```
////////////////////////////////////////////////////////////////////////
// Swift source code of Classes.swift Version 9.0.6 at greenchess.com //
////////////////////////////////////////////////////////////////////////

// import needed system files //
import SwiftUI
import AppKit
import AVFoundation

// the cursor tracker //
class CursorTracker
{
   // the marker cursor names //
   var Markers:[Int:String] = [304:"marker_red_icon", 305:"marker_green_icon", 306:
                               "marker_blue_icon", 307:"marker_cyan_icon", 308:"marker_magenta_icon",
                               309:"marker_yellow_icon", 310:"marker_black_icon", 311:
                               "marker_white_icon"]

   // if was previously inside the bounds //
   var PreviouslyInside:Bool = false

   // if inside the bounds //
   var Inside:Bool = false

   // the current point in the view //
   var CurrentPoint:CGPoint = CGPoint()

   // the window origin //
   var Origin:CGPoint = CGPoint()

   // the list of points in the tracker //
   var Points:[CGPoint] = []

   // a point in the list of points in the tracker //
   var Point:CGPoint = CGPoint()

   // the previous point in the view //
   var PreviousPoint:CGPoint = CGPoint()

   // the screen view bounds //
   var Bounds:CGRect = CGRect()

   // the view //
   var View:NSView!

   // the x offset //
   var XOffset:CGFloat = 0

   // the y offset //
   var YOffset:CGFloat = 0

   // the view cursor //
   var Cursor:NSCursor = NSCursor.arrow

   // whether currently tracking //
   var Tracking:Bool = false

   // set the points tracking dispatch queue //
   let TrackingQueue = DispatchQueue(label:"TrackingThread", attributes:.concurrent)

   // if the current point is in bounds //
   func InBounds() -> Bool {return Bounds.contains(Location())}

   // refresh the cursor //
   func RefreshCursor()
   {
      // run on the main queue to set the cursor based on if inside or outside the bounds //
      DispatchQueue.main.async {if self.InBounds() {self.Cursor.set()} else {NSCursor.arrow.set()}}
   }

   // update the cursor //
   func UpdateCursor(_ Color:String)
   {
      // update the cursor icon //
      Cursor = NSCursor.init(image:NSImage(named:Markers[(304...311).contains(Int(Color)!) ?
                             Int(Color)! : 309]!)!, hotSpot:CGPoint(x:2, y:30))
   }

   // initialize the mouse tracker //
   func Initialize(_ View:NSView, Margin:CGFloat, Bounds:CGRect, YOffset:CGFloat)
   {
      // set the bounds, view, x offset, y offset, tracking status, and cursor //
      (self.Bounds, self.View, self.XOffset, self.YOffset, Tracking, Cursor) = (CGRect(x:Margin, y:
      Margin, width:Bounds.width - 2 * Margin, height:Bounds.height - 2 * Margin), View, 0, YOffset,
      false, NSCursor.arrow)
   }

   // start the cursor tracking //
   func Start(_ XOffset:CGFloat)
   {
      // set the x offset, that currently tracking, previous point, and inside values to false //
      (self.XOffset, Tracking, PreviousPoint, PreviouslyInside, Inside, Origin) = (XOffset, true,
      Location(), false, false, View.window!.frame.origin)

      // clear the cursor tracking points //
      Points.removeAll()

      // start the tracking queue to collect the current mouse location //
      TrackingQueue.async {while self.Tracking {self.SaveMouseLocation()}}
   }

   // get the current mouse location //
   func Location() -> CGPoint
   {
      // get the current point //
      CurrentPoint = NSEvent.mouseLocation

      // return the current mouse location by subracting the window origin and the y offset //
      return CGPoint(x:CurrentPoint.x - View.window!.frame.origin.x, y:CurrentPoint.y - View.window!
             .frame.origin.y - YOffset)
   }

   // save the mouse location //
   func SaveMouseLocation()
   {
      // sleep the thread for a small amount of time //
      Thread.sleep(forTimeInterval:0.000001)

      // set the current mouse location //
      Point = NSEvent.mouseLocation

      // subtract the window origin from the current mouse location //
      Point = CGPoint(x:Point.x - Origin.x, y:Point.y - Origin.y - YOffset)

      // if the point is far enough away from the last point //
      if (abs(Point.x - PreviousPoint.x) > 0.001) || (abs(Point.y - PreviousPoint.y) > 0.001)
      {
         // set if point currently inside the bounds //
         Inside = Bounds.contains(Point)

         // append the point to the list of points //
         Points.append(CGPoint(x:2 * max(Bounds.minX, min(Point.x, Bounds.maxX)) - XOffset, y:2 *
                     max(Bounds.minY, min(Point.y, Bounds.maxY))))
```

```
         // refresh the cursor if cursor crossed the view bounds//
         if Inside != PreviouslyInside {RefreshCursor()}

         // set the previous point and whether previously inside the bounds //
         (PreviousPoint, PreviouslyInside) = (Point, Inside)
      }
   }
}

// the process frames queue //
class ProcessFramesQueue
{
   // whether the queue is done processing or not //
   var Done:Bool = false

   // a dispatch queue //
   var Queue:DispatchQueue!

   // the pgn moves data //
   var Data:PGNMovesDataSource!

   // the game attributes //
   var Attributes:GameAttributes!

   // the board and pieces compositor //
   var BPComp:BoardPieceCompositor!

   // the processed frames //
   var Frames:[CGImage] = []

   // the core graphics functions //
   let Graphics:CoreGraphics = CoreGraphics()

   // the pixel buffers //
   var Buffers:[CVPixelBuffer?] = []

   // the queue label number //
   var Label:Int = 0

   // the context //
   var Context:CGContext!

   // the number of threads to use //
   var Threads:Int = ProcessInfo.processInfo.processorCount

   // the process frames queue initializer //
   init(_ Label:Int, Data:PGNMovesDataSource, Attributes:GameAttributes, BPComp:
      BoardPieceCompositor)
   {
      // set the dispatch queue to process frames, data, attributes, bpcomp, and the label //
      (Queue, self.Data, self.Attributes, self.BPComp, self.Label) = (DispatchQueue(label:
      String(Label), attributes:.concurrent), Data, Attributes, BPComp, Label)
   }

   // get the image for a frame //
   func GetBoard(_ Size:CGFloat, Index:Int, Info:Bool) -> CGImage
   {
      // return the board image for the specified size and index and if overlay game info //
      return BPComp.Recomposite(Size, BPData:Data.GetData(Index), InPositionEditor:false,
                                FENEnpassantSquare:"", EnpassantSquare:APoint(), RankAndFile:
                                APoint(), UpdateBoard:false, Info:Info)
   }

   // reset the frames, buffers, and context to empty //
   func Reset()
   {
      // loop over all the image frames //
      for Index in 0..<Frames.count
      {
         // reset the image frame //
         Frames[Index] = Graphics.CreateContext(1, Height:1).makeImage()!
      }

      // loop over all the pixel buffers to reset them //
      for Index in 0..<Buffers.count {Buffers[Index] = nil}

      // remove all the frames //
      Frames.removeAll()

      // remove all the buffers //
      Buffers.removeAll()

      // reset the context //
      Context = Graphics.CreateContext(1, Height:1)

      // reset done to false //
      Done = false
   }

   // convert the boards to pixel buffers //
   func BoardsToBuffers(_ Size:CGFloat)
   {
      // run on the queue //
      Queue.async
      {
         // loop over all the boards and see if is the matching index to process for this thread //
         for Index in 0...self.Attributes.Inf.GetCount() {if Index % self.Threads == self.Label
         {
            // append a pixel buffer to the pixel buffers list //
            self.Buffers.append(self.Graphics.ImageToBuffer(self.GetBoard(Size, Index:Index, Info:
                false)))
         }}

         // set that done processing this queue //
         self.Done = true
      }
   }

   // convert the layer to pixel buffers //
   func LayerToBuffers(_ Layer:CGLayer, Width:CGFloat, Height:CGFloat, Scale:CGFloat, Count:Int,
                       FrameSize:CGFloat)
   {
      // run on the queue //
      Queue.async
      {
         // set the context //
         self.Context = self.Graphics.CreateContext(Width, Height:Height)

         // scale the context //
         self.Context.scaleBy(x:Scale, y:Scale)

         // loop over all the boards and see if is the matching index to process for this thread //
         for Index in 0...Count {if Index % self.Threads == self.Label
         {
            // draw the layer in the context at the specified point //
            self.Context.draw(Layer, at:CGPoint(x:CGFloat(-Index) * FrameSize, y:0))

            // append the pixel buffer to the pixel buffers list //
            self.Buffers.append(self.Graphics.ImageToBuffer(self.Context.makeImage()!))
         }}

         // set that done processing this queue //
         self.Done = true
      }
   }
```

```
   // convert the boards to frames //
   func BoardsToFrames(_ Size:CGFloat)
   {
      // run on the queue //
      Queue.async
      {
         // loop over the frames to process of this queue //
         for Index in 0...self.Attributes.Inf.GetCount() {if Index % self.Threads == self.Label
         {
            // append the board for the specified size and index to the frames list //
            self.Frames.append(self.GetBoard(Size, Index:Index, Info:false))
         }}

         // set that done processing this queue //
         self.Done = true
      }
   }
}

// the movie and frames creator //
class FramesMovieCreator:NSObject
{
   // the write the movie queue //
   var Queue:DispatchQueue!

   // whether the queues are done processing or not //
   var Done:Bool = false

   // the process frames queues //
   var Queues:[ProcessFramesQueue] = []

   // the number of threads //
   var Threads:Int = ProcessInfo.processInfo.processorCount

   // the preference controls //
   @IBOutlet var Ctr:PreferenceController!

   // the pgn moves list data //
   @IBOutlet var PGNMovesData:PGNMovesDataSource!

   // the game attributes //
   @IBOutlet var Attributes:GameAttributes!

   // the board and pieces compositor //
   @IBOutlet var BPComp:BoardPieceCompositor!

   // the information dictionary //
   var Inf:Information = Information()

   // the movie frames per second //
   var MovieFPS:CGFloat = 1

   // the video width //
   var Width:CGFloat = 0

   // the video height //
   var Height:CGFloat = 0

   // the cglayer //
   var Layer:CGLayer!

   // the total frames wide //
   var TotalFrames:Int = 0

   // the maximum position //
   var MaxPosition:Int = 0

   // the frame size //
   var FrameSize:CGFloat = 0

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

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

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

   // the core graphics functions //
   let Graphics:CoreGraphics = CoreGraphics()

   // save the preferences //
   func SavePrefs() {Inf.SavePrefs(["VBPref"], VITags:["VIPref"], VSTags:["VSPref"])}

   // override the frames and movie class constructor //
   override init()
   {
      // update the preferences with the defaults //
      Inf.Update("VIPref", VDefMinMaxIntItems:["MovieFPSFromSlider":[10, 0, 100], "MovieSize":[75,
               25, 100]])

      // get the saved preferences //
      Inf.GetPrefs(["VBPref"], VITags:["VIPref"], VSTags:["VSPref"])
   }

   // read the preferences //
   func ReadPrefs()
   {
      // set the movie size slider value, the display movie size, and the write movie queue //
      (Ctr.MovieSizeSlider.intValue, Ctr.MovieSizeView.stringValue, Queue) = (Int32(Inf
      .Vals("MovieSize").VI), "Movie size is " + Inf.IntStr("MovieSize") + "%", DispatchQueue(label:
      "Queue", attributes:.concurrent))

      // loop over all the threads //
      for Index in 0..<Threads
      {
         // append a render frame dispatch queue //
         Queues.append(ProcessFramesQueue(Index, Data:PGNMovesData, Attributes:Attributes, BPComp:
                     BPComp))
      }
   }

   // update the movie size //
   @IBAction func UpdateMovieSize(_:AnyObject)
   {
      // update the movie size preference value //
      Inf.Update("VIPref", VIItems:["MovieSize":Int(Ctr.MovieSizeSlider.intValue)])

      // display the movie size //
      Ctr.MovieSizeView.stringValue = "Movie size is " + Inf.IntStr("MovieSize") + "%"
   }

   // update the movie frames per second from slider value //
   func UpdateMovieFPS(_ Value:Double)
   {
      // update the movie frames per second from slider value //
      Inf.Update("VIPref", VIItems:["MovieFPSFromSlider":Int(Value)])
   }
```

```
// display the movie frames per second from slider value //
func DisplayMovieFPS(_ Value:Double) -> String
{
   // set the movie frames per second //
   MovieFPS = round(Inf.VF("MovieFPSFromSlider") * 3.9 + 10) / 100

   // return the movie frames per second //
   return "Movie frames per second is \(MovieFPS)"
}

// return the x coordinate //
func GetX(_ Index:Int) -> CGFloat {return CGFloat(-Index) * FrameSize}

// create a layer from the frames //
func Create(_ Size:CGFloat, Rows:Int, FramesWide:Int)
{
   // set the frame size //
   FrameSize = Size

   // convert all the boards to frames //
   for TheQueue in Queues {TheQueue.BoardsToFrames(FrameSize)}

   // while not done processing the frames from the queues //
   repeat {Done = true; for TheQueue in Queues {if !TheQueue.Done {Done = false}}} while !Done

   // the maximum position //
   MaxPosition = max(Attributes.Inf.GetCount() + 1 - FramesWide * Rows, 0)

   // the total frames wide //
   TotalFrames = MaxPosition + FramesWide

   // create the frame layer //
   Layer = Graphics.CreateLayer(CGFloat(TotalFrames) * Size, Height:Size * CGFloat(Rows))

   // loop over the boards //
   for Index in 0...Attributes.Inf.GetCount()
   {
      // loop over all the rows and see if the frame is in bounds //
      for Row in 0..<Rows {if (0..<TotalFrames).contains(Index - Row * FramesWide)
      {
         // draw the image into the layer context //
         Layer.context?.draw(GetFrame(Index), in:CGRect(x:CGFloat(Index - Row * FramesWide) *
               Size, y:CGFloat(Rows - 1 - Row) * Size, width:Size, height:Size), byTiling:false)
      }}
   }

   // reset all the queues //
   for TheQueue in Queues {TheQueue.Reset()}
}

// get the pixel buffer for the specified index //
func GetBuffer(_ Index:Int) -> CVPixelBuffer?
{
   // return the pixel buffer from the specified queue and index //
   return Queues[Index % Threads].Buffers[Index / Threads]
}

// get the frame at the specified index //
func GetFrame(_ Index:Int) -> CGImage {return Queues[Index % Threads].Frames[Index / Threads]}

// write to a movie //
func WriteMovie(_ Layer:CGLayer, VidWidth:CGFloat, VidHeight:CGFloat, Path:URL, Layered:Bool)
{
   // set the video width and height //
   (Width, Height) = (VidWidth * Inf.Fraction(100, Key:"MovieSize"), VidHeight * Inf
   .Fraction(100, Key:"MovieSize"))

   // if for a layer, loop over all the queues //
   if Layered {for TheQueue in Queues
   {
      // convert the layer to pixel buffers //
      TheQueue.LayerToBuffers(Layer, Width:Width, Height:Height, Scale:Inf.Fraction(100, Key:
                              "MovieSize"), Count:MaxPosition, FrameSize:FrameSize)
   }}

   // if not layered, convert the board frames to pixel buffers //
   if !Layered {for TheQueue in Queues {TheQueue.BoardsToBuffers(Width)}}

   // while not done processing the buffers from the queues //
   repeat {Done = true; for TheQueue in Queues {if !TheQueue.Done {Done = false}}} while !Done

   // if the movie file already exists //
   if FileManager.default.fileExists(atPath:Path.path)
   {
      // remove the old movie file since av can't overwrite existing movie //
      try? FileManager.default.removeItem(atPath:Path.path)
   }

   // assign the video writer and video input //
   (VidInput, VideoWriter) = (AVAssetWriterInput(mediaType:AVMediaType.video, outputSettings:
   [AVVideoCodecKey:AVVideoCodecType.h264, AVVideoWidthKey:NSNumber(value:Int(Width)),
   AVVideoHeightKey:NSNumber(value:Int(Height))]), try? AVAssetWriter(outputURL:Path, fileType:
   AVFileType.mov))

   // assign the adaptor //
   Adaptor = AVAssetWriterInputPixelBufferAdaptor(assetWriterInput:VidInput!,
             sourcePixelBufferAttributes:[kCVPixelBufferPixelFormatTypeKey as String:
             NSNumber(value:kCVPixelFormatType_32ARGB as UInt32), kCVPixelBufferWidthKey as
             String:NSNumber(value:Int(Width)), kCVPixelBufferHeightKey as String:NSNumber(value:
             Int(Height))])

   // add the video input //
   VideoWriter!.add(VidInput!)

