was not drawn //
    DrawnGame = false
}

// highlight the last move made button action //
@IBAction func HighlightLastMoveMadeButtonAction(_:AnyObject)
{
    // play the sound of a button click //
    PlayGameSoundThread("click")

    // toggle the highlight last move made option //
    HighlightLastMoveMade = !HighlightLastMoveMade

    // update the board and captured pieces //
    UpdateBoardAndCapturedPieces()
}

// set the number of threads to use for the uci engine //
func SetThreadsToUse()
{
    // if multithreading the uci engine //
    if MultithreadUCIEngine
    {
        // set the number of threads to use to the total number of available threads //
        ThreadsToUse = TotalThreads
    }
    // not multithreading the uci engine //
    else
    {
        // set the number of threads to use to one since not multithreading //
        ThreadsToUse = 1
    }
}

// the multithread uci engine button action //
@IBAction func MultithreadUCIEngineButtonAction(_:AnyObject)
{
    // play the sound of a button click //
    PlayGameSoundThread("click")

    // toggle the multithread uci engine state //
    MultithreadUCIEngine = !MultithreadUCIEngine

    // set the number of threads to use for the uci engine //
    SetThreadsToUse()
}

// highlight the threatened pieces button action //
@IBAction func HighlightThreatenedPiecesButtonAction(_:AnyObject)
{
    // play the sound of a button click //
    PlayGameSoundThread("click")

    // toggle the highlight the threatened pieces option //
    HighlightThreatenedPieces = !HighlightThreatenedPieces

    // update the board and captured pieces //
    UpdateBoardAndCapturedPieces()
}

// highlight the captured pieces button action //
@IBAction func HighlightCapturedPiecesButtonAction(_:AnyObject)
{
    // play the sound of a button click //
    PlayGameSoundThread("click")

    // toggle the highlight the captured pieces button //
    HighlightCapturedPieces = !HighlightCapturedPieces

    // update the board and captured pieces //
    UpdateBoardAndCapturedPieces()
}

// the autosave game button action //
@IBAction func AutosaveGameButtonAction(_:AnyObject)
{
    // play the sound of a button click //
    PlayGameSoundThread("click")

    // switch the autosave game state //
    AutosaveGame = !AutosaveGame

    // store the autosave data //
    StoreAutosaveData("")
}

// activate or deactivate using the encyclopedia of chess openings //
@IBAction func EcoBookButtonAction(_:AnyObject)
{
    // play the sound of a button click //
    PlayGameSoundThread("click")

    // switch whether currently using the chess opening book or not //
    UseEcoBook = !UseEcoBook
}

// change the state of whether game was drawn or not //
@IBAction func DrawButtonAction(_:AnyObject)
{
    // play the game sound of a click //
    PlayGameSoundThread("click")

    // turn the white won button off //
    WhiteWonButton.state = NSControl.StateValue(rawValue:0)

    // turn the black won button off //
    BlackWonButton.state = NSControl.StateValue(rawValue:0)

    // set that white did not win the game //
    WhiteWonGame = false

    // set that black did not win the game //
    BlackWonGame = false

    // change the state of whether the game was drawn or not //
    DrawnGame = !DrawnGame
}

// go left in the game annotator //
@IBAction func AnnotatorLeftAction(_:AnyObject)
{
    // if the annotator slider value is not less than one //
    if GameAnnotatorSlider.intValue - 1 >= 1
    {
        // decrement the game annotator slider //
        GameAnnotatorSlider.intValue -= 1

        // update the game annotator //
        UpdateGameAnnotatorFromSlider(self)
    }
}

// go right in the game annotator //
@IBAction func AnnotatorRightAction(_:AnyObject)
{
    // if the game annotator doesn't go past the maximum value //
    if GameAnnotatorSlider.intValue + 1 <= Int32(GameAnnotatorSlider.maxValue)
    {
        // increment the game annotator slider location //
        GameAnnotatorSlider.intValue += 1

        // update the game annotator //
        UpdateGameAnnotatorFromSlider(self)
    }
}

// go left in database explorer //
@IBAction func DatabaseExplorerLeftAction(_:AnyObject)
{
    // if the database explorer doesn't go lower than the minimum value //
    if DatabaseExplorerSlider.intValue - 1 >= 0 && ReadInGameList
    {
        // decrement the database explorer slider //
        DatabaseExplorerSlider.intValue -= 1

        // update the database explorer //
        UpdateDatabaseExplorer(self)
    }
}

// go left one frame in the game gallery or grid //
@IBAction func GameGalleryGridLeftAction(_:AnyObject)
{
    // if currently animating //
    if GalleryGridView.Animating
    {
        // stop the animation //
        GalleryGridView.StopAnimation()
    }
    // not currently animating //
    else
    {
        // go left one frame in the game gallery or grid //
        GalleryGridView.Animate(false)
    }
}

// go right one frame in the game gallery or grid //
@IBAction func GameGalleryGridRightAction(_:AnyObject)
{
    // if currently animating //
    if GalleryGridView.Animating
    {
        // stop the animation //
        GalleryGridView.StopAnimation()
    }
    // not currently animating //
    else
    {
        // go right one frame in the game gallery or grid //
        GalleryGridView.Animate(true)
    }
}

// go left in the game explorer //
@IBAction func GameExplorerLeftAction(_:AnyObject)
{
    // if the game explorer doesn't go lower than the minimum value //
    if GameExplorerSlider.intValue - 1 >= 0
    {
        // decrement the game explorer slider //
        GameExplorerSlider.intValue -= 1

        // update the game explorer //
        UpdateGameExplorer(self)
    }
}

// go right in the database explorer //
@IBAction func DatabaseExplorerRightAction(_:AnyObject)
{
    // if the database explorer doesn't go higher than the maximum value //
    if DatabaseExplorerSlider.intValue + 1 <= Int32(DatabaseExplorerSlider.maxValue) &&
       ReadInGameList
    {
        // increment the database explorer slider value //
        DatabaseExplorerSlider.intValue += 1

        // update the database explorer //
        UpdateDatabaseExplorer(self)
    }
}

// go right in the game explorer //
@IBAction func GameExplorerRightAction(_:AnyObject)
{
    // if doesn't go past the maximum value //
    if GameExplorerSlider.intValue + 1 <= Int32(GameExplorerSlider.maxValue)
    {
        // increment the game explorer slider //
        GameExplorerSlider.intValue += 1

        // update the game explorer //
        UpdateGameExplorer(self)
    }
}

// set the gallery grid buttons states //
func SetGalleryGridButtonsToState(_ State:Bool)
{
    // if in the game grid //
    if InGameGrid
    {
        // update the game grid left button state //
        GameGridLeft.isEnabled = State

        // update the game grid right button state //
        GameGridRight.isEnabled = State

        // update the game grid movie button state //
        GameGridMovie.isEnabled = State

        // update the game grid snapshot button state //
        GameGridSnapshot.isEnabled = State

        // update the game grid erase button state //
        GameGridErase.isEnabled = State

        // update the game grid blue color button state //
        GameGridBlueColor.isEnabled = State

        // update the game grid cyan color button state //
        GameGridCyanColor.isEnabled = State

        // update the game grid magenta color button state //
        GameGridMagentaColor.isEnabled = State

        // update the game grid green color button state //
        GameGridGreenColor.isEnabled = State

        // update the game grid red color button state //
        GameGridRedColor.isEnabled = State

        // update the game grid yellow color button state //
        GameGridYellowColor.isEnabled = State

        // update the game grid black color button state //
        GameGridBlackColor.isEnabled = State

        // update the game grid white color button state //
        GameGridWhiteColor.isEnabled = State

        // update the game grid undo shape button state //
        GameGridUndoShape.isEnabled = State
    }
    // if in the game gallery //
    else
    {
        // update the game gallery left button state //
        GameGalleryLeft.isEnabled = State

        // update the game gallery right button state //
        GameGalleryRight.isEnabled = State

        // update the game gallery movie button state //
        GameGalleryMovie.isEnabled = State

        // update the game gallery snapshot button state //
        GameGallerySnapshot.isEnabled = State

        // update the game gallery erase button state //
        GameGalleryErase.isEnabled = State

        // update the game gallery blue color button state //
        GameGalleryBlueColor.isEnabled = State

        // update the game gallery cyan color button state //
        GameGalleryCyanColor.isEnabled = State

        // update the game gallery magenta color button state //
        GameGalleryMagentaColor.isEnabled = State

        // update the game gallery green color button state //
        GameGalleryGreenColor.isEnabled = State

        // update the game gallery red color button state //
        GameGalleryRedColor.isEnabled = State