   // write the movie in the queue //
   Queue.async {self.WriteMovieQueue(Layered)}
}

// write the movie in a queue //
func WriteMovieQueue(_ Layered:Bool)
{
   // start writing the video //
   VideoWriter!.startWriting()

   // start the session at time zero //
   VideoWriter!.startSession(atSourceTime:CMTime.zero)

   // if not for converting a layer to a movie //
   if !Layered
   {
      // write the movie info frame to the adaptor //
      Adaptor!.append(Graphics.ImageToBuffer(Queues.last!.GetBoard(Width, Index:Attributes.Inf
               .GetCount(), Info:true))!, withPresentationTime:CMTimeMake(value:0, timescale:10))
   }

   // loop over all the layer positions or the frame images //
   for Index in 0...(Layered ? MaxPosition : Attributes.Inf.GetCount())
   {
      // wait until the adaptor is ready for more media to be added //
      while !(Adaptor?.assetWriterInput.isReadyForMoreMediaData)! {}

      // append the pixel buffer with the frame and specified duration //
      Adaptor!.append(GetBuffer(Index)!, withPresentationTime:CMTimeMake(value:(Layered ? 0 : 50)
               + Int64(CGFloat(Index * 10) / MovieFPS), timescale:10))
   }
```

```
      // wait until the adaptor is ready for more media to be added //
      while !(Adaptor?.assetWriterInput.isReadyForMoreMediaData)! {}

      // mark the video input as finished //
      VidInput!.markAsFinished()

      // finish writing the video //
      VideoWriter!.finishWriting(completionHandler:{() -> Void in})

      // set the vidinput, videowriter, and adaptor to nil //
      (VidInput, VideoWriter, Adaptor) = (nil, nil, nil)

      // loop over all the queues to remove all the frames and pixel buffers //
      for TheQueue in Queues {TheQueue.Reset()}
   }
}

// the information values structure //
struct InfoValues
{
   // the string value //
   var VS:String = ""

   // the string value default //
   var VDS:String = ""

   // the string value label //
   var VLS:String = ""

   // the string value notation //
   var VNS:String = ""

   // the boolean value //
   var VB:Bool = false

   // the boolean value default //
   var VDB:Bool = false

   // the integer value //
   var VI:Int = 0

   // the integer value default //
   var VDI:Int = 0

   // the integer value minimum //
   var VMinInt:Int = 0

   // the integer value maximum //
   var VMaxInt:Int = 0
}

// the information keys structure //
struct InfoKeys
{
   // the analysis end key //
   var AnalysisEnd:Int = 0

   // the end key for the main information sequence stored //
   var End:Int = 0

   // the end key for the maximum amount of information that can be recovered //
   var InfoEnd:Int = 0

   // the keys initializer //
   init(_ TheEnd:Int, Inf:Int, Analy:Int) {(End, InfoEnd, AnalysisEnd) = (TheEnd, Inf, Analy)}

   // the keys initializer //
   init() {}
}

// an information item class //
class InfoItem
{
   // whether the information is valid or not //
   var Valid:Bool = true

   // the information values //
   var IVals:InfoValues = InfoValues()

   // the information values backup //
   var IValsBackup:InfoValues = InfoValues()

   // the information values saved //
   var IValsSaved:InfoValues = InfoValues()

   // the tag that describes the information item being stored //
   var Tag:String = ""

   // initializer for storing specified tag //
   init(_ Tag:String) {self.Tag = Tag}
}

// a class to hold information for keys with specified values //
class Information
{
   // the information keys //
   var Keys:InfoKeys = InfoKeys()

   // the backup information keys //
   var BackupKeys:InfoKeys = InfoKeys()

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

   // a shift in index values //
   var Shift:Int = 0

   // the user preferences //
   var Prefs:UserPreferences = UserPreferences()

   // the information dictionary //
   var Info:[String:InfoItem] = [:]

   // mark the info item as valid //
   func MarkValid(_ Tag:String, Key:Int) {Item(Tag, Key:Key)?.Valid = true}

   // mark the info item as invalid //
   func MarkInvalid(_ Tag:String, Key:Int) {Item(Tag, Key:Key)?.Valid = false}

   // update the string values //
   func Update(_ Tag:String, VSItems:[String:String])
   {
      // loop over all the string value items //
      for (Key, VS) in VSItems {Item(Tag, Key:Key)?.IVals.VS = VS}
   }

   // update the string value defaults //
   func Update(_ Tag:String, VDSItems:[String:String])
   {
      // loop over all the string value default items //
      for (Key, VDS) in VDSItems {Item(Tag, Key:Key)?.IVals.VDS = VDS}
   }
```

```
// update the string value labels //
func Update(_ Tag:String, VLSItems:[String:String])
{
   // loop over all the string value label items //
   for (Key, VLS) in VLSItems {Item(Tag, Key:Key)?.IVals.VLS = VLS}
}

// update the string value notations //
func Update(_ Tag:String, VNSItems:[Int:String])
{
   // loop over all the string value notation items //
   for (Key, VNS) in VNSItems {Item(Tag, Key:Key)?.IVals.VNS = VNS}
}

// update the boolean values //
func Update(_ Tag:String, VBItems:[String:Bool])
{
   // loop over all the boolean value items //
   for (Key, VB) in VBItems {Item(Tag, Key:Key)?.IVals.VB = VB}
}

// update the boolean value defaults //
func Update(_ Tag:String, VDBItems:[String:Bool])
{
   // loop over all the boolean value default items //
   for (Key, VDB) in VDBItems {Item(Tag, Key:Key)?.IVals.VDB = VDB}
}

// update the integer values //
func Update(_ Tag:String, VIItems:[String:Int])
{
   // loop over all the integer value items //
   for (Key, VI) in VIItems {Item(Tag, Key:Key)?.IVals.VI = VI}
}

// update the default, min, and max integer values //
func Update(_ Tag:String, VDefMinMaxIntItems:[String:[Int]])
{
   // loop over all the default, min, and max integer value items //
   for (Key, VDefMinMaxInt) in VDefMinMaxIntItems {if VDefMinMaxInt.count == 3
   {
      // set the integer value default //
      Item(Tag, Key:Key)?.IVals.VDI = VDefMinMaxInt[0]

      // set the integer value minimum //
      Item(Tag, Key:Key)?.IVals.VMinInt = VDefMinMaxInt[1]

      // set the integer value maximum //
      Item(Tag, Key:Key)?.IVals.VMaxInt = VDefMinMaxInt[2]
   }}
}

// compress the information to remove invalid elements //
func Compress(_ Tag:String, Start:Int, End:Int)
{
   // set the shift to zero //
   Shift = 0

   // loop over the information elements from start to end //
   for Index in Start..<End
   {
      // increment the index shift if the information element at the index is not valid //
      if !Item(Tag, Key:Index)!.Valid
      {
         // increment the shift index //
         Shift += 1
      }
      // if the information element at the index is valid and the shift is greater than zero //
      else if Shift > 0
      {
         // copy over the next valid information element to compress the information //
         Item(Tag, Key:Index - Shift)?.IVals = Vals(Index)
      }
   }
}

// compress the information elements //
func Compress(_ Tag:String)
{
   // compress the information elements from zero to the end //
   Compress(Tag, Start:0, End:Keys.End)

   // update the information keys to the new values //
   Keys = InfoKeys(Keys.End - Shift, Inf:Keys.End - Shift, Analy:Keys.End - Shift)
}

// compress the analysis information elements //
func CompressAnalysis(_ Tag:String)
{
   // compress the information elements from the end key to the analysis end key //
   Compress(Tag, Start:Keys.End, End:Keys.AnalysisEnd)

   // update the information keys to the new values //
   Keys = InfoKeys(Keys.End, Inf:Keys.End, Analy:Keys.AnalysisEnd - Shift)
}

// toggle the boolean value //
func ToggleVB(_ Tag:String, Key:String) {Item(Tag, Key:Key)?.IVals.VB = !Vals(Key).VB}

// turn off the boolean value //
func TurnOffVB(_ Tag:String, Key:String) {Item(Tag, Key:Key)?.IVals.VB = false}

// turn on the boolean value //
func TurnOnVB(_ Tag:String, Key:String) {Item(Tag, Key:Key)?.IVals.VB = true}

// the integer value as a string //
func IntStr(_ Key:String) -> String {return String(Vals(Key).VI)}

// get the character at the specified index //
func GetC(_ Key:String, Index:Int) -> String
{
   // return the character at the specified index //
   return String(Vals(Key).VS.prefix(Index + 1).last!)
}

// the integer value as an opacity percentage //
func Perc(_ Key:String) -> String {return " opacity is " + IntStr(Key) + "%"}

// the integer value as a cgfloat //
func VF(_ Key:String) -> CGFloat {return CGFloat(Vals(Key).VI)}

// the integer value as a fraction //
func Fraction(_ Frac:CGFloat, Key:String) -> CGFloat {return VF(Key) / Frac}

// return the info values or the default info values if info item doesn't exist //
func Vals(_ Key:String) -> InfoValues
{
   // return the info values or the default info values if info item doesn't exist //
   return Info[Key] == nil ? InfoValues() : Info[Key]!.IVals
}

// return the info values or the default info values if info item doesn't exist //
func Vals(_ Key:Int) -> InfoValues {return Vals(String(Key))}
```

```
// return the backup info values or the default info values if info item doesn't exist //
func ValsBackup(_ Key:String) -> InfoValues
{
   // return the backup info values or the default info values if info item doesn't exist //
   return Info[Key] == nil ? InfoValues() : Info[Key]!.IValsBackup
}

// return the backup info values or the default info values if info item doesn't exist //
func ValsBackup(_ Key:Int) -> InfoValues {return ValsBackup(String(Key))}

// return the saved info values or the default info values if info item doesn't exist //
func ValsSaved(_ Key:String) -> InfoValues
{
   // return the saved info values or the default info values if info item doesn't exist //
   return Info[Key] == nil ? InfoValues() : Info[Key]!.IValsSaved
}

// get the info item for the specified tag and key //
func Item(_ Tag:String, Key:String) -> InfoItem?
{
   // if the key doesn't exist, create it with the specified tag //
   if Info[Key] == nil {Info.updateValue(InfoItem(Tag), forKey:Key)}

   // return the info item for the specified tag and key //
   return Info[Key]
}

// get the info item for the specified tag and key //
func Item(_ Tag:String, Key:Int) -> InfoItem? {return Item(Tag, Key:String(Key))}

// add to the string value //
func AddTo(_ Tag:String, Key:String, AddOn:String) {Item(Tag, Key:Key)?.IVals.VS += AddOn}

// save values for the specified tag and key //
func SaveValue(_ Tag:String, Key:String) {Item(Tag, Key:Key)?.IValsSaved = Vals(Key)}

// save the values for the specified tags //
func SaveValues(_ Tags:[String])
{
   // loop over all the keys with the specified tags to save the values //
   for (Key, Value) in Info {if Tags.contains(Value.Tag) {SaveValue(Value.Tag, Key:Key)}}
}

// restore the saved value //
func RestoreSavedValue(_ Tag:String, Key:String) {Item(Tag, Key:Key)?.IVals = ValsSaved(Key)}

// restore the saved values for the specified tags //
func RestoreSavedValues(_ Tags:[String])
{
   // loop over all the keys with the specified tags to restore the saved values //
   for (Key, Value) in Info {if Tags.contains(Value.Tag) {RestoreSavedValue(Value.Tag, Key:Key)}}
}

// backup the information for the specified tags and the corresponding keys //
func BackupValues(_ Tags:[String], Tag:String)
{
   // loop over all the keys and values and backup //
   for (Key, Value) in Info
   {
      // if the specified tag matches //
      if Tags.contains(Value.Tag) {Item(Value.Tag, Key:Key)?.IValsBackup = Vals(Key)}
   }

   // backup the keys //
   BackupKeys = InfoKeys(Keys.End, Inf:Keys.InfoEnd, Analy:Keys.InfoEnd)

   // loop over the rest of the keys and backup //
   for Index in 0..<Keys.InfoEnd {Item(Tag, Key:Index)?.IValsBackup = Vals(Index)}
}

// restore the information for the specified tag and the corresponding keys //
func RestoreBackupValues(_ Tags:[String], Tag:String)
{
   // loop over all the keys and values //
   for (Key, Value) in Info
   {
      // if the specified tag matches, restore the backup values //
      if Tags.contains(Value.Tag) {Item(Value.Tag, Key:Key)?.IVals = ValsBackup(Key)}
   }

   // restore the backup keys //
   Keys = BackupKeys

   // loop over the rest of the keys and restore from backups //
   for Index in 0..<Keys.InfoEnd {Item(Tag, Key:Index)?.IVals = ValsBackup(Index)}
}

// reset the information for the specified tag and the keys to default values //
func VSToDefaults(_ Tags:[String])
{
   // loop over all the tags and keys //
   for (Key, Value) in Info
   {
      // if the specified tag matches, reset the string value to the string value default //
      if Tags.contains(Value.Tag) {Item(Value.Tag, Key:Key)?.IVals.VS = Vals(Key).VDS}
   }
}

// convert the string value to an integer //
func VSToNum(_ Key:String) -> Int {return Int(Vals(Key).VS) != nil ? Int(Vals(Key).VS)! : 0}

// clear the string value at the specified tag and key //
func ClearVS(_ Tag:String, Key:String) {Item(Tag, Key:Key)?.IVals.VS = ""}

// clear the values for all the keys with the specified tags //
func ClearVS(_ Tags:[String])
{
   // loop over all the keys and clear the values for the specified tags //
   for (Key, Value) in Info {if Tags.contains(Value.Tag) {ClearVS(Value.Tag, Key:Key)}}
}

// check to see if the value string at the specified tag and key is empty or not //
func VSIsEmpty(_ Key:String) -> Bool {return Vals(Key).VS.isEmpty}

// check to see if the value string label at the specified tag and key is empty or not //
func VLSIsEmpty(_ Key:Int) -> Bool {return Vals(Key).VLS.isEmpty}

// check to see if the integer value at specified key is equal to passed in value //
func VIIsEqual(_ Key:String, VI:Int) -> Bool {return Vals(Key).VI == VI}

// check to see if the value string at specified key is equal to passed in value //
func VSIsEqual(_ Key:String, VS:String) -> Bool {return Vals(Key).VS == VS}

// check to see if the value at the specified key is equal to the default value //
func VSIsDefault(_ Key:String) -> Bool {return Vals(Key).VS == Vals(Key).VDS}

// get the current count of the values //
func GetCount() -> Int {return Keys.End}

// get the current count including the analysis values //
func AnalysisCount() -> Int {return Keys.AnalysisEnd}

// parse the value string into components and store in the info dictionary //
func Parse(_ Tag:String, VSs:String, Separator:String)
{
   // set the analysis end key to the end of the main values //
   Keys.AnalysisEnd = Keys.End

   // loop over all the components in the values string //
   for VS in VSs.components(separatedBy:Separator)
   {
      // update the key value at the current info key //
      Update(Tag, VSItems:[String(Keys.AnalysisEnd):VS])

      // increment the analysis end key //
      Keys.AnalysisEnd += 1
   }
}

// parse the key values into components and store in the info dictionary //
func Parse(_ Tag:String, VSKey:String, VLSKey:String, VSSeparator:String, VLSSeparator:String)
{
   // parse the key value strings into components and store in the info dictionary //
   Parse(Tag, VSs:Vals(VSKey).VS, VLSs:Vals(VLSKey).VS, VSSeparator:VSSeparator, VLSSeparator:
        VLSSeparator)
}
```

```
   // parse the key value strings into components and store in the info dictionary //
   func Parse(_ Tag:String, VSs:String, VLSs:String, VSSeparator:String, VLSSeparator:String)
   {
      // set the end key to zero //
      Keys.End = 0

      // loop over all the components in the values string //
      for VS in VSs.components(separatedBy:VSSeparator)
      {
         // update the key value at the current info key //
         Update(Tag, VSItems:[String(Keys.End):VS])

         // increment the end key //
         Keys.End += 1
      }

      // update the information keys to new values and the index to zero //
      (Keys, Index) = (InfoKeys(Keys.End, Inf:Keys.End, Analy:Keys.End), 0)

      // loop over all the components in the values string //
      for VLS in VLSs.components(separatedBy:VLSSeparator)
      {
         // update the label value at the current info key //
         Update(Tag, VLSItems:[String(Index):VLS])

         // increment the index //
         Index += 1
      }
   }

   // get the preferences for the boolean value and value string //
   func GetPrefs(_ VBTags:[String], VITags:[String], VSTags:[String])
   {
      // loop over all the keys in the information dictionary //
      for (Key, Value) in Info
      {
         // if the tag matches //
         if VSTags.contains(Value.Tag)
         {
            // update the key from the saved preference value //
            Update(Value.Tag, VSItems:[Key:Prefs.PrefString(Key, Default:Vals(Key).VDS)])
         }
         // if the info item contains a tag in the status tags array //
         else if VBTags.contains(Value.Tag)
         {
            // update the preference bool for the specified key //
            Update(Value.Tag, VBItems:[Key:Prefs.PrefBool(Key, Default:Vals(Key).VDB)])
         }
         // if the info item contains a tag in the integer tags array //
         else if VITags.contains(Value.Tag)
         {
            // update the preference int for the specified key //
            Update(Value.Tag, VIItems:[Key:Prefs.PrefInt(Key, Default:Vals(Key).VDI, Min:Vals(Key)
                   .VMinInt, Max:Vals(Key).VMaxInt)])
         }
      }
   }

   // get the saved preferences from the app container //
   func GetPrefs(_ Tags:[String], Suffix:String)
   {
      // loop over all the keys in the information dictionary //
      for (Key, Value) in Info {if Tags.contains(Value.Tag)
      {
         // update the preference string for the specified suffix //
         Update(Value.Tag, VSItems:[Key:Prefs.PrefString(Key + Suffix, Default:Vals(Key).VDS)])
      }}
   }

   // concatenate the key values and labels for all the info items //
   func ConcatenateLists(_ Tag:String, VSKey:String, VLSKey:String)
   {
      // clear the values for this tag //
      ClearVS([Tag])

      // loop over all the keys //
      for Index in 0..<Keys.End
      {
         // concatenate the current info key value onto the end of the specified key value //
         AddTo(Tag, Key:VSKey, AddOn:Vals(Index).VS + "**********")

         // concatenate the current info key label value onto the end of the specified key value //
         AddTo(Tag, Key:VLSKey, AddOn:Vals(Index).VLS + "**********")
      }
   }

   // save the preferences for the boolean, string, and integer values //
   func SavePrefs(_ VBTags:[String], VITags:[String], VSTags:[String])
   {
      // loop over all the keys //
      for (Key, Value) in Info
      {
         // if the tag matches, save the boolean value //
         if VBTags.contains(Value.Tag) {Prefs.Save(Vals(Key).VB, Key:Key)}

         // if the tag matches, save the integer value // //
         if VITags.contains(Value.Tag) {Prefs.Save(Vals(Key).VI, Key:Key)}

         // if the tag matches, save the string value //
         if VSTags.contains(Value.Tag) {Prefs.Save(Vals(Key).VS, Key:Key)}
      }
   }

   // save the preferences for value strings //
   func SavePrefs(_ Tags:[String], Suffix:String)
   {
      // loop over all the keys //
      for (Key, Value) in Info
      {
         // if the tag matches, save the value string //
         if Tags.contains(Value.Tag) {Prefs.Save(Vals(Key).VS, Key:Key + Suffix)}
      }
   }

   // clear all keys in the dictionary by resetting the information keys //
   func ClearAllKeyIndices() {Keys = InfoKeys()}

   // save the subset of values //
   func SaveSubset(_ End:Int) {Keys = InfoKeys(End, Inf:End, Analy:End)}

   // save the next value string into the info dictionary for the specified tag //
   func SaveNextVS(_ Tag:String, VS:String)
   {
      // update the value string for the specified tag and key //
      Update(Tag, VSItems:[String(Keys.End):VS])

      // update the value string label for the specified tag and key //
      Update(Tag, VLSItems:[String(Keys.End):""])

      // update the information keys to the new values //
      Keys = InfoKeys(Keys.End + 1, Inf:Keys.End + 1, Analy:Keys.End + 1)
   }

   // save the notation for the specified tag //
   func SaveNotation(_ Tag:String, VNS:String) {Update(Tag, VNSItems:[Keys.End - 1:VNS])}

   // redo to the next value string in the info dictionary //
   func RedoToNextVS() -> Bool
   {
      // if the end key is less than the information key //
      if Keys.End < Keys.InfoEnd
      {
         // update the informations keys to the new values //
         Keys = InfoKeys(Keys.End + 1, Inf:Keys.InfoEnd, Analy:Keys.End + 1)

         // return that were able to redo to the next value string //
         return true
      }

      // return that were not able to redo to the next value string //
      return false
   }

   // undo to the previous value string in the info dictionary //
   func UndoToPreviousVS() -> Bool
   {
      // if the end key is greater than zero //
      if Keys.End > 0
      {
         // update the information keys to the new values //
         Keys = InfoKeys(Keys.End - 1, Inf:Keys.InfoEnd, Analy:Keys.End - 1)

         // return that were able to undo to the previous value string //
         return true
      }

      // return that were not able to undo to the previous value string //
      return false
   }

   // get the value strings as a list //
   func GetVSList(_ Tag:String) -> [String]
   {
      // initialize a list of values //
      var List:[String] = []

      // loop over each value up to and including the end value string and append to the list //
      for Index in 0..<Keys.End {List.append(Vals(Index).VS)}

      // return the list of values //
      return List
   }
}

// the game explorer view //
class BoardImageView:NSView
{
   // the cgimage to draw //
   var Image:CGImage!