        // update the game gallery yellow color button state //
        GameGalleryYellowColor.isEnabled = State

        // update the game gallery black color button state //
        GameGalleryBlackColor.isEnabled = State

        // update the game gallery red color button state //
        GameGalleryWhiteColor.isEnabled = State

        // update the game gallery undo shape button state //
        GameGalleryUndoShape.isEnabled = State
    }

    // update the gallery grid slider state //
    GalleryGridSlider.isEnabled = State

    // if reviewing //
    if GalleryGridView.Reviewing
    {
        // set the gallery grid review button icon to the pause icon //
        GalleryGridReview.image = PauseIcon
    }
    // if not reviewing //
    else
    {
        // set the gallery grid review button icon to the play icon //
        GalleryGridReview.image = PlayIcon
    }
}

// review the gallery grid //
@IBAction func ReviewGalleryGrid(_:AnyObject)
{
    // if animating and not reviewing //
    if !GalleryGridView.Reviewing && GalleryGridView.Animating
    {
        // stop the gallery grid view animation //
        GalleryGridView.StopAnimation()

        // exit function //
        return
    }

    // switch the gallery grid view reviewing state //
    GalleryGridView.Reviewing = !GalleryGridView.Reviewing

    // stop the gallery grid animation //
    GalleryGridView.StopAnimation()

    // if currently reviewing the gallery grid //
    if GalleryGridView.Reviewing
    {
        // go right in the game gallery grid view //
        GameGalleryGridRightAction(self)
    }
}

// review the current game being played //
@IBAction func ReviewGame(_:AnyObject)
{
    // if currently running core animation //
    if CoreAnimating
    {
        // set that want to pause the game review //
        PauseGameReview = true

        // remove all current animations from the animation layer //
        AnimationLayer.removeAllAnimations()
    }
    // not currently core animating //
    else
    {
        // play the game review //
        PlayGameReview()
    }
}

// review the game up to the current move made //
func PlayGameReview()
{
    // set the game explorer buttons to false //
    SetGameExplorerButtons(false)

    // enable the game review button //
    GameReviewButton.isEnabled = true

    // set that user is currently reviewing the game //
    ReviewingGame = true

    // set that not pausing a game review //
    PauseGameReview = false

    // if the game explorer slider is at the end of the game //
    if Int(GameExplorerSlider.intValue) == MoveList.count
    {
        // set the game explorer slider initial value //
        GameExplorerSlider.intValue = 0
    }

    // replay all the moves in the given range //
    ReplayMoves(Int(GameExplorerSlider.intValue), End:MoveList.count, Display:true)
}

// update the castle item labels on the toolbar //
func UpdateCastleItemLabel(_ ToolbarItem:NSToolbarItem, On:Bool, Kingside:Bool)
{
    // if castling is on and kingside //
    if On && Kingside
    {
        // set that kingside castling is on //
        ToolbarItem.label = "Kingside Castle On"
    }
    // if castling is on and queenside //
    else if On && !Kingside
    {
        // set that queenside castling is on //
        ToolbarItem.label = "Queenside Castle On"
    }
    // if not castling and kingside //
    else if !On && Kingside
    {
        // set that kingside castling is off //
        ToolbarItem.label = "Kingside Castle Off"
    }
    // if not castling and queenside //
    else if !On && !Kingside
    {
        // set that queenside castling is off //
        ToolbarItem.label = "Queenside Castle Off"
    }
}

// update the position editor toolbar buttons //
func UpdatePositionEditorToolbarButtons()
{
    // if not currently in the position editor //
    if !InPositionEditor
    {
        // return since not currently in the position editor //
        return
    }

    // if white moves next //
    if Attributes.FENColor == "w"
    {
        // update the white moves toolbar button image //
        WhiteMovesToolbarButton.image = GetPieceImage(1)

        // set the white moves toolbar button label to "white moves" //
        WhiteMovesToolbarButton.label = "White Moves"
    }
    // black moves next //
    else
    {
        // update the white moves toolbar button image //
        WhiteMovesToolbarButton.image = GetPieceImage(-1)

        // update the white moves toolbar button label //
        WhiteMovesToolbarButton.label = "Black Moves"
    }

    // update the white kingside castle toolbar button image //
    WhiteKingsideCastleToolbarButton.image = GetPieceImage(6)

    // update the white queenside castle toolbar button image //
    WhiteQueensideCastleToolbarButton.image = GetPieceImage(5)