   // update the cgimage //
   func Update(_ Image:CGImage) {self.Image = nil; self.Image = Image; display(bounds)}

   // the draw rect function //
   override func draw(_ Rect:CGRect)
   {
      // draw the image if it exists //
      if Image != nil {NSGraphicsContext.current?.cgContext.draw(Image, in:bounds)}
   }
}

// class for square label functions on the board //
class ASquareLabel
{
   // rank characters from 1 to 8 //
   let RankIndex:[Character:Int] = ["1":0, "2":1, "3":2, "4":3, "5":4, "6":5, "7":6, "8":7]

   // file characters from a to h //
   let FileIndex:[Character:Int] = ["a":0, "b":1, "c":2, "d":3, "e":4, "f":5, "g":6, "h":7]

   // rank indices from 0 to 7 //
   let RankString:[Int:String] = [0:"1", 1:"2", 2:"3", 3:"4", 4:"5", 5:"6", 6:"7", 7:"8"]

   // file indices from 0 to 7 //
   let FileString:[Int:String] = [0:"a", 1:"b", 2:"c", 3:"d", 4:"e", 5:"f", 6:"g", 7:"h"]

   // get the corresponding rank index from the rank character //
   func RankIndex(_ Rank:Character) -> Int {return RankIndex[Rank] == nil ? 0 : RankIndex[Rank]!}

   // get the corresponding file index from the file character //
   func FileIndex(_ File:Character) -> Int {return FileIndex[File] == nil ? 0 : FileIndex[File]!}

   // get the rank index as a string //
   func RankString(_ Rank:Int, Rotated:Bool) -> String
   {
      // if the board is rotated //
      return Rotated ? RankString(7 - Rank) : RankString(Rank)
   }

   // get the rank index as a string //
   func RankString(_ Rank:Int) -> String {return RankString[Rank] == nil ? "" : RankString[Rank]!}

   // get the file index as a string //
   func FileString(_ File:Int, Rotated:Bool) -> String
   {
      // if the board is rotated //
      return Rotated ? FileString(7 - File) : FileString(File)
   }

   // get the file index as a string //
   func FileString(_ File:Int) -> String {return FileString[File] == nil ? "" : FileString[File]!}

   // check if character is a valid board rank //
   func IsRank(_ Rank:Character) -> Bool {return RankIndex[Rank] != nil}

   // check if character is a valid board file //
   func IsFile(_ File:Character) -> Bool {return FileIndex[File] != nil}

   // get the square label for the specified point //
   func SquareLabel(_ Point:APoint) -> String {return FileString(Point.XI) + RankString(Point.YI)}

   // get a square location from a uci move string //
   func SquareFromUCI(_ UCIMove:String, Start:Bool, Dims:BoardDimensions, Rotated:Bool) -> APoint
   {
      // a board square location //
      let Point:APoint = APoint()

      // if need to get a start square and have at least two characters in the uci move //
      if Start && (UCIMove.count >= 2)
      {
         // set the start square board location //
         Point.Set(FileIndex(UCIMove.prefix(1).last!), Y:RankIndex(UCIMove.prefix(2).last!), Dims:
                   Dims, Rotated:Rotated)
      }
      // if need to get an end square and have at least 4 characters in the uci move //
      else if !Start && (UCIMove.count >= 4)
      {
         // set the end square board location //
         Point.Set(FileIndex(UCIMove.prefix(3).last!), Y:RankIndex(UCIMove.prefix(4).last!), Dims:
                   Dims, Rotated:Rotated)
      }

      // return the board square location //
      return Point
   }
}

// the moving piece layer //
class AMovingPiece:CALayer
{
   // if is an illegal move //
   var IllegalMove:Bool = false

   // the extracted piece promotion dictionary //
   let PromoteDict:[Character:Int] = ["n":2, "b":3, "r":4, "q":5]

   // the promote pawn to piece dictionary //
   let PromoteToDictionary:[Int:String] = [0:"q", 1:"r", 2:"b", 3:"n"]

   // a basic move animation //
   var MoveAnimation:CABasicAnimation = CABasicAnimation(keyPath:"position")

   // the square label functions //
   var SquareLabel:ASquareLabel = ASquareLabel()

   // the uci move string //
   var UCIMove:String = ""

   // the moving piece //
   var MovingPiece:Int = 0

   // the piece that can promote to if is a pawn promotion //
   var PromotionType:Int = 0

   // the current location of the piece moving //
   var Location:APoint = APoint()

   // the board dimensions //
   let Dims:BoardDimensions = BoardDimensions()

   // the board and piece compositor //
   var BPComp:BoardPieceCompositor!

   // set illegal move to false //
   func ResetIllegal() {IllegalMove = false}

   // set that move is illegal //
   func SetIllegal() {IllegalMove = true}

   // find if move is legal //
   func IsLegal() -> Bool {return !IllegalMove}

   // find if move is illegal //
   func IsIllegal() -> Bool {return IllegalMove}

   // if currently animating for position key //
   func Animating() -> Bool {return animation(forKey:"position") != nil}

   // check if the piece can't move //
   func PieceCantMove(_ Board:UnsafeMutableRawPointer) -> Bool
   {
      // return if the piece can't move //
      return !((IsWhite() && BoardIsWhite(Board)) || (IsBlack() && BoardIsBlack(Board)))
   }

   // check if have a castling move or not //
   func Castle(_ Move:String, Piece:Int) -> Bool {return (UCIMove == Move) && (Piece == MovingPiece)}

   // see if can promote the pawn to the selected piece //
   func PromoteTo(_ Board:UnsafeMutableRawPointer, ReturnCode:Int)
   {
      // return if piece is not in the promotion dictionary //
      if PromoteToDictionary[ReturnCode] == nil {return}

      // set the moving piece from the button return code //
      Set(BoardIsWhite(Board) ? 5 - Int(ReturnCode) : Int(ReturnCode) - 5)

      // update the last move on the move list to show a promotion occurred //
      UCIMove += PromoteToDictionary[ReturnCode]!
   }

   // prepare for the move animation //
   func PrepareAnimation(_ Start:APoint, End:APoint, Rate:Int)
   {
      // remove the animation effects //
      CATransaction.setDisableActions(true)

      // create the animation from and to values, position, and duration //
      (MoveAnimation.fromValue, MoveAnimation.toValue, position, MoveAnimation.duration) =
      (NSValue(point:IllegalMove ? End.GetCoord() : Start.GetCoord()), NSValue(point:IllegalMove ?
      Start.GetCoord() : End.GetCoord()), IllegalMove ? Start.GetCoord() : End.GetCoord(),
      IllegalMove ? 0.25 : CFTimeInterval(sqrt(pow(End.XF - Start.XF, 2) + pow(End.YF - Start.YF,
      2)) * CGFloat(101 - Rate) / 50))

      // set the moving piece contents //
      SetContents()

      // add the move animation to do //
      add(MoveAnimation, forKey:"position")
   }

   // set the moving piece image contents //
   func SetContents()
   {
      // remove the animation effects //
      CATransaction.setDisableActions(true)

      // set the moving piece image contents and unhide the piece //
      (contents, isHidden) = (BPComp.GetPieceImage(MovingPiece), false)
   }

   // if have pawn promotion, extract the piece type the pawn is promoting to //
   func ExtractPromotionTypeIfNeeded()
   {
      // extract a pawn promotion if needed //
      PromotionType = (PromoteDict[UCIMove.last!] != nil) ? ((UCIMove.prefix(4).last! == "1") ?
                      -PromoteDict[UCIMove.last!]! : PromoteDict[UCIMove.last!]!) : 0
   }

   // update the mouse location //
   func Update(_ MouseLocation:CGPoint)
   {
      // update the mouse location for a piece and if it is rotated or not //
      Location.Set(MouseLocation, Dims:Dims, Rotated:BPComp.Inf.Vals("RotateBoard").VB)
   }

   // get the square label for the current location //
   func GetSquareLabel() -> String {return SquareLabel.SquareLabel(Location)}

   // set the ucimove start square //
   func SetUCIMoveStartSquare() {UCIMove = SquareLabel.SquareLabel(Location)}

   // reset the moving piece and promotion type to zero //
   func Reset() {(MovingPiece, PromotionType) = (0, 0)}

   // update the current moving piece position //
   func UpdatePos() {CATransaction.setDisableActions(true); position = Location.GetCGPoint()}

   // update the location and position of the moving piece //
   func UpdateLocAndPos(_ Location:CGPoint) {Update(Location); UpdatePos()}

   // check if have a piece currently moving //
   func HaveMovingPiece() -> Bool {return MovingPiece != 0}

   // check if have a white pawn currently moving //
   func WhitePawnIsMoving() -> Bool {return MovingPiece == 1}

   // check if have a black pawn currently moving //
   func BlackPawnIsMoving() -> Bool {return MovingPiece == -1}

   // check if have no piece currently moving //
   func NoMovingPiece() -> Bool {return MovingPiece == 0}

   // check if piece currently moving is white //
   func IsWhite() -> Bool {return MovingPiece > 0}

   // check if piece currently moving is black //
   func IsBlack() -> Bool {return MovingPiece < 0}

   // check if currently placing an enpassant square in the position editor //
   func PlacingEnpassantSquare() -> Bool {return MovingPiece == 10}

   // check if the piece is a pawn and can promote to another piece //
   func PawnCanPromote() -> Bool
   {
      // return if piece is a pawn and can promote to another piece //
      return (WhitePawnIsMoving() && RankEquals(7)) || (BlackPawnIsMoving() && RankEquals(0))
   }

   // determine if the current rank is equal to the specified rank //
   func RankEquals(_ Rank:Int) -> Bool {return Location.YI == Rank}

   // save the uci move //
   func SaveUCIMove(_ UCIMove:String) {self.UCIMove = UCIMove}

   // set the piece currently moving //
   func Set(_ MovingPiece:Int) {self.MovingPiece = MovingPiece}

   // update to the promotion type //
   func UpdateToPromotionType()
   {
      // if not an illegal move and have a piece to promote to, update to the promoted piece //
      if !IllegalMove && (PromotionType != 0) {MovingPiece = PromotionType}
   }

   // hide the piece animating and update to the promoted type if needed //
   func HideAnimation() {UpdateToPromotionType(); isHidden = true}

   // unhide the moving piece //
   func UnHideMovingPiece() {UpdatePos(); SetContents()}

   // hide the moving piece layer contents //
   func HideMovingPiece(_ Point:CGPoint)
   {
      // update the location and position //
      UpdateLocAndPos(Point)

      // add the end square to the uci move and hide the moving piece layer contents //
      (UCIMove, isHidden) = (UCIMove + SquareLabel.FileString(Location.XI) + SquareLabel
      .RankString(Location.YI), true)
   }

   // initialize the moving piece attributes //
   func Initialize(_ View:NSView, BPComp:BoardPieceCompositor, Delegate:GreenChess)
   {
      // set delegate, board and piece compositor, hidden value, shadow opacity value, and bounds //
      (MoveAnimation.delegate, self.BPComp, isHidden, shadowOpacity, bounds) = (Delegate, BPComp,
      true, 0.5, CGRect(x:0, y:0, width:Dims.Square.Width, height:Dims.Square.Width))

      // add the moving piece layer to the main view //
      View.layer?.addSublayer(self)
   }
}

// the display image class //
class DImage
{
   // the date the display image was created //
   var Created:Date = Date()

   // the image to display to the screen //
   var Image:CGImage?

   // if the image to display is empty or not //
   var Empty:Bool = true

   // update the display image information //
   func Update(_ Created:Date, Image:CGImage)
   {
      // update the variables //
      (self.Created, self.Image, Empty) = (Created, Image, false)
   }
}

// the sketch image class //
class SketchImage
{
   // the first display image //
   var DImage1:DImage = DImage()

   // the second display image //
   var DImage2:DImage = DImage()

   // the selected image date to display //
   var Created:Date = Date()

   // if have an empty display image or not //
   func EmptyImage() -> Bool {return DImage1.Empty || DImage2.Empty}

   // update the corresponding display image //
   func Update(_ Image:CGImage)
   {
      // if the first display image is empty, update the first display image //
      if DImage1.Empty {DImage1.Update(Date(), Image:Image); return}

      // if the second display image is empty, update the second display image //
      if DImage2.Empty {DImage2.Update(Date(), Image:Image); return}
   }

   // if have a display image ready to display or not //
   func ReadyImage() -> Bool
   {
      // if have no empty image, save the earliest date of the created image //
      if !EmptyImage() {Created = min(DImage1.Created, DImage2.Created); return true}

      // if image not empty, save the creation date of the first display image and return true //
      if !DImage1.Empty {Created = DImage1.Created; return true}

      // if image not empty, save the creation date of the second display image and return true //
      if !DImage2.Empty {Created = DImage2.Created; return true}

      // return that there is no image ready to display //
      return false
   }

   // reset both images to empty //
   func ResetImages() {(DImage1.Empty, DImage2.Empty) = (true, true)}

   // update the screen //
   func UpdateScreen(_ Bounds:CGRect)
   {
      // draw the image in the specified bounds //
      NSGraphicsContext.current?.cgContext.draw((Created == DImage1.Created ? DImage1.Image :
                                                 DImage2.Image)!, in:Bounds, byTiling:false)

      // reset the image that matches the creation date to empty //
      if Created == DImage1.Created {DImage1.Empty = true} else {DImage2.Empty = true}
   }
}

// the class for a sketch path //
class SketchPath
{
   // the points in the path //
   var Points:[CGPoint] = []

   // the color of the path //
   var Color:CGColor = NSColor.yellow.cgColor

   // the path initializer //
   init(_ Color:CGColor) {self.Color = Color}
}

// the class for sketching a path //
class Sketching
{
   // the line segment to draw //
   var LineSegment:[CGPoint] = []

   // the mouse tracker //
   var Tracker:CursorTracker = CursorTracker()

   // the sketch images //
   var SketchImages:SketchImage = SketchImage()

   // the default color of yellow //
   var Color:Int = 309

   // the draw colors for the markers //
   var Colors:[Int:CGColor] = [304:NSColor.red.cgColor, 305:NSColor.green.cgColor, 306:NSColor.blue
                              .cgColor, 307:NSColor.cyan.cgColor, 308:NSColor.magenta.cgColor, 309:
                              NSColor.yellow.cgColor,310:NSColor.black.cgColor, 311:NSColor.white
                              .cgColor]

   // the background layer //
   var Background:CGLayer!

   // the diagram layer //
   var Diagram:CGLayer!

   // set the draw dispatch queue //
   let DrawQueue = DispatchQueue(label:"DrawThread", attributes:.concurrent)

   // set the display dispatch queue //
   let DisplayQueue = DispatchQueue(label:"DisplayThread", attributes:.concurrent)

   // the view //
   var View:NSView!

   // whether is drawing to the screen or not //
   var Displaying:Bool = false

   // the list of sketch paths //
   var Paths:[SketchPath] = []

   // the current sketch path //
   var Path:SketchPath = SketchPath(NSColor.yellow.cgColor)

   // the line segment end index //
   var SegmentEnd:Int = 0

   // if currently drawing a path //
   var Drawing:Bool = false

   // the screen view bounds //
   var Bounds:CGRect = CGRect()

   // the horizontal offset //
   var XOffset:CGFloat = 0

   // the context //
   var Context:CGContext!

   // the core graphics functions //
   let Graphics:CoreGraphics = CoreGraphics()

   // get the snapshot image //
   func Snapshot() -> CGImage {Refresh(false); return Context.makeImage()!}

   // update the color and diagram cursor to the selected value //
   func UpdateColor(_ Color:String)
   {
      // set the selected color with a default of yellow if selected color doesn't exist //
      self.Color = (304...311).contains(Int(Color)!) ? Int(Color)! : 309

      // update the cursor to the current icon for specified color //
      Tracker.UpdateCursor(Color)
   }

   // clear the information used //
   func ClearInfo()
   {
      // remove all the paths //
      Paths.removeAll()

      // update the path color to yellow //
      UpdateColor("309")

      // clear the diagram, background, and visible context layers //
      (Diagram, Background, Context) = (Graphics.CreateLayer(1, Height:1), Graphics.CreateLayer(1,
      Height:1), Graphics.CreateContext(1, Height:1))
   }

   // set the background layer //
   func SetBackground(_ BackgroundImage:CGImage)
   {
      // draw the background image into the context //
      Background.context?.draw(BackgroundImage, in:CGRect(x:0, y:0, width:BackgroundImage.width,
                               height:BackgroundImage.height), byTiling:false)
   }

   // update the horizontal layer offset //
   func UpdateOffset(_ XOffset:CGFloat)
   {
      // update the x offset value //
      self.XOffset = XOffset

      // if the diagram layer exists, draw it into the full context at the specified x offset //
      if Diagram != nil {Context.draw(Diagram, at:CGPoint(x:XOffset, y:0))}

      // reset the sketch images to empty //
      SketchImages.ResetImages()

      // update the sketch image //
      SketchImages.Update(Context.makeImage()!)

      // display the sketch image in the specified bounds if there is one available to draw //
      if SketchImages.ReadyImage() {Displaying = true; View.display(Bounds)}
   }

   // refresh the layer context //
   func Refresh(_ Clear:Bool)
   {
      // if need to clear the diagram, remove all the diagram paths //
      if Clear {Paths.removeAll()}

      // draw the background layer into the diagram layer at the origin if it exists //
      if Background != nil {Diagram.context?.draw(Background, at:CGPoint())}

      // loop over all the draw paths //
      for ThePath in Paths
      {
         // begin the path //
         Diagram.context?.beginPath()

         // set the stroke color //
         Diagram.context?.setStrokeColor(ThePath.Color)

         // add the path between the points //
         Diagram.context?.addLines(between:ThePath.Points)

         // stroke the current path //
         Diagram.context?.strokePath()
      }

      // update the horizontal layer offset //
      UpdateOffset(XOffset)
   }

   // begin the current path //
   func BeginPath()
   {
      // reset the sketch images to empty //
      SketchImages.ResetImages()

      // set the path, segment indices, if drawing, and if displaying //
      (Path, SegmentEnd, Drawing, Displaying) = (SketchPath(Colors[Color] != nil ? Colors[Color]! :
      Colors[309]!), 0, true, false)

      // begin the path //
      Diagram.context?.beginPath()

      // set the diagram context color to that of the current path //
      Diagram.context?.setStrokeColor(Path.Color)

      // start the mouse tracker //
      Tracker.Start(XOffset)

      // start the display thread //
      DisplayQueue.async {self.DisplayThread()}

      // start the draw thread //
      DrawQueue.async {self.DrawThread()}

      // start tracking mouse events //
      View.window?.trackEvents(matching:[.leftMouseDragged, .leftMouseUp], timeout:NSEvent
           .foreverDuration, mode:RunLoop.Mode.eventTracking, handler:{(Event, Done) in guard let
           Event = Event else {return}; if Event.type == .leftMouseUp {Done.pointee = true}})

      // reset displaying, tracking, and drawing to false //
      (Displaying, Tracker.Tracking, Drawing) = (false, false, false)

      // if there are at least two points in the path //
      if Tracker.Points.count > 1
      {
         // add the mouse tracking points to the path points //
         Path.Points.append(contentsOf:Tracker.Points)

         // append the path to the paths //
         Paths.append(Path)
      }
   }

   // the display thread //
   func DisplayThread()
   {
      // while currently drawing //
      while Drawing
      {
         // sleep the thread for a small amount of time //
         Thread.sleep(forTimeInterval:0.000001)

         // run on the main queue //
         DispatchQueue.main.async {if !self.Displaying {if self.SketchImages.ReadyImage()
         {
            // set that currently displaying image to the screen //
            self.Displaying = true

            // display the image to the screen on the main thread //
            self.View.display(self.Bounds)
         }}}
      }
   }

   // the thread to draw a path //
   func DrawThread()
   {
      // while currently drawing and empty image to fill and have new line segment to draw //
      while Drawing
      {
         // sleep for a small amount of time //
         Thread.sleep(forTimeInterval:0.000001)

         // if there is a sketch image empty and have additional points to draw in the path //
         if SketchImages.EmptyImage() && (Tracker.Points.count > SegmentEnd + 1)
         {
            // set the end line segment index to draw //
            SegmentEnd = Tracker.Points.count - 1

            // get the last 100 points in the points tracker //
            LineSegment = Tracker.Points.suffix(100)

            // if have at least two points in the line segment //
            if LineSegment.count > 1
            {
               // begin the draw path //
               Diagram.context?.beginPath()

               // add the path segment to the diagram layer //
               Diagram.context?.addLines(between:LineSegment)

               // stroke the path to the diagram layer //
               Diagram.context?.strokePath()

               // if diagram layer exists, draw diagram into context at specified offset point //
               if Diagram != nil {Context.draw(Diagram, at:CGPoint(x:XOffset, y:0))}

               // update the sketch image //
               SketchImages.Update(Context.makeImage()!)
            }
         }
      }
   }

   // remove the last path if there is one and refresh the layer //
   func UndoPath() {_ = Paths.popLast(); Refresh(false)}

   // initialize the sketch layer //
   func Initialize(_ View:NSView, Width:CGFloat, Height:CGFloat, Length:CGFloat, YOffset:CGFloat,
                   Margin:CGFloat)
   {
      // set all the diagram values //
      (self.View, XOffset, Bounds, Background, Diagram, Context) = (View, 0, CGRect(x:0, y:0, width:
      Width + 0.5, height:Height + 0.5), Graphics.CreateLayer(max(Width, Length), Height:Height),
      Graphics.CreateLayer(max(Width, Length), Height:Height), Graphics.CreateContext(Width, Height:
      Height))

      // initialize the mouse tracker //
      Tracker.Initialize(View, Margin:Margin, Bounds:Bounds, YOffset:YOffset)

      // set the line width //
      Diagram.context?.setLineWidth(6)

      // set the line flatness //
      Diagram.context?.setFlatness(0.1)

      // set the line join to round //
      Diagram.context?.setLineJoin(CGLineJoin.round)

      // set the line cap style to round //
      Diagram.context?.setLineCap(CGLineCap.round)

      // set the background fill color to black //
      Background.context?.setFillColor(NSColor.black.cgColor)

      // fill the background context with black //
      Background.context?.fill(CGRect(x:0, y:0, width:max(Width, Length), height:Height))
   }
}

// class for getting a string //
class GetString
{
   // get the string from the text field //
   func FromTextField(_ 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)
   }
}

// the game sound effects object //
class GameSoundEffects:NSObject
{
   // a game sound //
   var GameSound:NSSound = NSSound()

   // the sound volumes //
   let SoundVolumes:[Float] = [1, 0.25, 0.4, 0.1, 0.25, 0.3, 0.4, 0.3, 0.7, 0.7, 0.3]

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

   // array of all the game sounds //
   var GameSounds:[NSSound] = []

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

   // the sound dictionary //
   let SoundDict:[String:Int] = ["thump":0, "wrongsquare":1, "pop":2, "click":3, "clear":4,
                                 "warning":5, "switch":6, "buttonclick":7, "camera":8, "marker":9,
                                 "movie":10]

   // the sound queue //
   let SoundQueue = DispatchQueue(label:"SoundThread", attributes:.concurrent)

   // play the game sound in a thread if sound is turned on //
   func Play(_ Sound:String) {if !SoundOff {SoundQueue.async {self.PlaySound(Sound)}}}

   // play a game sound //
   func PlaySound(_ Sound:String)
   {
      // return if the sound doesn't exist //
      if SoundDict[Sound] == nil {return}

      // stop the current game sound //
      GameSound.stop()

      // set the game sound selection //
      GameSound = GameSounds[SoundDict[Sound]!]

      // set the game sound volume //
      GameSound.volume = SoundVolumes[SoundDict[Sound]!]

      // play the game sound //
      GameSound.play()
   }

   // override the default initializer to add the game sounds //
   override init() {for TheSound in SoundNames {GameSounds.append(NSSound(named:TheSound)!)}}
}

// class for the pgn tags controls //
class PGNTagsController:NSObject
{
   // the information dictionary //
   var Inf:Information = Information()

   // the game sounds //
   @IBOutlet var GameSounds:GameSoundEffects!

   // the date formatter //
   var TheDateFormat:DateFormatter = DateFormatter()

   // the text field for the title of the chess event or tournament //
   @IBOutlet var EventTitleTextField:NSTextField!

   // the text field for the geographic location of the game site //
   @IBOutlet var SiteLocationTextField:NSTextField!

   // the date picker for date the game was played on //
   @IBOutlet var GameDate:NSDatePicker!

   // the text field for the current game round being played //
   @IBOutlet var RoundNumberTextField:NSTextField!

   // the text field for the name of the person playing as white //
   @IBOutlet var WhiteNameTextField:NSTextField!

   // the text field for the name of the person playing as black //
   @IBOutlet var BlackNameTextField:NSTextField!

   // the button to click on if white won the game //
   @IBOutlet var WhiteWon:NSButton!
```

```
   // the button to click on if black won the game //
   @IBOutlet var BlackWon:NSButton!

   // the button to click on if the game ended in a draw //
   @IBOutlet var DrawButton:NSButton!

   // the text field for the time control of the game //
   @IBOutlet var TimeControlTextField: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 text field for the eco of the game //
   @IBOutlet var ECOTextField: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!

   // get the date game was played //
   func GetDateGamePlayed() -> String {return TheFormat.string(from:GameDate.dateValue)}

   // get the current date //
   func GetCurrentDate() -> String {return TheFormat.string(from:Date())}

   // toggle the key //
   func ToggleKey(_ Key:String) {GameSounds.Play("click"); Inf.ToggleVB("VB", Key:Key)}

   // toggle the result //
   func ToggleRes(_ Key:String, Off:[String], Buttons:[NSButton])
   {
      // toggle the key //
      ToggleKey(Key)

      // turn the specified button states off //
      for TheButton in Buttons {TheButton.state = NSControl.StateValue.off}

      // set specified keys to false //
      for TheKey in Off {Inf.Update("VB", VBItems:[TheKey:false])}
   }

   // change state of whether white won the game or not //
   @IBAction func WhiteWonAction(_:AnyObject)
   {
      // toggle the result //
      ToggleRes("WhiteWonGame", Off:["DrawnGame", "BlackWonGame"], Buttons:[BlackWon, DrawButton])
   }

   // change state of whether black won the game or not //
   @IBAction func BlackWonButtonAction(_:AnyObject)
   {
      // toggle the result //
      ToggleRes("BlackWonGame", Off:["DrawnGame", "WhiteWonGame"], Buttons:[WhiteWon, DrawButton])
   }

   // change the state of whether game was drawn or not //
   @IBAction func DrawButtonAction(_:AnyObject)
   {
      // toggle the result //
      ToggleRes("DrawnGame", Off:["WhiteWonGame", "BlackWonGame"], Buttons:[WhiteWon, BlackWon])
   }

   // initialize the pgn tags controls //
   func Initialize()
   {
      // set drawn game, white won game, and black won game to false //
      Inf.Update("VB", VBItems:["WhiteWonGame":false, "BlackWonGame":false, "DrawnGame":false])

      // set the specified buttons to off //
      for Button in [WhiteWon, BlackWon, DrawButton] {Button.state = NSControl.StateValue.off}

      // specify the date format to use and set the date to the current date //
      (TheDateFormat.dateFormat, GameDate.dateValue) = ("yyyy.MM.dd", Date())

      // loop over the specified text fields //
      for TheField in [WhiteNameTextField, BlackNameTextField, SiteLocationTextField,
                       EventTitleTextField, RoundNumberTextField, TimeControlTextField,
                       OpeningTextField, SectionTextField, BoardNumberTextField, WhiteEloTextField,
                       BlackEloTextField, WhiteUSCFTextField, BlackUSCFTextField,
                       AnnotatorTextField, ECOTextField]
      {
         // set the text field string value to empty //
         TheField!.stringValue = ""
      }
   }

   // update the pgn tags controls from the game data //
   func UpdateGameInfoFromGameData(_ GameData:UnsafeMutableRawPointer)
   {
      // update all the text fields //
      (WhiteNameTextField.stringValue, BlackNameTextField.stringValue, TimeControlTextField
      .stringValue, OpeningTextField.stringValue, SectionTextField.stringValue, BoardNumberTextField
      .stringValue, BlackEloTextField.stringValue, WhiteEloTextField.stringValue, WhiteUSCFTextField
      .stringValue, BlackUSCFTextField.stringValue, AnnotatorTextField.stringValue, ECOTextField
      .stringValue, SiteLocationTextField.stringValue, EventTitleTextField.stringValue,
      RoundNumberTextField.stringValue) = (String(validatingUTF8:DataWhitePlayer(GameData))!,
      String(validatingUTF8:DataBlackPlayer(GameData))!, String(validatingUTF8:
      DataGetTimeControl(GameData))!, String(validatingUTF8:DataGetOpening(GameData))!,
      String(validatingUTF8:DataGetSection(GameData))!, String(validatingUTF8:
      DataGetBoardNumber(GameData))!, String(validatingUTF8:DataGetBlackElo(GameData))!,
      String(validatingUTF8:DataGetWhiteElo(GameData))!, String(validatingUTF8:
      DataGetWhiteUSCF(GameData))!, String(validatingUTF8:DataGetBlackUSCF(GameData))!,
      String(validatingUTF8:DataGetAnnotator(GameData))!, String(validatingUTF8:
      DataGetECO(GameData))!, String(validatingUTF8:DataGetSite(GameData))!,
      String(validatingUTF8:DataGetEvent(GameData))!, String(validatingUTF8:
      DataGetRound(GameData))!)

      // switch over the game result value //
      switch String(validatingUTF8:DataGetResult(GameData))!
      {
         // turn on the white won button //
         case "1-0" : WhiteWon.state = NSControl.StateValue.on

                      // set that white won the game //
                      Inf.TurnOnVB("VB", Key:"WhiteWonGame")

         // turn on the black won button //
         case "0-1" : BlackWon.state = NSControl.StateValue.on

                      // set that black won the game //
                      Inf.TurnOnVB("VB", Key:"BlackWonGame")

         // turn on the draw button //
         case "1/2-1/2" : DrawButton.state = NSControl.StateValue.on

                          // set that the game was drawn //
                          Inf.TurnOnVB("VB", Key:"DrawnGame")

         // the default case //
         default : break
      }

      // exit if the date format is invalid //
      if TheDateFormat.date(from:String(validatingUTF8:DataGetDate(GameData))!) == nil {return}

      // update the date picker calendar //
      GameDate.dateValue = TheDateFormat.date(from:String(validatingUTF8:DataGetDate(GameData))!)!
   }
}

// the preferences controller class //
class PreferenceController:NSObject
{
   // the movie size slider //
   @IBOutlet var MovieSizeSlider:NSSlider!

   // the movie size view //
   @IBOutlet var MovieSizeView:NSTextField!

   // the gallery animation rate slider //
   @IBOutlet var GalleryAnimationRateSlider:NSSlider!

   // the gallery animation rate view //
   @IBOutlet var GalleryAnimationRateView:NSTextField!

   // the game gallery grid rows button //
   @IBOutlet var GridRowsButton:NSPopUpButton!

   // the button to click on if want to multithread the uci engine //
   @IBOutlet var MultithreadUCIEngineButton:NSButton!

   // the stockfish analysis button //
   @IBOutlet var ShowStockfishAnalysisButton:NSButton!

   // the uci engine difficulty slider //
   @IBOutlet var UCIEngineDifficultySlider:NSSlider!

   // the text field for the uci engine difficulty //
   @IBOutlet var UCIEngineDifficultyView:NSTextField!

   // the uci engine hash slider //
   @IBOutlet var UCIEngineHashSlider:NSSlider!

   // the text field for the uci engine hash size //
   @IBOutlet var UCIEngineHashView:NSTextField!

   // the display illustrations button //
   @IBOutlet var DisplayIllustrationsButton:NSButton!

   // the illustration size slider //
   @IBOutlet var IllustrationsSizeSlider:NSSlider!

   // the illustration size view //
   @IBOutlet var IllustrationsSizeView:NSTextField!

   // the diagram preview //
   @IBOutlet var DiagramPreview:NSImageView!

   // the animation rate slider //
   @IBOutlet var AnimationRateSlider:NSSlider!

   // the text field for the animation rate of the pieces moving //
   @IBOutlet var AnimationRateView:NSTextField!

   // the search time slider //
   @IBOutlet var SearchTimeSlider:NSSlider!

   // the text field for the time the chess engine will search for //
   @IBOutlet var SearchTimeView:NSTextField!

   // the display game clock button //
   @IBOutlet var DisplayClockButton:NSButton!

   // the game clock color text field //
   @IBOutlet var GameClockColorField:NSTextField!

   // the game clock border color view //
   @IBOutlet var ClockColorView:ColorsView!

   // the game clock background color text field //
   @IBOutlet var GameClockBackgroundColorField:NSTextField!

   // the game clock background color view //
   @IBOutlet var ClockBackgroundColorView:ColorsView!

   // the clock background visibility slider //
   @IBOutlet var ClockBackgroundVisibilitySlider:NSSlider!

   // the clock background opacity view //
   @IBOutlet var ClockBackgroundOpacityView:NSTextField!

   // the time clock preview //
   @IBOutlet var TimeClockPreview:TimeClockDisplay!