    // update the black kingside castle toolbar button image //
    BlackKingsideCastleToolbarButton.image = GetPieceImage(-6)

    // update the black queenside castle toolbar button image //
    BlackQueensideCastleToolbarButton.image = GetPieceImage(-5)

    // update the position editor white pawn image //
    PositionEditorWhitePawn.image = GetPieceImage(1)

    // update the position editor white knight image //
    PositionEditorWhiteKnight.image = GetPieceImage(2)

    // update the position editor white bishop image //
    PositionEditorWhiteBishop.image = GetPieceImage(3)

    // update the position editor white rook image //
    PositionEditorWhiteRook.image = GetPieceImage(4)

    // update the position editor white queen image //
    PositionEditorWhiteQueen.image = GetPieceImage(5)

    // update the position editor white king image //
    PositionEditorWhiteKing.image = GetPieceImage(6)

    // update the position editor black pawn image //
    PositionEditorBlackPawn.image = GetPieceImage(-1)

    // update the position editor black knight image //
    PositionEditorBlackKnight.image = GetPieceImage(-2)

    // update the position editor black bishop image //
    PositionEditorBlackBishop.image = GetPieceImage(-3)

    // update the position editor black rook image //
    PositionEditorBlackRook.image = GetPieceImage(-4)

    // update the position editor black queen image //
    PositionEditorBlackQueen.image = GetPieceImage(-5)

    // update the position editor black king image //
    PositionEditorBlackKing.image = GetPieceImage(-6)

    // update the white kingside castle toolbar button label //
    UpdateCastleItemLabel(WhiteKingsideCastleToolbarButton, On:WhiteKingsideCastle, Kingside:
                          true)

    // update the white queenside castle toolbar button label //
    UpdateCastleItemLabel(WhiteQueensideCastleToolbarButton, On:WhiteQueensideCastle, Kingside:
                          false)

    // update the black kingside castle toolbar button label //
    UpdateCastleItemLabel(BlackKingsideCastleToolbarButton, On:BlackKingsideCastle, Kingside:
                          true)

    // update the black queenside castle toolbar button label //
    UpdateCastleItemLabel(BlackQueensideCastleToolbarButton, On:BlackQueensideCastle, Kingside:
                          false)

    // update position editor cursor //
    SetPosCursor(PositionMovingPiece)
}

// start a new game //
@IBAction func NewGameMenuItemAction(_:AnyObject)
{
    // turn on the new game menu item //
    NewGameMenuItem.state = NSControl.StateValue(rawValue:1)

    // set that currently querying if want to start a new game or not //
    QueryStartNewGame = true

    // create the buttons for the start new game query //
    AlertButtons = ["Yes", "No"]

    // make the new game alert //
    MakeAlert(AlertButtons, Query:"New Game?")
}

// process the alert //
func ProcessAlert()
{
    // if querying to quit the application //
    if QueryQuitApplication
```

```
{
   // play the game sound of a button click //
   PlayGameSoundThread("buttonclick")

   // if user chose the first alert button //
   if ReturnCode == NSApplication.ModalResponse.alertFirstButtonReturn.rawValue
   {
      // terminate the application //
      NSApplication.shared.terminate(self)
   }
   // if user doesn't want to quit the application //
   else
   {
      // set that no longer querying whether want to quit the application //
      QueryQuitApplication = false
   }
}
// if save position query //
else if QuerySavePosition
{
   // play the game sound of a button click //
   PlayGameSoundThread("buttonclick")

   // set query save position to false //
   QuerySavePosition = false

   // turn off the save position menu item //
   SavePositionMenuItem.state = NSControl.StateValue(rawValue:0)
}
// if loading a saved position and there are no saved positions //
else if QueryLoadPosition
{
   // play the sound of a button click //
   PlayGameSoundThread("buttonclick")

   // set loading saved position to false //
   QueryLoadPosition = false

   // turn off the load position menu item //
   LoadPositionMenuItem.state = NSControl.StateValue(rawValue:0)
}
// if querying that timed game is over //
else if QueryTimedGameOver
{
   // remove the timed game offset //
   RemoveTimedGameOffset()
}
// if querying whether wants to save the edited board position //
else if QuerySaveEditedPosition
{
   // play the game sound of a button click //
   PlayGameSoundThread("buttonclick")