   // the window texture opacity slider //
   @IBOutlet var WindowTextureOpacitySlider:NSSlider!

   // the window texture opacity view //
   @IBOutlet var WindowTextureOpacityView:NSTextField!

   // the window background colors text field //
   @IBOutlet var WindowColorsBackgroundField:NSTextField!

   // the window background colors view //
   @IBOutlet var WindowBackgroundColorView:ColorsView!

   // the window color preview //
   @IBOutlet var WindowColorPreview:NSImageView!

   // the board labels text field //
   @IBOutlet var BoardLabelsField:NSTextField!

   // the board labels font button //
   @IBOutlet var BoardLabelsFontButton:NSPopUpButton!

   // the board labels bold button //
   @IBOutlet var BoardLabelsBoldButton:NSPopUpButton!

   // the board labels font sizes button //
   @IBOutlet var BoardLabelsSizesButton:NSPopUpButton!

   // the board labels color view //
   @IBOutlet var BoardLabelsColorView:ColorsView!

   // the game info text field //
   @IBOutlet var GameInfoField:NSTextField!

   // the game info font button //
   @IBOutlet var GameInfoFontButton:NSPopUpButton!

   // the game info bold button //
   @IBOutlet var GameInfoBoldButton:NSPopUpButton!

   // the game info sizes button //
   @IBOutlet var GameInfoSizesButton:NSPopUpButton!

   // the game info color view //
   @IBOutlet var GameInfoColorView:ColorsView!

   // the move list text field //
   @IBOutlet var MoveListField:NSTextField!

   // the move list font button //
   @IBOutlet var MoveListFontButton:NSPopUpButton!

   // the move list bold button //
   @IBOutlet var MoveListBoldButton:NSPopUpButton!

   // the move list font sizes button //
   @IBOutlet var MoveListSizesButton:NSPopUpButton!

   // the move list color view //
   @IBOutlet var MoveListColorView:ColorsView!

   // the game list text field //
   @IBOutlet var GameListField:NSTextField!

   // the game list font button //
   @IBOutlet var GameListFontButton:NSPopUpButton!

   // the game list bold button //
   @IBOutlet var GameListBoldButton:NSPopUpButton!

   // the game list sizes button //
   @IBOutlet var GameListSizesButton:NSPopUpButton!

   // the game list color view //
   @IBOutlet var GameListColorView:ColorsView!

   // the board preview view //
   @IBOutlet var BoardPreview:NSImageView!

   // the board textures button //
   @IBOutlet var BoardTexturesButton:NSPopUpButton!

   // the piece set button //
   @IBOutlet var PieceSetButton:NSPopUpButton!

   // the autosave game data option button //
   @IBOutlet var AutosaveGameButton:NSButton!

   // the game sounds button //
   @IBOutlet var GameSoundsButton:NSButton!

   // the button to click on if want to use the encyclopedia of chess openings //
   @IBOutlet var EcoBookButton:NSButton!

   // the button to click on if want to highlight the last move made during a game //
   @IBOutlet var HighlightLastMove:NSButton!

   // the button to click on if want to highlight pieces which can be captured //
   @IBOutlet var HighlightCaptures:NSButton!

   // the button to click on if want to highlight pieces which are threatened //
   @IBOutlet var HighlightThreats:NSButton!

   // the move by selecting square button //
   @IBOutlet var MoveBySelectingSquares:NSButton!

   // the display captured pieces button //
   @IBOutlet var DisplayCaptures:NSButton!

   // the rotate board button //
   @IBOutlet var RotateBoardButton:NSButton!

   // the chess board border texture view //
   @IBOutlet var BorderTextureView:NSTextField!

   // the chess board border texture slider //
   @IBOutlet var BorderTextureSlider:NSSlider!

   // the chess board border visibility view //
   @IBOutlet var BorderVisibilityView:NSTextField!

   // the chess board border visibility slider //
   @IBOutlet var BorderVisibilitySlider:NSSlider!

   // the square border view //
   @IBOutlet var SquareBorderView:NSTextField!

   // the square border slider //
   @IBOutlet var SquareBorderOpacitySlider:NSSlider!

   // the board roughness view //
   @IBOutlet var BoardRoughnessView:NSTextField!

   // the board roughness slider //
   @IBOutlet var BoardRoughnessSlider:NSSlider!

   // the board tint color view //
   @IBOutlet var BoardTintColorView:NSTextField!

   // the board tint color slider //
   @IBOutlet var BoardTintColorSlider:NSSlider!

   // the board tint visibility view //
   @IBOutlet var BoardTintVisibilityView:NSTextField!

   // the board tint visibility slider //
   @IBOutlet var BoardTintVisibilitySlider:NSSlider!

   // the preferences toolbar //
   @IBOutlet var PreferencesToolbar:NSToolbar!

   // the preference controls dictionary //
   var PrefControls:[Int:Controls] = [:]

   // the board preference controls //
   var BoardControls:Controls = Controls()

   // the game clock preference controls //
   var ClockControls:Controls = Controls()

   // the preference stockfish controls //
   var StockfishControls:Controls = Controls()

   // the preference font controls //
   var FontControls:Controls = Controls()

   // the preference background controls //
   var BackgroundControls:Controls = Controls()

   // the preference speed controls //
   var SpeedControls:Controls = Controls()

   // the preference diagram controls //
   var DiagramControls:Controls = Controls()

   // the preference gallery controls //
   var GalleryControls:Controls = Controls()

   // the preference movie controls //
   var MovieControls:Controls = Controls()

   // get the updated control state //
   func UpdatedState(_ State:Bool) -> NSControl.StateValue
   {
      // return the updated control state //
      return NSControl.StateValue(rawValue:Bool.ToInt(State))
   }

   // initialize the preference controls //
   func Initialize()
   {
      // set the color view identifiers and controls //
      (BoardLabelsColorView.Identifier, ClockColorView.Identifier, ClockBackgroundColorView
      .Identifier, GameInfoColorView.Identifier, MoveListColorView.Identifier, GameListColorView
      .Identifier, WindowBackgroundColorView.Identifier, PrefControls) = (0, 1, 2, 3, 4, 5, 6,
      [20:BoardControls, 21:BoardControls, 22:FontControls, 23:ClockControls, 24:
      SpeedControls, 25:ClockControls, 26:MovieControls, 27:GalleryControls, 28:DiagramControls])

      // initialize the board controls //
      BoardControls.Initialize([BoardPreview, BoardTexturesButton, PieceSetButton,
                     AutosaveGameButton, GameSoundsButton, EcoBookButton, HighlightLastMove,
                     HighlightCaptures, HighlightThreats, MoveBySelectingSquares, DisplayCaptures,
                     RotateBoardButton, BorderTextureView, BorderTextureSlider, BorderVisibilityView,
                     BorderVisibilitySlider, SquareBorderView, SquareBorderOpacitySlider,
                     BoardRoughnessView, BoardRoughnessSlider, BoardTintColorView,
                     BoardTintColorSlider, BoardTintVisibilityView, BoardTintVisibilitySlider],
                     Frames:[[30, 215, 400, 400], [439, 590, 200, 25], [439, 550, 200, 25], [440,
                     505, 215, 25], [440, 475, 215, 25], [440, 445, 215, 25], [440, 415, 215, 25],
                     [440, 385, 215, 25], [440, 330, 215, 25], [440, 295, 215, 25], [440, 260, 215,
                     25], [440, 225, 215, 25], [440, 164, 215, 25], [30, 165, 400, 25], [440, 134,
                     215, 25], [30, 135, 400, 25], [440, 104, 215, 25], [30, 105, 400, 25], [440, 74,
                     215, 25], [30, 75, 400, 25], [440, 44, 215, 25], [30, 45, 400, 25], [440, 14,
                     215, 25], [30, 15, 400, 25]])

      // initialize the font controls //
      FontControls.Initialize([BoardLabelsField, BoardLabelsFontButton, BoardLabelsBoldButton,
                     BoardLabelsSizesButton, BoardLabelsColorView, GameInfoField, GameInfoFontButton,
                     GameInfoBoldButton, GameInfoSizesButton, GameInfoColorView, MoveListField,
                     MoveListFontButton, MoveListBoldButton, MoveListSizesButton, MoveListColorView,
                     GameListField, GameListFontButton, GameListBoldButton, GameListSizesButton,
                     GameListColorView], Frames:[[230, 595, 200, 25], [230, 570, 200, 25], [230, 545,
                     200, 25], [230, 520, 200, 25], [30, 480, 600, 40], [230, 445, 200, 25], [230,
                     420, 200, 25], [230, 395, 200, 25], [230, 370, 200, 25], [30, 330, 600, 40],
                     [230, 295, 200, 25], [230, 270, 200, 25], [230, 245, 200, 25], [230, 220, 200,
                     25], [30, 180, 600, 40], [230, 145, 200, 25], [230, 120, 200, 25], [230, 95, 200,
                     25], [230, 70, 200, 25], [30, 30, 600, 40]])

      // initialize the background controls //
      BackgroundControls.Initialize([WindowColorPreview, WindowColorsBackgroundField,
                     WindowBackgroundColorView, WindowTextureOpacitySlider,
                     WindowTextureOpacityView], Frames:[[230, 420, 200, 200], [230, 330, 200,
                     25], [30, 290, 600, 40], [30, 205, 600, 25], [30, 180, 600, 25]])

      // initialize the clock controls //
      ClockControls.Initialize([TimeClockPreview, DisplayClockButton, GameClockColorField,
                     ClockColorView, GameClockBackgroundColorField, ClockBackgroundColorView,
                     ClockBackgroundVisibilitySlider, ClockBackgroundOpacityView], Frames:[[280, 515,
                     300, 100], [30, 570, 200, 25], [230, 470, 600, 40], [30, 390, 600, 40], [230,
                     330, 200, 25], [30, 270, 600, 40], [30, 180, 600, 25], [30, 155, 600, 25]])

      // initialize the speed controls //
      SpeedControls.Initialize([AnimationRateSlider, AnimationRateView, SearchTimeSlider,
                     SearchTimeView], Frames:[[30, 595, 600, 25], [30, 570, 600, 25], [30, 505, 600,
                     25], [30, 480, 600, 25]])

      // initialize the diagram controls //
      DiagramControls.Initialize([DiagramPreview, DisplayIllustrationsButton,
                     IllustrationsSizeSlider, IllustrationsSizeView], Frames:[[230, 520, 250, 250],
                     [30, 480, 300, 25], [30, 270, 600, 40], [30, 225, 600, 25]])

      // initialize the stockfish controls //
      StockfishControls.Initialize([UCIEngineDifficultySlider, UCIEngineDifficultyView,
                     UCIEngineHashSlider, UCIEngineHashView, MultithreadUCIEngineButton,
                     ShowStockfishAnalysisButton], Frames:[[30, 595, 600, 25], [30, 570, 600,
                     25], [30, 505, 600, 25], [30, 480, 600, 40], [30, 390, 200, 25], [30, 300,
                     200, 25]])

      // initialize the gallery controls //
      GalleryControls.Initialize([GridRowsButton, GalleryAnimationRateSlider,
                     GalleryAnimationRateView], Frames:[[170, 550, 320, 25], [30, 505, 600, 25],
                     [30, 480, 600, 25]])

      // initialize the movie controls //
      MovieControls.Initialize([MovieSizeSlider, MovieSizeView], Frames:[[30, 505, 600, 25],
                     [30, 480, 600, 25]])
   }
}

// the board and piece compositor class //
class BoardPieceCompositor:NSObject
{
   // the information dictionary //
   var Inf:Information = Information()

   // square label functions //
   var SquareLabel:ASquareLabel = ASquareLabel()

   // the preference controls //
   @IBOutlet var Ctr:PreferenceController!

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

   // the game sounds //
   @IBOutlet var GameSounds:GameSoundEffects!

   // the chess board data //
   var BPData:ChessBoardData = ChessBoardData()

   // the window background texture //
   var WindowTexture:WindowBackgroundTexture = WindowBackgroundTexture()

   // a board label //
   var BoardLabel:NSAttributedString = NSAttributedString()

   // if have a selected square or not //
   var SelectedSquare:Bool = false

   // the chess board images //
   var BoardImages:ChessBoardImages = ChessBoardImages()

   // the chess piece images //
   var PieceImages:ChessPieceImages = ChessPieceImages()

   // the board square highlights //
   var Highlights:BoardHighlightSquares = BoardHighlightSquares()

   // the board dimensions //
   let Dims:BoardDimensions = BoardDimensions()

   // the board roughness opacity //
   var BoardRoughnessOpacity:CGFloat = 0

   // the tint color opacity //
   var TintColorOpacity:CGFloat = 0

   // the board tint hues //
   var BoardTintHues:[NSColor] = []

   // the chess board border opacity //
   var BorderOpacity:CGFloat = 0

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

   // the square border borders //
   var SquareBorders:BoardSquareBorders = BoardSquareBorders()

   // the chess board border texture //
   var BorderTexture:BorderRoughnessTexture = BorderRoughnessTexture()

   // the chess board textures //
   var BoardTexture:BoardRoughnessTexture = BoardRoughnessTexture()

   // the board labels font //
   var BoardLabelsFont:FontSet = FontSet()

   // the recomposite background image //
   var RecompositeBackground:CGImage?

   // the game attributes //
   @IBOutlet var Attributes:GameAttributes!

   // the movie info attributes //
   var InfoAttributes:NSDictionary = [:]

   // the movie info tag position on the movie info frame //
   var TagPosition:CGFloat = 0

   // the movie info tag //
   var Tag:NSAttributedString = NSAttributedString()

   // the core graphics functions //
   let Graphics:CoreGraphics = CoreGraphics()

   // recomposite the board and the pieces //
   func Recompose(_ Size:CGFloat, BPData:ChessBoardData, InPositionEditor:Bool,
                  FENEnpassantSquare:String, EnpassantSquare:APoint, RankAndFile:APoint,
                  UpdateBoard:Bool, Info:Bool) -> CGImage
   {
      // set the context to draw images into //
      let Context = Graphics.CreateContext(Size, Height:Size)

      // scale the context coordinates to draw the images into //
      Context.scaleBy(x:Size / 1320, y:Size / 1320)

      // if need to update the board or the recomposite background image is nil //
      if UpdateBoard || (RecompositeBackground == nil)
      {
         // draw the chess board into the context //
         Context.draw(GetBoardImage(), in:Dims.Board.BoundsRet, byTiling:false)

         // set the current context to draw into //
         NSGraphicsContext.current = NSGraphicsContext(cgContext:Context, flipped:false)

         // set chess board border color //
         GetBorderTexture().set()

         // draw the chess board border //
         Dims.BorderBounds.stroke()

         // loop over all the board labels //
         for Index in 0...7
         {
            // set the file label for the board //
            BoardLabel = BoardLabelsFont.GetString(SquareLabel.FileString(Index, Rotated:Inf
                                                   .Vals("RotateBoard").VB), White:false)

            // draw the file label at the specified point //
            BoardLabel.draw(at:CGPoint(x:Dims.Square.BoundsRet.midX + CGFloat(Index) * Dims.Square
                            .WidthRet - BoardLabel.size().width * 0.5, y:max(Dims.Border.BoundsRet
                            .midX - BoardLabel.size().height * 0.5, 0)))

            // set the rank label for the board //
            BoardLabel = BoardLabelsFont.GetString(SquareLabel.RankString(Index, Rotated:Inf
                                                   .Vals("RotateBoard").VB), White:false)

            // draw the rank label at the specified point //
            BoardLabel.draw(at:CGPoint(x:max(Dims.Border.BoundsRet.midX - BoardLabel.size().width *
                            0.5, 0), y:Dims.Square.BoundsRet.midX + CGFloat(Index) * Dims.Square
                            .WidthRet - BoardLabel.size().height * 0.5))
         }

         // set the board roughness texture fill color //
         GetBoardTexture().setFill()

         // fill the context with the board roughness texture color //
         Dims.InnerBounds.BoundsRet.fill()

         // loop over all the square border paths //
         for Index in 0..<SquareBorders.Paths.count
         {
            // set the square border color //
            SquareBorders.Colors[Index].set()

            // stroke the square border path with the given color //
            SquareBorders.Paths[Index].stroke()
         }

         // save the recomposite background image //
         RecompositeBackground = Context.makeImage()
      }
      // don't need to update the recomposite background image //
      else if RecompositeBackground != nil
      {
         // draw the recomposite background image into the context //
         Context.draw(RecompositeBackground, in:Dims.Board.BoundsRet, byTiling:false)
      }

      // if in position editor and fen enpassant square is not empty and on main sized board //
      if InPositionEditor && (FENEnpassantSquare != "-") && (Size == 1320)
      {
         // draw the en passant square in the rectangle //
         Context.draw(Highlights.EnpassantSquareImage, in:Dims.GetRect(Inf.Vals("RotateBoard").VB,
                      Point:EnpassantSquare), byTiling:false)
      }
      // not in the position editor //
      else if !InPositionEditor
      {
         // if chose to highlight possible captures //
         if Inf.Vals("HighlightCapturedPieces").VB {for TheCapSquare in BPData.CaptureSquares
         {
            // display the captured highlight image on this square //
            Context.draw(Highlights.CaptureSquareImage, in:Dims.GetRect(Inf.Vals("RotateBoard").VB,
                         Square:TheCapSquare), byTiling:false)
         }}

         // if chose to highlight threatened pieces //
         if Inf.Vals("HighlightThreatenedPieces").VB {for TheThreatSquare in BPData.ThreatSquares
         {
            // draw the threatened highlight in the rectangle //
            Context.draw(Highlights.ThreatenSquareImage, in:Dims.GetRect(Inf.Vals("RotateBoard").VB,
                         Square:TheThreatSquare), byTiling:false)
         }}

         // if start and end squares not equal and need to highlight last move made //
         if BPData.SquaresNotEqual() && (Inf.Vals("HighlightLastMoveMade").VB || (Size < 800))
         {
            // draw the last move highlight for the start square //
            Context.draw(Highlights.LastMoveImage, in:Dims.GetRect(Inf.Vals("RotateBoard").VB,
                         Point:BPData.Start), byTiling:false)

            // draw the last move highlight for the end square //
            Context.draw(Highlights.LastMoveImage, in:Dims.GetRect(Inf.Vals("RotateBoard").VB,
                         Point:BPData.End), byTiling:false)
         }

         // if move by selecting squares and have a selected square for main sized board //
         if SelectedSquare && (Size == 1320)
         {
            // draw the selected square into the rectangle //
            Context.draw(Highlights.SelectedSquareImage, in:Dims.GetRect(Inf.Vals("RotateBoard").VB,
                         Point:RankAndFile), byTiling:false)
         }

         // if not hiding the captured pieces on the board //
         if !Inf.Vals("HideCapturedPiecesInBoard").VB
         {
            // set the capture pieces background color to match that of the main window //
            Context.setFillColor(WindowTexture.ScaledTexture.cgColor)

            // if have captured pieces, fill the background color for the captured pieces //
            if !BPData.Captures.isEmpty {Context.fill(Dims.CapturesBounds.BoundsRet)}

            // loop over the list of captured pieces to display //
            for Index in 0..<BPData.Captures.count
            {
               // draw the captured piece into the context //
               Context.draw(GetPieceCGImage(BPData.Captures[Index]), in:Dims.GetCaptureRect(Index),
                            byTiling:false)
            }
         }
      }

      // loop over all files and ranks //
      for File in 0...7 {for Rank in 0...7 {if BPData.RFV(File, Rank) != 0
      {
         // draw the chess piece image onto the board //
         Context.draw(GetPieceCGImage(BPData.RFV(File, Rank)), in:Dims.GetRect(Inf
                      .Vals("RotateBoard").VB, Rank:Rank, File:File), byTiling:false)
      }}}

      // if need to print the movie info frame //
      if Info
      {
         // create the movie info attributes and initial tag position and set current context //
         (InfoAttributes, TagPosition, NSGraphicsContext.current) = ([NSAttributedString.Key.font:
         NSFont.boldSystemFont(ofSize:40), NSAttributedString.Key.foregroundColor:NSColor.black],
         1164, NSGraphicsContext(cgContext:Context, flipped:false))

         // loop over all the pgn tags //
         for Key in ["EventTitle", "SiteLocation", "DateGamePlayed", "RoundNumber", "WhiteName",
                     "BlackName", "EndResult", "TimeControl"] {if !Attributes.Inf.VSIsEmpty(Key)
         {
            // add the label to the attributed string //
            Tag = NSAttributedString(string:Attributes.AppTag(Key).first!, attributes:InfoAttributes
                  as? [NSAttributedString.Key:Any])

            // set the text background color to white //
            NSColor.white.withAlphaComponent(0.8).setFill()

            // draw the text background //
            NSBezierPath(roundedRect:(CGRect(x:65, y:TagPosition - 10, width:Tag.size().width + 66,
                         height:66)), xRadius:33, yRadius:33).fill()

            // draw the attributed string for the given tag //
            Tag.draw(at:CGPoint(x:100, y:TagPosition))

            // decrement the movie info tag position //
            TagPosition -= 150
         }}
      }

      // return the board and pieces cgimage //
      return Context.makeImage()!
   }

   // read the saved preferences //
   func Read(_ WindowTexture:WindowBackgroundTexture)
   {
      // get the boolean, integer, and string preferences //
      Inf.GetPrefs(["VBPref"], VITags:["VIPref"], VSTags:["VSPref"])

      // save the window texture and board rendering values //
      (self.WindowTexture, Ctr.BoardRoughnessSlider.intValue, Ctr.BoardTintColorSlider.intValue,
      Ctr.BoardTintVisibilitySlider.intValue, Ctr.BorderTextureSlider.intValue, Ctr
      .BorderVisibilitySlider.intValue, Ctr.SquareBorderOpacitySlider.intValue, Ctr
      .HighlightLastMove.state, Ctr.HighlightCaptures.state, Ctr.HighlightThreats.state, Ctr
      .RotateBoardButton.state, Ctr.DisplayCaptures.state, Ctr.MoveBySelectingSquares.state) =
      (WindowTexture, Int32(Inf.Vals("BoardRoughness").VI), Int32(Inf.Vals("BoardTintHue").VI),
      Int32(Inf.Vals("BoardTintVisibility").VI), Int32(Inf.Vals("BorderTextureColor").VI), Int32(Inf
      .Vals("BorderVisibility").VI), Int32(Inf.Vals("SquareBorder").VI), Ctr.UpdatedState(Inf
      .Vals("HighlightLastMoveMade").VB), Ctr.UpdatedState(Inf.Vals("HighlightCapturedPieces").VB),
      Ctr.UpdatedState(Inf.Vals("HighlightThreatenedPieces").VB), Ctr.UpdatedState(Inf
      .Vals("RotateBoard").VB), Ctr.UpdatedState(!Inf.Vals("HideCapturedPiecesInBoard").VB), Ctr
      .UpdatedState(Inf.Vals("MoveBySelectingSquares").VB))

      // populate the piece images button titles //
      PieceImages.PopulateButtonTitles(Ctr.PieceSetButton)

      // populate the board images button titles //
      BoardImages.PopulateButtonTitles(Ctr.BoardTexturesButton)

      // update the piece set button for the chosen piece set //
      Ctr.PieceSetButton.selectItem(at:Inf.Vals("ChessPieceSetType").VI)

      // update the piece set type in piece images //
      PieceImages.UpdatePieceSet(Inf.Vals("ChessPieceSetType").VI)

      // update the board textures button to the chosen board type //
      Ctr.BoardTexturesButton.selectItem(at:Inf.Vals("BoardType").VI)

      // display the board roughness //
      DisplayBoardRoughness()

      // display the current board tint color //
      DisplayBoardTintColor()

      // display the board tint visibility //
      DisplayTheBoardTintVisibility()

      // display the current chess board border texture //
      DisplayBorderTexture()

      // display the chess board border visibility //
      DisplayBorderVisibility()

      // display the square border opacity //
      DisplayTheSquareBorderOpacity()

      // read in the board labels font //
      BoardLabelsFont.Read(Ctr.BoardLabelsColorView, DefaultHue:41, TheFont:Ctr
                           .BoardLabelsFontButton, Sizes:Ctr.BoardLabelsSizesButton, Bold:Ctr
                           .BoardLabelsBoldButton, TheScale:2, VertOffset:0, UseVertOffset:false,
                           Tag:"BoardLabels")
   }

   // toggle the key //
   func ToggleKey(_ Key:String)
   {
      // play the sound of a button click //
      GameSounds.Play("click")

      // toggle the key //
      Inf.ToggleVB("VBPref", Key:Key)

      // update the preference previews //
      Board.UpdatePreviews(true, Diagram:false)
   }

   // rotate the board view //
   @IBAction func RotateBoardAction(_:AnyObject) {ToggleKey("RotateBoard")}

   // hide the captured pieces in the board action //
   @IBAction func HideCapturedPiecesAction(_:AnyObject) {ToggleKey("HideCapturedPiecesInBoard")}

   // the move by selecting squares action //
   @IBAction func MoveBySelectingSquaresAction(_:AnyObject) {ToggleKey("MoveBySelectingSquares")}

   // set that have selected a square on the board //
   func SetSelectedSquare() {SelectedSquare = true}

   // clear that have selected a square on the board //
   func InitSelectedSquare() {SelectedSquare = false}

   // highlight the captured pieces button action //
   @IBAction func HighlightCapturedPiecesAction(_:AnyObject) {ToggleKey("HighlightCapturedPieces")}

   // highlight the last move made button action //
   @IBAction func HighlightLastMoveMadeAction(_:AnyObject) {ToggleKey("HighlightLastMoveMade")}

   // highlight the threatened pieces button action //
   @IBAction func HighlightThreatPiecesAction(_:AnyObject) {ToggleKey("HighlightThreatenedPieces")}

   // display the board roughness //
   func DisplayBoardRoughness()
   {
      // update the board roughness opacity //
      BoardRoughnessOpacity = Inf.Fraction(350, Key:"BoardRoughness")

      // update the board roughness opacity view //
      Ctr.BoardRoughnessView.stringValue = "Roughness" + Inf.Perc("BoardRoughness")
   }

   // update the board roughness //
   @IBAction func UpdateBoardRoughness(_:AnyObject)
   {
      // update the current board roughness from the board roughness slider //
      Inf.Update("VIPref", VIItems:["BoardRoughness":Int(Ctr.BoardRoughnessSlider.intValue)])

      // display the board roughness //
      DisplayBoardRoughness()

      // update the preference previews //
      Board.UpdatePreviews(true, Diagram:false)
   }

   // save the preferences //
   func SavePreferences()
   {
      // save the boolean, integer, and string preferences //
      Inf.SavePrefs(["VBPref"], VITags:["VIPref"], VSTags:["VSPref"])

      // save the board labels font preferences //
      BoardLabelsFont.SavePreferences()
   }

   // display the board tint color //
   func DisplayBoardTintColor()
   {
      // update the board tint color view //
      Ctr.BoardTintColorView.stringValue = "Tint color is " + Inf.IntStr("BoardTintHue")
   }

   // display the board tint opacity //
   func DisplayTheBoardTintVisibility()
   {
      // update the board tint opacity and opacity view//
      (TintColorOpacity, Ctr.BoardTintVisibilityView.stringValue) = (Inf.Fraction(400, Key:
      "BoardTintVisibility"), "Tint" + Inf.Perc("BoardTintVisibility"))
   }

   // display the square border opacity //
   func DisplayTheSquareBorderOpacity()
   {
      // update the square border colors //
      SquareBorders.UpdateSquareBorderColors(Inf.Fraction(200, Key:"SquareBorder"))

      // update the square border opacity view //
      Ctr.SquareBorderView.stringValue = "Square border" + Inf.Perc("SquareBorder")
   }

   // display the chess board border visibility //
   func DisplayBorderVisibility()
   {
      // update the chess board border opacity and border visibility //
      (BorderOpacity, Ctr.BorderVisibilityView.stringValue) = (Inf.Fraction(100, Key:
      "BorderVisibility"), "Border" + Inf.Perc("BorderVisibility"))
   }

   // get the board texture //
   func GetBoardTexture() -> NSColor
   {
      // return the board texture //
      return BoardTexture.GetTexture(BoardTintHues[Inf.Vals("BoardTintHue").VI], TintOpacity:
                                     TintColorOpacity, RoughnessOpacity:BoardRoughnessOpacity)
   }

   // get the chess board border texture //
   func GetBorderTexture() -> NSColor
   {
      // return the chess board border texture //
      return BorderTexture.GetTexture(Ctr.WindowBackgroundColorView.GetCurrentHue(), BorderColor:
             BorderColors[Inf.Vals("BorderTextureColor").VI], BorderOpacity:BorderOpacity)
   }

   // display the border texture //
   func DisplayBorderTexture()
   {
      // display the border texture for the chess board //
      Ctr.BorderTextureView.stringValue = "Border texture is " + Inf.IntStr("BorderTextureColor")
```

```swift
      Ctr.BorderTextureView.stringValue = "Border texture is " + Inf.IntStr("BorderTextureColor")
   }

   // get the piece nsimage //
   func GetPieceNSImage(_ MovingPiece:Int) -> NSImage {PieceImages.GetPieceNSImage(MovingPiece)}

   // get the piece cgimage //
   func GetPieceCGImage(_ Piece:Int) -> CGImage {return PieceImages.GetPieceCGImage(Piece)}

   // get the board image //
   func GetBoardImage() -> CGImage {return BoardImages.GetBoardImage(Inf.Vals("BoardType").VI)}

   // update the chess board border visibility //
   @IBAction func UpdateBorderVisibility(_:AnyObject)
   {
      // update the current border visibility from the border visibility slider //
      Inf.Update("VIPref", VIItems:["BorderVisibility":Int(Ctr.BorderVisibilitySlider.intValue)])

      // display the border visibiity //
      DisplayBorderVisibility()

      // update the preference previews //
      Board.UpdatePreviews(true, Diagram:false)
   }

   // update the border texture //
   @IBAction func UpdateBorderTexture(_:AnyObject)
   {
      // update the current border texture from the border texture slider //
      Inf.Update("VIPref", VIItems:["BorderTextureColor":Int(Ctr.BorderTextureSlider.intValue)])

      // display the chess board border texture //
      DisplayBorderTexture()

      // update the preference previews //
      Board.UpdatePreviews(true, Diagram:false)
   }

   // update the board tint color //
   @IBAction func UpdateBoardTintColor(_:AnyObject)
   {
      // update the board tint hue from the board tint color slider //
      Inf.Update("VIPref", VIItems:["BoardTintHue":Int(Ctr.BoardTintColorSlider.intValue)])

      // display the current board tint color //
      DisplayBoardTintColor()

      // update the preference previews //
      Board.UpdatePreviews(true, Diagram:false)
   }

   // update the board tint visibility //
   @IBAction func UpdateBoardTintVisibility(_:AnyObject)
   {
      // update the board tint visibility from the board tint visibility slider //
      Inf.Update("VIPref", VIItems:["BoardTintVisibility":Int(Ctr.BoardTintVisibilitySlider
                 .intValue)])

      // display the board tint visibility //
      DisplayTheBoardTintVisibility()

      // update the preference previews //
      Board.UpdatePreviews(true, Diagram:false)
   }

   // update the square border opacity //
   @IBAction func UpdateSquareBorderOpacity(_:AnyObject)
   {
      // update the square border opacity from the square border opacity slider //
      Inf.Update("VIPref", VIItems:["SquareBorder":Int(Ctr.SquareBorderOpacitySlider.intValue)])

      // display the square border opacity //
      DisplayTheSquareBorderOpacity()

      // update the preference previews //
      Board.UpdatePreviews(true, Diagram:false)
   }

   // update the board type //
   @IBAction func UpdateBoardTypeLayer(_:AnyObject)
   {
      // update the board type from the board textures button //
      Inf.Update("VIPref", VIItems:["BoardType":Ctr.BoardTexturesButton.indexOfSelectedItem])

      // update the preference previews //
      Board.UpdatePreviews(true, Diagram:false)
   }

   // update the chess piece set type //
   @IBAction func UpdatePieceSetType(_:AnyObject)
   {
      // update the chess piece set type from the piece set button //
      Inf.Update("VIPref", VIItems:["ChessPieceSetType":Ctr.PieceSetButton.indexOfSelectedItem])

      // update the piece set type //
      PieceImages.UpdatePieceSet(Inf.Vals("ChessPieceSetType").VI)

      // update the preference previews //
      Board.UpdatePreviews(true, Diagram:false)

      // update the position editor toolbar buttons //
      Board.UpdatePositionEditorToolbarButtons()
   }

   // the class initializer //
   override init()
   {
      // save the default status values for the boolean preferences //
      Inf.Update("VBPref", VBItems:["RotateBoard":false, "HighlightLastMove":false,
                 "HighlightCapturedPieces":false, "HighlightThreatenedPieces":false,
                 "HideCapturedPiecesInBoard":false, "MoveBySelectingSquares":false])

      // save the default integer values for the integer preferences //
      Inf.Update("VIPref", VIMinMaxInItems:["ChessPieceSetType":[0, 0, 3], "BoardType":
                 [0, 0, 18], "BoardRoughness":[0, 0, 100], "BoardTintHue":[30, 0, 100],
                 "BoardTintVisibility":[0, 0, 100], "BorderTextureColor":[100, 0, 100],
                 "BorderVisibility":[100, 0, 100], "SquareBorder":[0, 0, 100]])

      // loop over all the border and board tint colors //
      for Index in 0...100
      {
         // append a color to the chess board border colors //
         BorderColors.append(NSColor(calibratedRed:sin(CGFloat(Index) * CGFloat.pi / 200) / 3, green:
                    sin(CGFloat(Index) * CGFloat.pi / 200 + CGFloat.pi / 4) / 3, blue:
                    sin(CGFloat(Index) * CGFloat.pi / 200 + CGFloat.pi / 2) / 3, alpha:1))

         // append a board tint hue //
         BoardTintHues.append(NSColor(hue:CGFloat(Index) / 100, saturation:1, brightness:1, alpha:
                    1))
      }
   }
}

// class to convert an integer to a bool //
class IntTo {func Boo(_ Value:Int) -> Bool {return Value != 0}}

// class to convert a bool to an integer or a cgfloat //
class BoolTo
{
   // convert a bool to an integer //
   func Num(_ Value:Bool) -> Int {return Value ? 1 : 0}

   // convert a bool to a cgfloat //
   func CG(_ Value:Bool) -> CGFloat {return Value ? 1 : 0}
}

// the class for toolbar tabs //
class ToolbarTabs
{
   // the toolbar items and other associated objects //
   var Items:[AnyObject] = []

   // the play button in the toolbar //
   var PlayButton:NSToolbarItem!

   // the play icon //
   var PlayIcon:NSImage = NSImage(named:"start_icon")!

   // the pause icon //
   var PauseIcon:NSImage = NSImage(named:"pause_icon")!