   // if the fen layout is legal //
   if FENLayoutIsLegal()
   {
      // if user chose to save the edited position //
      if ReturnCode == NSApplication.ModalResponse.alertFirstButtonReturn.rawValue
      {
         // save the edited position //
         WillSaveEditedPosition = true
      }

      // if didn't choose to edit the position again //
      if ReturnCode != NSApplication.ModalResponse.alertSecondButtonReturn.rawValue
      {
         // exit the position editor //
         ExitPositionEditor()
      }
   }
   // the fen layout was not legal and chose to exit //
   else if ReturnCode == NSApplication.ModalResponse.alertSecondButtonReturn.rawValue
   {
      // exit the position editor //
      ExitPositionEditor()
   }

   // set that no longer querying to save edited position //
   QuerySaveEditedPosition = false
}
// if querying user if wants to start a new game or not //
else if QueryStartNewGame
{
   // play the game sound of a button click //
   PlayGameSoundThread("buttonclick")

   // if user chose the first alert button //
   if ReturnCode == NSApplication.ModalResponse.alertFirstButtonReturn.rawValue
   {
      // start a new game of chess //
      StartNewGame(true)

      // reset the game clock //
      ResetGameClock()

      // store the autosave game data //
      StoreAutosaveData("")
   }

   // set that done querying user if wants to start a new game or not //
   QueryStartNewGame = false

   // turn off the new game menu item //
   NewGameMenuItem.state = NSControl.StateValue(rawValue:0)
}
// if querying to open a game file //
else if QueryOpenGameFile
{
   // play the game sound of a button click //
   PlayGameSoundThread("buttonclick")

   // if user chose the first alert button //
   if ReturnCode == NSApplication.ModalResponse.alertFirstButtonReturn.rawValue
   {
      // open the game with game filename in 0.01 seconds //
      RunLoop.current.add(Timer(timeInterval:0.01, target:self, selector:
                          #selector(GreenChess.OpenGame), userInfo:nil, repeats:false),
                          forMode:RunLoopMode.commonModes)
   }
   // user chose the other alert button //
   else
   {
      // set that done querying whether to open a game file //
      QueryOpenGameFile = false
   }
}
// if querying what types of piece user wants to promote pawn to //
else if QueryPawnPromotion
{
   // play the game sound of a button click //
   PlayGameSoundThread("buttonclick")
```

```
      // if user chose the first alert button //
      if ReturnCode == NSApplication.ModalResponse.alertFirstButtonReturn.rawValue
      {
         // set the promotion type symbol to be a queen //
         PromotionSymbol = "q"

         // set the moving piece to be a queen //
         MovingPiece = 5
      }
      // if user chose the second alert button //
      else if ReturnCode == NSApplication.ModalResponse.alertSecondButtonReturn.rawValue
      {
         // set the promotion type symbol to be a rook //
         PromotionSymbol = "r"

         // set the moving piece to be a rook //
         MovingPiece = 4
      }
      // if user chose the third alert button //
      else if ReturnCode == NSApplication.ModalResponse.alertThirdButtonReturn.rawValue
      {
         // set the promotion type symbol to be a bishop //
         PromotionSymbol = "b"

         // set the moving piece to be a bishop //
         MovingPiece = 3
      }
      // if user chose the fourth alert button //
      else if ReturnCode == NSApplication.ModalResponse.alertThirdButtonReturn.rawValue + 1
      {
         // set the promotion type symbol to be a knight //
         PromotionSymbol = "n"

         // set the moving piece to be a knight //
         MovingPiece = 2
      }

      // if black is currently moving //
      if BoardIsBlack(Board)
      {
         // make the moving piece black //
         MovingPiece = -MovingPiece
      }

      // update the last move on the move list to show a promotion occurred //
      MoveList[MoveList.count - 1] += PromotionSymbol

      // finish up the move just made //
      FinishUpMove()

      // set that done querying what piece user wants to promote pawn to //
      QueryPawnPromotion = false
   }
   // if querying whether the uci chess engine was loaded or not //
   else if QueryUCIEngineLoad
   {
      // play the game sound of a button click //
      PlayGameSoundThread("buttonclick")

      // set that no longer querying whether the uci chess engine was loaded or not //
      QueryUCIEngineLoad = false
   }
}

// get the file symbol for mouse coordinate x //
func FileSymbol(_ X:Int) -> String
{
   // subtract board offset in x direction and divide by square width //
   RankAndFile.x = CGFloat((X - 30) / SquareSize)