   // update the state of the toolbar buttons //
   func UpdateState(_ Enabled:Bool)
   {
      // loop over an object in items //
      for Object in Items {switch Object
      {
         // update is enabled in the case where the toolbar object is a toolbar item //
         case let ToolbarObject as NSToolbarItem : ToolbarObject.isEnabled = Enabled

         // update is enabled in the case where the toolbar object is a slider //
         case let ToolbarObject as NSSlider : ToolbarObject.isEnabled = Enabled

         // the default case //
         default : break
      }}

      // set the play button image //
      PlayButton.image = Enabled ? PlayIcon : PauseIcon
   }

   // initialize the toolbar items //
   func Initialize(_ Items:[AnyObject], PlayButton:NSToolbarItem)
   {
      // save the toolbar items and the play button //
      (self.Items, self.PlayButton) = (Items, PlayButton)

      // update the toolbar item states //
      UpdateState(true)
   }
}

// the class for controller objects //
class Controls
{
   // the controller items //
   var Items:[AnyObject] = []

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

   // a controller frame //
   var Frame:CGRect = CGRect()

   // update the controller states //
   func UpdateState(_ Hidden:Bool)
   {
      // loop over each object in items //
      for Object in Items {switch Object
      {
         // update is hidden in the case where the object is a popup button //
         case let PrefObject as NSPopUpButton : PrefObject.isHidden = Hidden

         // update is hidden in the case where the object is an image view //
         case let PrefObject as NSImageView : PrefObject.isHidden = Hidden

         // update is hidden in the case where the object is a slider //
         case let PrefObject as NSSlider : PrefObject.isHidden = Hidden

         // update is hidden in the case where the object is a text field //
         case let PrefObject as NSTextField : PrefObject.isHidden = Hidden

         // update is hidden in the case where the object is a button //
         case let PrefObject as NSButton : PrefObject.isHidden = Hidden

         // update is hidden in the case where the object is a colors view //
         case let PrefObject as ColorsView : PrefObject.isHidden = Hidden

         // update is hidden in the case where the object is a time clock display //
         case let PrefObject as TimeClockDisplay : PrefObject.isHidden = Hidden

         // the default case //
         default : break
      }}
   }

   // initialize the controls //
   func Initialize(_ ControlItems:[AnyObject], Frames:[[CGFloat]])
   {
      // save the controller items and set index to zero //
      (Items, Index) = (ControlItems, 0)

      // loop over all the objects in items //
      for Object in Items
      {
         // if there is an available frame in the frames lists //
         if Index < Frames.count && Frames[Index].count == 4
         {
            // set the frame from the extracted frame components //
            Frame = CGRect(x:Frames[Index][0], y:Frames[Index][1], width:Frames[Index][2],height:
                    Frames[Index][3])
         }}

         // increment the index //
         Index += 1

         // switch over the object //
         switch Object
         {
            // update the frame in the case where object is a popup button //
            case let PrefObject as NSPopUpButton : PrefObject.frame = Frame

            // update the frame in the case where object is an image view //
            case let PrefObject as NSImageView : PrefObject.frame = Frame

            // update the frame in the case where object is a slider //
            case let PrefObject as NSSlider : PrefObject.frame = Frame

            // update the frame in the case where the object is a text field //
            case let PrefObject as NSTextField : PrefObject.frame = Frame

            // update the frame in the case where object is a button //
            case let PrefObject as NSButton : PrefObject.frame = Frame

            // update the frame in the case where object is a color view //
            case let PrefObject as ColorsView : PrefObject.frame = Frame

            // update the frame in the case where object is a time clock display //
            case let PrefObject as TimeClockDisplay : PrefObject.frame = Frame

            // the default case //
            default : break
         }
      }

      // update the controller states //
      UpdateState(true)
   }
}

// the user preferences class //
class UserPreferences
{
   // the preferences //
   var Prefs:UserDefaults = UserDefaults.standard

   // save an integer value for the specified key //
   func Save(_ Value:Int, Key:String) {Prefs.set(Value, forKey:Key)}

   // save a boolean value for the specified key //
   func Save(_ Value:Bool, Key:String) {Prefs.set(Value, forKey:Key)}

   // save a string value for the specified key //
   func Save(_ Value:String, Key:String) {Prefs.set(Value, forKey:Key)}

   // get the user saved preference for the specified key //
   func PrefInt(_ Key:String, Default:Int, Min:Int, Max:Int) -> Int
   {
      // if the user preference already exists //
      if Prefs.bool(forKey:Key + "Exists")
      {
         // if in bounds, return the saved preference and if not in bounds, return the default //
         return (Min...Max).contains(Prefs.integer(forKey:Key)) ? Prefs.integer(forKey:Key) :
                Default
      }

      // set that the user preference exists now //
      Prefs.set(true, forKey:Key + "Exists")

      // set the default value for the specified key //
      Prefs.set(Default, forKey:Key)

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

   // get the user saved preference for the key //
   func PrefBool(_ Key:String, Default:Bool) -> Bool
   {
      // return the saved user preference if exists //
      if Prefs.bool(forKey:Key + "Exists") {return Prefs.bool(forKey:Key)}

      // set that the user preference exists now //
      Prefs.set(true, forKey:Key + "Exists")

      // set the default value for the specified key //
      Prefs.set(Default, forKey:Key)

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

   // get the user saved preference for this key //
   func PrefString(_ Key:String, Default:String) -> String
   {
      // return the saved user preference if it exists //
      if Prefs.bool(forKey:Key + "Exists") {return Prefs.string(forKey:Key)!}

      // set that the user preference now exists for this key //
      Prefs.set(true, forKey:Key + "Exists")

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

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

// the fonts set //
class FontsSet
{
   // the information dictionary //
   var Inf:Information = Information()

   // the paragraph style //
   var ParagraphStyle:NSMutableParagraphStyle = NSMutableParagraphStyle()

   // the font color view //
   var ColorView:ColorsView!

   // the font type popup button //
   var FontTypeButton:NSPopUpButton!

   // the size index button //
   var SizesIndexButton:NSPopUpButton!

   // the bold button //
   var BoldButton:NSPopUpButton!

   // the prefix for the font strings //
   var Prefix:String = ""

   // the font size //
   var Size:CGFloat = 10

   // the resolution scale //
   var Scale:CGFloat = 1

   // the current font //
   var CurrentFont:NSFont = NSFont()

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

   // font attributes //
   var Attributes:NSDictionary = [:]

   // the vertically offset font attributes //
   var VerticallyOffsetAttributes:NSDictionary = [:]

   // the white font attributes //
   var WhiteFontAttributes:NSDictionary = [:]

   // whether to use a paragraph vertical offset or not //
   var UseParagraphOffset:Bool = false

   // 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"]

   // read the saved preferences for the font //
   func Read(_ View:ColorsView, DefaultHue:Int, TheFont:NSPopUpButton, Sizes:NSPopUpButton, Bold:
             NSPopUpButton, Scale:CGFloat, VertOffset:Int, UseOffset:Bool, Tag:String)
   {
      // save the prefix string, color view, and font values //
      (Prefix, ColorView, FontTypeButton, SizesIndexButton, BoldButton, Scale) = (Tag, View,
       TheFont, Sizes, Bold, TheScale)

      // save the default status values for the font preferences //
      Inf.Update("VBPref", VBItems:[Prefix + "Bold":false])

      // save the default integer values for the font preferences //
      Inf.Update("VIPref", VIMinMaxInItems:[Prefix + "FontType":[17, 0, Fonts.count - 1], Prefix
                 + "Sizes":[2, 0, 2], Prefix + "Hue":[DefaultHue, 0, View.MaxHue()]])

      // get the font preferences //
      Inf.GetPrefs(["VBPref"], VIPrefs:["VIPref"], VSTags:["VSPref"])

      // clear all the font type button items //
      FontTypeButton.removeAllItems()

      // loop over all the fonts //
      for TheFont in Fonts {FontTypeButton.addItem(withTitle:TheFont)}

      // initialize the font color view with the saved hue //
      ColorView.Initialize(Inf.Vals(Prefix + "Hue").VI)

      // update the font type button selection //
      FontTypeButton.selectItem(at:Inf.Vals(Prefix + "FontType").VI)

      // update the bold button selection //
      BoldButton.selectItem(at:BoolTo().Num(Inf.Vals(Prefix + "Bold").VB))

      // update the size index button //
      SizesIndexButton.selectItem(at:Inf.Vals(Prefix + "Sizes").VI)

      // update the font //
      UpdateFont(VertOffset, UseVertOffset:UseVertOffset)
   }

   // save the font preferences //
   func SavePreferences() {Inf.SavePrefs(["VBPref"], VIPrefs:["VIPref"], VSPrefs:["VSPref"])}

   // update the font //
   func UpdateFont(_ VertOffset:Int, UseVertOffset:Bool)
   {
      // get if bold font from the bold button //
      Inf.Update("VBPref", VBItems:[Prefix + "Bold":IntTo().Boo(BoldButton.indexOfSelectedItem)])

      // get the font type from the font type button, size, and hue //
      Inf.Update("VIPref", VIItems:[Prefix + "FontType":FontTypeButton.indexOfSelectedItem,
                 Prefix + "Sizes":SizesIndexButton.indexOfSelectedItem, Prefix + "Hue":ColorView
                 .GetHueValue()])

      // get the font size and font bold string and other variables //
      (Size, FontBoldString, UseParagraphOffset, ParagraphStyle.paragraphSpacing) = (Scale * (16 +
       3 * Scale * Inf.Vals(Prefix + "Sizes").VI, Inf.Vals(Prefix + "Bold").VB ? " Bold " : "",
       UseVertOffset, CGFloat(max(0, VertOffset - 10)))

      // set the current font //
      CurrentFont = NSFont(name:Fonts[Inf.Vals(Prefix + "FontType").VI] + FontBoldString, size:
                    Size) == nil ? (Inf.Vals(Prefix + "Bold").VB ? NSFont.boldSystemFont(ofSize:
                    Size) : NSFont.systemFont(ofSize:Size)) : NSFont(name:Fonts[Inf.Vals(Prefix +
                    "FontType").VI] + FontBoldString, size:Size)!

      // set the font attributes with and without vertical offset //
      (VerticallyOffsetAttributes, Attributes, WhiteFontAttributes) = ([NSAttributedString.Key
       .font:CurrentFont, NSAttributedString.Key.foregroundColor:ColorView.GetCurrentHue(),
       NSAttributedString.Key.paragraphStyle:ParagraphStyle], [NSAttributedString.Key.font:
       CurrentFont, NSAttributedString.Key.foregroundColor:ColorView.GetCurrentHue()],
       [NSAttributedString.Key.font:CurrentFont, NSAttributedString.Key.foregroundColor:NSColor
       .white])
   }

   // get the attributed text string //
   func GetString(_ Text:String, White:Bool) -> NSAttributedString
   {
      // return the attributed string based on type of string needed //
      return NSAttributedString(string:Text, attributes:(White ? WhiteFontAttributes :
             (UseParagraphOffset ? VerticallyOffsetAttributes : Attributes)) as? [NSAttributedString
             .Key:Any])
   }

   // get the text string height //
   func GetStringHeight(_ Text:NSAttributedString, Width:Int) -> CGFloat
   {
      // return the text string height needed to draw in the specified rectangle //
      return Text.boundingRect(with:NSSize(width:Width, height:10000), options:
                               [.usesLineFragmentOrigin, .usesFontLeading], context:nil).size.height
   }
}

// the chess board border roughness texture class //
class BorderRoughnessTexture
{
   // the border roughness texture //
   var BorderRoughnessTexture:NSImage = NSImage()

   // the roughness square //
   var RoughnessSquare:NSBezierPath = NSBezierPath()

   // get the chess board border texture //
   func GetTexture(_ WindowColor:NSColor, BorderColor:NSColor, BorderOpacity:CGFloat) -> NSColor
   {
      // create the border roughness texture image //
      BorderRoughnessTexture = NSImage(size:CGSize(width:8, height:8))

      // lock the border roughness texture focus //
      BorderRoughnessTexture.lockFocus()

      // set the blended fill color from the current window color and the border color //
      WindowColor.blended(withFraction:BorderOpacity, of:BorderColor)!.setFill()

      // fill the border roughness texture with the blended color //
      NSBezierPath(rect:CGRect(x:0, y:0, width:8, height:8)).fill()

      // set the texture color //
      NSColor(red:1, green:1, blue:1, alpha:BorderOpacity / 18).set()

      // stroke the roughness square with the texture color //
      RoughnessSquare.stroke()

      // unlock the border roughness texture image //
      BorderRoughnessTexture.unlockFocus()

      // return the chess board border texture color //
      return NSColor(patternImage:BorderRoughnessTexture)
   }

   // the chess board border texture class initializer //
   init()
   {
      // create the roughness square //
      RoughnessSquare = NSBezierPath(rect:CGRect(x:2, y:2, width:4, height:4))

      // set the line width of the roughness square //
      RoughnessSquare.lineWidth = 2
   }
}

// the chess board roughness texture class //
class BoardRoughnessTexture
{
   // board roughness texture image //
   var BoardRoughnessTexture:NSImage = NSImage()

   // the roughness circle //
   var RoughnessCircle:NSBezierPath = NSBezierPath()

   // get the board roughness texture color //
   func GetTexture(_ Tint:NSColor, TintOpacity:CGFloat, RoughnessOpacity:CGFloat) -> NSColor
   {
      // create the board roughness texture image //
      BoardRoughnessTexture = NSImage(size:CGSize(width:8, height:8))

      // lock the board roughness texture focus //
      BoardRoughnessTexture.lockFocus()

      // set the current board tint fill color with the given opacity //
      Tint.withAlphaComponent(TintOpacity).setFill()

      // fill the image with the tint //
      NSBezierPath(rect:CGRect(x:0, y:0, width:8, height:8)).fill()

      // set the stroke color to light gray //
      NSColor(red:0.85, green:0.85, blue:0.85, alpha:RoughnessOpacity).set()

      // draw the roughness circle //
      RoughnessCircle.stroke()

      // unlock the board roughness texture focus //
      BoardRoughnessTexture.unlockFocus()

      // return the board texture color //
      return NSColor(patternImage:BoardRoughnessTexture)
   }

   // the chess board texture class initializer //
   init()
   {
      // create the roughness circle //
      RoughnessCircle = NSBezierPath(ovalIn:CGRect(x:2, y:2, width:4, height:4))

      // set the line width of the roughness circle //
      RoughnessCircle.lineWidth = 2
   }
}

// the progress circle class //
class ProgressCircleLayer
{
   // the progress circle layers //
   var Layers:[CALayer] = [CALayer(), CALayer(), CALayer()]

   // the board dimensions //
   let Dims:BoardDimensions = BoardDimensions()

   // whether to animate progress //
   var AnimateProgress:Bool = false

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

   // the progress circles //
   var Circles:[CGImage] = []

   // start the progress circle animation //
   func StartProgressAnimation() {AnimateProgress = true}

   // stop the progress circle animation //
   func StopProgressAnimation() {AnimateProgress = false}

   // the core graphics functions //
   let Graphics:CoreGraphics = CoreGraphics()

   // update the progress circle //
   func UpdateProgress()
   {
      // update layer opacities //
      for TheLayer in Layers {TheLayer.opacity = AnimateProgress ? 1 : 0}

      // if need to animate the progress circle //
      if AnimateProgress
      {
         // update the progress circle index //
         Index = ((Index + 1) >= Circles.count) ? 0 : Index + 1

         // loop over all the layers //
         for Count in 0..<Layers.count
         {
            // remove the animation effects //
            CATransaction.setDisableActions(true)

            // update the layer contents //
            Layers[Count].contents = Circles[(Index + Count * (Circles.count / Layers.count)) %
                                     Circles.count]
         }
      }
   }

   // add the progress circle layers //
   func AddLayer(_ View:NSView) {for TheLayer in Layers {View.layer?.addSublayer(TheLayer)}}

   // update the progress circle color //
   func UpdateCircleColor(_ Color:NSColor)
   {
      // remove all the circles //
      Circles.removeAll()

      // loop over 100 circles //
      for Index in 0...99
      {
         // append a progress circle //
         Circles.append(Graphics.CreateShape(Dims.Square.WidthRet, Fraction:sin(CGFloat(Index) / 99
                        * CGFloat.pi), Color:Color.withAlphaComponent(0.5 * sin(CGFloat(Index) / 99
                        * CGFloat.pi)), Square:false))
      }
   }

   // the progress circle class initializer //
   init()
   {
      // loop over all the layers //
      for TheLayer in Layers
      {
         // set the layer position and bounds //
         (TheLayer.position, TheLayer.bounds) = (CGPoint(x:Dims.Board.Bounds.midX, y:Dims.Board
          .Bounds.midX), CGRect(x:0, y:0, width:Dims.Square.Width, height:Dims.Square.Width))
      }
   }
}

// the position view layers class //
class PositionViewLayers
{
   // the position layers //
   var Layers:[CALayer] = [CALayer(), CALayer()]

   // the saved position animation //
   var PositionAnimation:CABasicAnimation = CABasicAnimation(keyPath:"opacity")

   // add the position layers to the view //
   func AddLayers(_ View:NSView) {for TheLayer in Layers {View.layer?.addSublayer(TheLayer)}}

   // update the position layer //
   func UpdateLayer(_ Image:CGImage)
   {
      // remove the animation effects //
      CATransaction.setDisableActions(true)

      // copy the current saved position into position layer one and set layer two opacity to 0 //
      (Layers[0].contents, Layers[1].opacity) = (Image, 0)
   }

   // remove all the layer animations //
   func RemoveAnimations() {for TheLayer in Layers {TheLayer.removeAllAnimations()}}

   // animate the position layers //
   func Animate(_ View:NSView, Image:CGImage)
   {
      // remove the animations //
      RemoveAnimations()

      // remove position layer two from the super layer //
      Layers[1].removeFromSuperlayer()

      // create position layer two //
      Layers[1] = CALayer()

      // initialize position layer two //
      Initialize(1)

      // set the position layer two contents to position layer one contents //
      Layers[1].contents = Layers[0].contents

      // add the position layer two to the load position view //
      View.layer?.addSublayer(Layers[1])

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

      // set the position layer values //
      (Layers[0].contents, PositionAnimation.fromValue, PositionAnimation.toValue, Layers[1]
       .opacity) = (Image, 1, 0, 0)

      // add an animation to position layer two to animate opacity //
      Layers[1].add(PositionAnimation, forKey:"opacity")
   }

   // initialize a position layer //
   func Initialize(_ Index:Int)
   {
      // set the bounds of the position layer and the position //
      (Layers[Index].bounds, Layers[Index].position) = (CGRect(x:0, y:0, width:450, height:450),
       CGPoint(x:225, y:225))
   }

   // the position layers class initializer //
   init() {Initialize(0); Initialize(1); PositionAnimation.duration = 0.5}
}

// the window background texture class //
class WindowBackgroundTexture
{
   // the window texture overlay //
   var Overlay:NSImage = NSImage()

   // the window texture color //
   var Texture:NSColor = NSColor()

   // the window scaled texture color //
   var ScaledTexture:NSColor = NSColor()

   // the window pattern image //
   var Pattern:NSImage = NSImage()

   // the window preview image //
   var Preview:NSImage = NSImage()

   // get the window texture //
   func GetTexture(_ CurrentColor:NSColor, Opacity:Int) -> NSColor
   {
      // initialize the pattern image //
      Pattern = NSImage(size:CGSize(width:40, height:40))

      // lock the pattern image focus //
      Pattern.lockFocus()

      // set the fill color from the base color //
      CurrentColor.setFill()

      // fill the rectangle with the base color //
      NSBezierPath(rect:CGRect(x:0, y:0, width:40, height:40)).fill()

      // draw the window background overlay into the rectangle //
      Overlay.draw(in:CGRect(x:0, y:0, width:40, height:40), from:NSZeroRect, operation:
                NSCompositingOperation.sourceOver, fraction:0.003 * CGFloat(Opacity))

      // unlock the pattern image focus //
      Pattern.unlockFocus()

      // set the window texture from the pattern image and create window texture preview image //
      (Texture, ScaledTexture, Preview) = (NSColor(patternImage:Pattern), NSColor(patternImage:
      NSImage(cgImage:Pattern.cgImage(forProposedRect:nil, context:nil, hints:nil)!, size:
      CGSize(width:80, height:80))), NSImage(size:CGSize(width:200, height:200)))

      // lock the window texture preview image focus //
      Preview.lockFocus()

      // set the fill color from the window texture //
      Texture.setFill()

      // fill the rectangle with the window texture //
      NSBezierPath(rect:CGRect(x:0, y:0, width:200, height:200)).fill()

      // unlock the window color preview image //
      Preview.unlockFocus()

      // return the window texture color //
      return Texture
   }

   // the window background texture class initializer //
   init()
   {
      // initialize the window background overlay image //
      Overlay = NSImage(size:CGSize(width:40, height:40))

      // lock the window background overlay focus //
      Overlay.lockFocus()

      // loop over the x index and the y indices //
      for X in 0...9 {for Y in 0...9
      {
         // set the fill color to light gray //
         NSColor.lightGray.setFill()

         // make a light gray rectangle //
         NSBezierPath(rect:CGRect(x:CGFloat(X) * 4, y:CGFloat(Y) * 4 + 2.5, width:4, height:1))
                  .fill()

         // make a light gray rectangle //
         NSBezierPath(rect:CGRect(x:CGFloat(X) * 4 + 2.5, y:CGFloat(Y) * 4, width:1, height:4))
                  .fill()

         // set the fill color to gray //
         NSColor.gray.setFill()

         // make a gray rectangle //
         NSBezierPath(rect:CGRect(x:CGFloat(X) * 4, y:CGFloat(Y) * 4 + 0.5, width:4, height:1))
                  .fill()

         // make a gray rectangle //
         NSBezierPath(rect:CGRect(x:CGFloat(X) * 4 + 0.5, y:CGFloat(Y) * 4, width:1, height:4))
                  .fill()

         // set the fill color to light gray //
         NSColor.lightGray.setFill()

         // make a light gray rectangle //
         NSBezierPath(rect:CGRect(x:CGFloat(X) * 4 + (((X + Y) % 2 == 0) ? 0.5 : 2.5), y:CGFloat(Y)
                  * 4 + (((X + Y) % 2 == 0) ? 2.5 : 0.5), width:1, height:1)).fill()
      }}

      // unlock the window background overlay focus //
      Overlay.unlockFocus()
   }
}

// the annotation view layers class //
class AnnotationViewLayers
{
   // the annotation view layers //
   var Layers:[CALayer] = [CALayer(), CALayer(), CALayer()]

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

   // the annotation view layers class initializer //
   init()
   {
      // set the rotation with perspective value to 0.004 //
      RotationWithPerspective.m34 = 0.004

      // set the annotation layer values //
      (Layers[1].bounds, Layers[1].position, Layers[0].bounds, Layers[0].position, Layers[0]
      .anchorPoint, Layers[0].transform, Layers[2].bounds, Layers[2].position, Layers[2].anchorPoint,
      Layers[2].transform) = (CGRect(x:0, y:0, width:250, height:250), CGPoint(x:330, y:125),
      CGRect(x:0, y:0, width:250, height:250), CGPoint(x:204, y:125), CGPoint(x:1, y:0.5),
      CATransform3DRotate(RotationWithPerspective, 0.35, 0, 1, 0), CGRect(x:0, y:0, width:250,
      height:250), CGPoint(x:456, y:125), CGPoint(x:0, y:0.5),
      CATransform3DRotate(RotationWithPerspective, -0.35, 0, 1, 0))
   }

   // clear the annotation layers //
   func ClearLayers(_ Start:Int)
   {
      // loop over all the layers from the start index and on //
      for TheLayer in Layers[Start...]