   // if the board has been rotated //
   if RotateBoard
   {
      // the rotated rank //
      RankAndFile.x = 7 - RankAndFile.x
   }

   // return the appropriate file symbol //
   return FileSymbols[Int(RankAndFile.x)]
}

// get the rank symbol for mouse coordinate y //
func RankSymbol(_ Y:Int) -> String
{
   // subtract board offset in y direction and divide by square width //
   RankAndFile.y = CGFloat((Y - 30) / SquareSize)

   // if the board has been rotated //
   if RotateBoard
   {
      // the rotated file //
      RankAndFile.y = 7 - RankAndFile.y
   }

   // return the appropriate rank symbol //
   return RankSymbols[Int(RankAndFile.y)]
}

// function to create the square label for mouse coordinate location //
func SquareLabel() -> String
{
   // return the square label for mouse coordinate location //
   return FileSymbol(Int(MouseLocation.x)) + RankSymbol(Int(MouseLocation.y))
}

// hide the application window //
@IBAction func HideAppMenuItemAction(_:AnyObject)
{
   // if the main window is visible //
   if MainWindow.occlusionState.contains(NSWindow.OcclusionState.visible)
   {
      // hide the application windows //
      NSApp.hide(nil)

      // set the cursor to arrow //
      NSCursor.arrow.set()
   }
   // the main window is not visible //
   else
   {
      // set the cursor to arrow //
      NSCursor.arrow.set()

      // make the main window visible //
      MainWindow.makeKeyAndOrderFront(nil)
   }
}
```

```
// if have an attached sheet //
func HaveAttachedSheet() -> Bool
{
   // return whether a sheet is attached or not //
   return MainWindow.attachedSheet != nil
}

// close the attached sheet if have one //
func CloseAttachedSheet()
{
   // set that going into a panel //
   InPanel = true

   // get the previously attached sheet //
   PreviouslyAttachedSheet = MainWindow.attachedSheet as! AWindow?

   // if have a previously attached sheet //
   if PreviouslyAttachedSheet != nil
   {
      // end the sheet //
      MainWindow.endSheet(PreviouslyAttachedSheet!)

      // close the previously attached sheet //
      PreviouslyAttachedSheet!.close()
   }
}

// reopen the previously attached sheet if had one //
func ReopenPreviouslyAttachedSheet()
{
   // if there was a previously attached sheet //
   if PreviouslyAttachedSheet != nil
   {
      // reopen the previously attached sheet //
      MainWindow.beginSheet(PreviouslyAttachedSheet!)

      // reset the previously attached sheet to nil //
      PreviouslyAttachedSheet = nil
   }

   // set that no longer in a panel //
   InPanel = false
}

// determine if user wants to quit the application //
@IBAction func QuitAppMenuItemAction(_:AnyObject)
{
   // set that currently querying user if wants to quit the application //
   QueryQuitApplication = true

   // set the alert buttons //
   AlertButtons = ["Quit", "Cancel"]

   // alert the user if they want to quit the application or not //
   MakeAlert(AlertButtons, Query:"Quit GreenChess?")

   // make the alert to query if want to quit application or not //
   MakeAlert(AlertButtons, Query:"Quit GreenChess?")

   // make the main window visible //
   MainWindow.makeKeyAndOrderFront(nil)
}

// undo the last chess move played in the game //
@IBAction func UndoMoveMenuItemAction(_:AnyObject)
{
   // add a timer to run the undo menu item action //
   RunLoop.current.add(Timer(timeInterval:0.01, target:self, selector:#selector(GreenChess
                       .TheUndoMoveMenuItemAction), userInfo:nil, repeats:false), forMode:
                       RunLoopMode.commonModes)
}

// the undo move menu item action //
@objc func TheUndoMoveMenuItemAction()
{
   // if there are no moves in the move list to undo //
   if MoveList.isEmpty
   {
      // exit function since there are no moves in the move list to undo //
      return
   }

   // set the previous move list to the current move list //
   PreviousMoveList = MoveList

   // set the previous annotation list to the current annotation list //
   PreviousAnnotationList = AnnotationList

   // remove the last move from the move list //
   MoveList.removeLast()

   // remove the last annotation from the move annotation list //
   AnnotationList.removeLast()

   // replay all the moves excluding the last move //
   ReplayMoves(0, End:MoveList.count, Display:false)

   // compact the transposition table //
   EngineCompactTransTable(Engine, false)

   // store the autosave game data //
   StoreAutosaveData("")
}

// append a frame for the grid, gallery, or game movie //
func AppendFrame()
{
   // if recording a movie //
   if RecordingMovie
   {
      // take a board snapshot //
      TakeBoardSnapshot(660)

      // append the board snapshot to the frames //
      Frames.append(SnapshotImage)
   }
   // in the grid or gallery //
   else
   {
      // add the pieces layer to the frames //
      Frames.append(PiecesLayer)
   }
}

// convert the game to a sequence of frames //
func ConvertGameToFrames()
{
   // clear the frames //
   Frames.removeAll()

   // initialize the board //
   InitializeBoard()
```

```
   // update the board and captured pieces //
   UpdateBoardAndCapturedPieces()

   // append a frame //
   AppendFrame()

   // loop over all the moves in the move list //
   for Index in 0 ..< MoveList.count
   {
      // get the move from the move list to animate //
      UCIMove = MoveList[Index]

      // make the move on the board //
      GCMakeMoveOnBoard(UCIMove)

      // if the move looking for does exist //
      if MoveExists
      {
         // process the current uci move //
         ProcessUCIMove()

         // if had a promotion //
         if PromotionType != 0
         {
            // set the piece moving to the type of piece pawn promoted to //
            MovingPiece = PromotionType
         }

         // put the piece that moved onto the destination square //
         UpdateDestinationSquare()

         // perform en passant move and capture if needed //
         PerformEnpassantMoveIfNeeded()

         // perform castling move if had one //
         PerformCastleIfNeeded()

         // update the board and captured pieces //
         UpdateBoardAndCapturedPieces()

         // append a frame //
         AppendFrame()
      }
   }
}

// get the game gallery or grid images //
@objc func GetGalleryGridImages()
{
   // set that currently creating the game gallery or grid images //
   CreatingGalleryGrid = true

   // convert the game to frames //
   ConvertGameToFrames()

   // save the frames to the gallery grid view //
   GalleryGridView.Frames = Frames

   // create the sized board composite for the width of a frame //
   CreateSizedBoardComposite(CGFloat(GalleryGridView.FrameWidth))

   // assign the sized board composite to the gallery grid view board composite //
   GalleryGridView.BoardComposite = SizedBoardComposite

   // create the game gallery or grid image //
   GalleryGridView.CreateGalleryGridImage(InGameGrid)

   // update the game gallery or grid view //
   GalleryGridView.UpdateGalleryGridView(GalleryGridSlider.doubleValue)

   // run on the main thread //
   DispatchQueue.main.async
   {
      // begin the game gallery or grid window //
      self.MainWindow.beginSheet(self.GalleryGridWindow, completionHandler:nil)

      // set the main window title to be "game gallery" or "game grid" //
      self.MainWindow.title = self.GalleryGridTitle
   }

   // set that done creating the game gallery //
   CreatingGalleryGrid = false
}

// replay the moves from start to end //
func ReplayMoves(_ Start:Int, End:Int, Display:Bool)
{
   // if in the position editor //
   if InPositionEditor
   {
      // return since don't want to replay the moves from within the position editor //
      return
   }

   // set the starting ply count //
   PlyCount = Start

   // if starting from the beginning move //
   if Start == 0
   {
      // initialize the board //
      InitializeBoard()

      // remove all the illustrations //
      Illustrations.removeAll()
   }

   // if need to update the board display //
   if Display
   {
      // update the board //
      UpdateBoardAndCapturedPieces()

      // print the game information //
      PrintGameInfo()
   }

   // set the starting move index //
   Index = Start
```

```
   // if replaying moves with display //
   if Display
   {
      // run core animation of game replay //
      RunCoreAnimation()
   }
   // not replaying moves with display //
   else
   {
      // loop over all the moves in the move list //
      for Index in Start ..< End
      {
         // get the move from the move list to animate //
         UCIMove = MoveList[Index]

         // make the move on the board //
         GCMakeMoveOnBoard(UCIMove)

         // if the move looking for does exist //
         if MoveExists
         {
            // increment the ply count //
            PlyCount += 1

            // process the current uci move //
            ProcessUCIMove()

            // if had a promotion //
            if PromotionType != 0
            {
               // set the piece moving to the type of piece pawn promoted to //
               MovingPiece = PromotionType
            }

            // put the piece that moved onto the destination square //
            UpdateDestinationSquare()

            // perform en passant move and capture if needed //
            PerformEnpassantMoveIfNeeded()

            // perform castling move if had one //
            PerformCastleIfNeeded()

            // if not in the unneeded portions of the database or game explorer move list //
            if !((InDatabaseExplorer || GameExplorerWindowOpen) && (End - Index >= 6))
            {
               // update the pieces layer for the illustrations //
               UpdateIllustrationPiecesLayer()
            }
            // if in the unneeded portions of the database or game explorer move list //
            else
            {
               // append an empty image to the illustrations move list //
               Illustrations.append(NSImage())
            }
         }
         // the move doesn't exist //
         else
         {
            // append an empty image for the illustrations //
            Illustrations.append(NSImage())

            // set the move to be an invalid move since it doesn't exist //
            MoveList[Index] = "Invalid Move"
         }
      }

      // update the board //
      UpdateBoardAndCapturedPieces()

      // print the game info //
      PrintGameInfo()
   }
}

// update the chessboard display //
func UpdateBoardAndCapturedPieces()
{
   // get the list of squares where pieces can be captured on //
   ListOfCaptureSquares = String(validatingUTF8:BoardGetListOfCaptureSquares(Board))!
                                .components(separatedBy:"*")

   // remove the last object in the list of capture squares since it is empty //
   ListOfCaptureSquares.removeLast()

   // set the board so can check for squares where pieces are threatened //
   BoardCopy(ThreatenedBoard, Board)

   // get the list of squares where pieces can be threatened //
   ListOfThreatenSquares = String(validatingUTF8:
                                 BoardGetListOfThreatenedSquares(ThreatenedBoard))!
                                 .components(separatedBy:"*")

   // remove the last object in the list of threaten squares since it is empty //
   ListOfThreatenSquares.removeLast()

   // if annotating the game //
   if AnnotatingGame
   {
      // update the pieces layer for the annotation size //
      UpdatePiecesLayer(250)
   }
   // if in the game grid or gallery //
   else if InGameGrid || InGameGallery
   {
      // update the pieces layer for the gallery grid size //
      UpdatePiecesLayer(CGFloat(GalleryGridView.FrameWidth))
   }
   // not in the game grid or gallery //
   else
   {
      // update the pieces layer for the regular board size //
      UpdatePiecesLayer(660)

      // if not recording a movie //
      if !RecordingMovie
      {
         // set that need to update the display //
         self.needsDisplay = true

         // display if needed //
         displayIfNeeded()
      }

      // if the previous layer was -1 //
      if PreviousLayer == -1
      {
         // recomposite the entire board //
         RecompositeBoard(0, Span:1, Width:660)
      }
```

```
         // if entering preferences //
         if EnteringPreferences
         {
            // update the board preview //
            UpdateBoardPreview()
         }
      }

      // if not in game grid, not in gallery grid, not recording movie, and not annotating //
      if !InGameGrid && !InGameGallery && !RecordingMovie && !AnnotatingGame
      {
         // update the captured white pieces display //
         CapturedPiecesWhiteView.UpdateCapturedPiecesDisplay(CapturedWhitePieces, White:true, Set:
                                                             ChessPieceSetType)

         // update the captured black pieces display //
         CapturedPiecesBlackView.UpdateCapturedPiecesDisplay(CapturedBlackPieces, White:false, Set:
                                                             ChessPieceSetType)
      }
   }

   // determine if a white pawn is currently moving //
   func WhitePawnIsMoving() -> Bool
   {
      // return whether a white pawn is currently moving //
      return MovingPiece == 1
   }

   // determine if a black pawn is currently moving //
   func BlackPawnIsMoving() -> Bool
   {
      // return whether a black pawn is moving //
      return MovingPiece == -1
   }

   // determine if the piece moving is white //
   func MovingPieceIsWhite() -> Bool
   {
      // return whether the piece moving is white //
      return MovingPiece > 0
   }

   // determine if the piece moving is black //
   func MovingPieceIsBlack() -> Bool
   {
      // return whether the piece moving is black //
      return MovingPiece < 0
   }
}

// get the application value //
let ReturnValue:Int32 = NSApplicationMain(CommandLine.argc, CommandLine.unsafeArgv)
```