      {
         // remove the animation effects //
         CATransaction.setDisableActions(true)

         // set the annotation board contents to an empty image for left slot //
         TheLayer.contents = NSImage(size:CGSize(width:250, height:250))
      }
   }

   // add the annotation layers //
   func AddLayers(_ View:NSView) {for TheLayer in Layers {View.layer?.addSublayer(TheLayer)}}

   // update the specified layer //
   func UpdateLayer(_ LayerNumber:Int, Image:CGImage)
   {
      // disable the animation effects //
      CATransaction.setDisableActions(true)

      // update the layer image //
      Layers[LayerNumber].contents = Image
   }
}

// the board highlight squares //
class BoardHighlightSquares //
{
   // the board dimensions //
   let Dims:BoardDimensions = BoardDimensions()

   // the capture square image //
   var CaptureSquareImage:CGImage!

   // the threaten square image //
   var ThreatenSquareImage:CGImage!

   // the last move image //
   var LastMoveImage:CGImage!

   // the selected square image //
   var SelectedSquareImage:CGImage!

   // the enpassant square cgimage //
   var EnpassantSquareImage:CGImage!

   // the enpassant square nsimage //
   var EnpassantSquareNSImage:NSImage!

   // the core graphics functions //
   let Graphics:CoreGraphics = CoreGraphics()

   // the board highlight squares class initializer //
   init()
   {
      // set the board highlight squares //
      (LastMoveImage, CaptureSquareImage, ThreatenSquareImage, SelectedSquareImage,
      EnpassantSquareImage) = (Graphics.CreateShape(Dims.Square.WidthRet, Fraction:0.9, Color:
      NSColor(red:1, green:1, blue:0.65, alpha:1), Square:false), Graphics.CreateShape(Dims.Square
      .WidthRet, Fraction:0.95, Color:NSColor(red:1, green:0.3, blue:0.3, alpha:1), Square:false),
      Graphics.CreateShape(Dims.Square.WidthRet, Fraction:0.95, Color:NSColor(red:1, green:0.6,
      blue:0, alpha:1), Square:false), Graphics.CreateShape(Dims.Square.WidthRet, Fraction:0.97,
      Color:NSColor(red:0.2, green:0.7, blue:1, alpha:1), Square:true), Graphics.CreateShape(Dims
      .Square.WidthRet, Fraction:0.95, Color:NSColor(red:0.45, green:0.45, blue:1, alpha:1), Square:
      false))

      // set the enpassant square nsimage //
      EnpassantSquareNSImage = Graphics.CGImageToNSImage(EnpassantSquareImage, Size:Dims.Square
```

```
      EnpassantSquareImage = Graphics.CGImageFromNSImage(EnpassantSquareImage, Size:Dims.Square
                                                            .WidthRet)
   }
}

// the board square borders class //
class BoardSquareBorders
{
   // the square origin //
   var Origin:CGPoint = CGPoint()

   // the square border offset //
   var Offset:CGFloat = 0

   // the board dimensions //
   let Dims:BoardDimensions = BoardDimensions()

   // the square border path colors //
   var Colors:[NSColor] = [NSColor.black, NSColor.white, NSColor.gray]

   // the square border paths //
   var Paths:[NSBezierPath] = [NSBezierPath(), NSBezierPath(), NSBezierPath()]

   // the board square borders class initializer //
   init()
   {
      // loop over all the square border paths //
      for Index in 0..<Paths.count
      {
         // set the square border path line width and line cap style //
         (Paths[Index].lineWidth, Paths[Index].lineCapStyle) = (2, .square)

         // loop over all the files and ranks //
         for File in 0...7 {for Rank in 0...7
         {
            // set the current square origin and the square border offset from the edge //
            (Origin, Offset) = (CGPoint(x:CGFloat(File) * Dims.Square.WidthRet + Dims.Border
            .WidthRet, y:CGFloat(Rank) * Dims.Square.WidthRet + Dims.Border.WidthRet),
            CGFloat(Index) * 2)

            // move to the lower left corner of the square border //
            Paths[Index].move(to:CGPoint(x:Origin.x + Offset, y:Origin.y + Offset))

            // add a line to the upper left corner of the square border //
            Paths[Index].line(to:CGPoint(x:Origin.x + Offset, y:Origin.y + Dims.Square.WidthRet -
                              Offset))

            // add a line to the upper right corner of the square border //
            Paths[Index].line(to:CGPoint(x:Origin.x + Dims.Square.WidthRet - Offset, y:Origin.y +
                              Dims.Square.WidthRet - Offset))

            // add a line to the lower right corner of the square border //
            Paths[Index].line(to:CGPoint(x:Origin.x + Dims.Square.WidthRet - Offset, y:Origin.y +
                              Offset))

            // add a line to the lower left corner of the square border //
            Paths[Index].line(to:CGPoint(x:Origin.x + Offset, y:Origin.y + Offset))
         }}
      }
   }

   // update the square border colors //
   func UpdateSquareBorderColors(_ Opacity:CGFloat)
   {
      // set the opacity colors for the square border rectangles //
      (Colors[0], Colors[1], Colors[2]) = (NSColor.black.withAlphaComponent(Opacity), NSColor.white
      .withAlphaComponent(Opacity), NSColor.gray.withAlphaComponent(Opacity))
   }
}

// shape dimensions //
class Dimensions
{
   // the width //
   var Width:CGFloat = 0

   // the retina width //
   var WidthRet:CGFloat = 0

   // the height //
   var Height:CGFloat = 0

   // the retina height //
   var HeightRet:CGFloat = 0

   // the bounds //
   var Bounds:CGRect = CGRect()

   // the retina bounds //
   var BoundsRet:CGRect = CGRect()

   // the default initializer //
   init() {}

   // the initializer //
   init(_ XDim:CGFloat, YDim:CGFloat, X:CGFloat, Y:CGFloat)
   {
      // set the dimension values //
      (Width, WidthRet, Height, HeightRet, Bounds, BoundsRet) = (XDim, XDim * 2, YDim, YDim * 2,
      CGRect(x:X, y:Y, width:XDim, height:YDim), CGRect(x:X * 2, y:Y * 2, width:XDim * 2, height:
      YDim * 2))
   }
}

// the board dimensions class //
class BoardDimensions
{
   // the board dimensions //
   var Board:Dimensions = Dimensions(660, YDim:660, X:0, Y:0)

   // the border dimensions //
   var Border:Dimensions = Dimensions(30, YDim:30, X:0, Y:0)

   // the square dimensions //
   var Square:Dimensions = Dimensions(75, YDim:75, X:30, Y:30)

   // the inner board bounds dimensions //
   var InnerBounds:Dimensions = Dimensions(600, YDim:600, X:30, Y:30)

   // the capture bounds dimensions //
   var CaptureBounds:Dimensions = Dimensions(660, YDim:29, X:0, Y:631)

   // the board border bounds dimensions //
   var BorderBounds:NSBezierPath = NSBezierPath(rect:CGRect(x:30, y:30, width:1260, height:1260))

   // the square rectangles //
   var Squares:[CGRect] = []

   // the captured pieces rectangles //
   var Captures:[CGRect] = []

   // the board dimensions class initializer //
   init()
   {
      // loop over all the ranks and files //
      for Rank in 0...7 {for File in 0...7
      {
         // append a square rectangle //
         Squares.append(CGRect(x:CGFloat(File) * Square.WidthRet + Border.WidthRet, y:CGFloat(Rank)
                        * Square.WidthRet + Border.WidthRet, width:Square.WidthRet, height:Square
                        .WidthRet))
      }}

      // loop over thirty rectangles for the captured pieces //
      for Index in 0...29
      {
         // append a captured piece rectangle //
         Captures.append(CGRect(x:CGFloat(Index) * 44, y:1262, width:56, height:56))
      }

      // set the board border rectangle line width //
      BorderBounds.lineWidth = Border.WidthRet
   }

   // get the captured piece rectangle //
   func GetCaptureRect(_ Index:Int) -> CGRect {return Captures[Index]}

   // get the square rectangle for the specified board set and square location //
   func GetRect(_ Rotated:Bool, Point:APoint) -> CGRect
   {
      // return the square rectangle based on if the board is rotated or not //
      return Rotated ? Squares[63 - Point.YI * 8 - Point.XI] : Squares[Point.YI * 8 + Point.XI]
   }

   // get the square rectangle for the specified board set and square location //
   func GetRect(_ Rotated:Bool, Rank:Int, File:Int) -> CGRect
   {
      // return the square rectangle for the specified board set and square location //
      return GetRect(Rotated, Point:APoint(File, Y:Rank))
   }

   // get the square rectangle for the specified board set and square location //
   func GetRect(_ Rotated:Bool, Square:String) -> CGRect
   {
      // return the rectangle index from the point and square location //
      return GetRect(Rotated, Point:APoint((Int(Square) == nil) ? 0 : Int(Square)! % 8, Y:
                     (Int(Square) == nil) ? 0 : Int(Square)! / 8))
   }
}

// the chess board images class //
class ChessBoardImages
{
   // the chess board squares //
   let Squares:[String] = ["light_bamboo", "dark_bamboo", "light_white", "dark_blue", "light_white",
                           "dark_brown", "light_chalk", "dark_chalk", "light_cloth", "dark_cloth",
                           "light_felt", "dark_felt", "light_glass", "dark_glass", "light_gold",
                           "dark_gold", "light_granite", "dark_granite", "light_white", "dark_gray",
                           "light_white", "dark_green", "light_leather", "dark_leather",
                           "light_marble", "dark_marble", "light_plastic", "dark_plastic",
                           "light_rust", "dark_rust", "light_stone", "dark_stone", "light_silver",
                           "dark_silver", "light_white", "dark_teal", "light_wood", "dark_wood"]

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

   // the board images //
   var BoardImages:[CGImage] = []

   // the board dimensions //
   let Dims:BoardDimensions = BoardDimensions()

   // the core graphics functions //
   let Graphics:CoreGraphics = CoreGraphics()

   // the class initializer //
   init()
   {
      // create the context //
      let Context = Graphics.CreateContext(Dims.Board.WidthRet, Height:Dims.Board.WidthRet)

      // loop over all the types of chess board squares //
      for Index in 0..<Squares.count / 2
      {
         // loop over all the ranks and files //
         for Rank in 0...7 {for File in 0...7
         {
            // draw the light or dark square into the board context //
            Context.draw(Graphics.NSImageToCGImage(NSImage(named:Squares[Index * 2 + (Rank + File +
                         1) % 2])!, Size:Dims.Square.WidthRet), in:Dims.GetRect(false, Rank:Rank,
                         File:File), byTiling:false)
         }}

         // append the chess board image to the board images //
         BoardImages.append(Context.makeImage()!)
      }
   }

   // populate the chess board button titles //
   func PopulateButtonTitles(_ Button:NSPopUpButton)
   {
      // remove all the button items //
      Button.removeAllItems()

      // loop over all the board names and add the button title //
      for TheTitle in Boards {Button.addItem(withTitle:TheTitle)}
   }

   // get the board image //
   func GetBoardImage(_ BoardType:Int) -> CGImage {return BoardImages[BoardType]}
}

// the chess piece images class //
class ChessPieceImages
{
   // array of all the chess piece names //
   let Pieces:[String] = ["whitepawn", "whiteknight", "whitebishop", "whiterook", "whitequeen",
                          "whiteking", "blackpawn", "blackknight", "blackbishop", "blackrook",
                          "blackqueen", "blackking"]

   // the chess piece set names //
   let Sets:[String] = ["Standard Pieces", "3D Pieces", "3D Textured Pieces", "Figurine Pieces"]

   // the current piece set //
   var PieceSet:Int = 0

   // array of the chess piece nsimages //
   var NSImages:[NSImage] = []

   // array of the chess piece cgimages //
   var CGImages:[CGImage] = []

   // the board dimensions //
   let Dims:BoardDimensions = BoardDimensions()

   // the core graphics functions //
   let Graphics:CoreGraphics = CoreGraphics()

   // the chess piece images class initializer //
   init()
   {
      // loop over all the chess piece names to append the nsimage //
      for Num in 1...4 {for Piece in Pieces {NSImages.append(NSImage(named:Piece + "\(Num)")!)}}

      // loop over all the board sets and images //
      for TheImage in NSImages
      {
         // append a cgimage for the specified dimensions //
         CGImages.append(Graphics.NSImageToCGImage(TheImage, Size:Dims.Square.WidthRet))
      }
   }

   // populate the piece set type button titles //
   func PopulateButtonTitles(_ Button:NSPopUpButton)
   {
      // remove all the button items //
      Button.removeAllItems()

      // loop over all the piece sets and add the button title //
      for TheTitle in Sets {Button.addItem(withTitle:TheTitle)}
   }

   // update the piece set type //
   func UpdatePieceSet(_ Set:Int) {PieceSet = Set}

   // get the piece nsimage //
   func GetPieceNSImage(_ Piece:Int) -> NSImage
   {
      // get the piece image if is a white or black piece //
      return Piece > 0 ? NSImages[Piece + PieceSet * 12 - 1] : NSImages[PieceSet * 12 - Piece + 5]
   }

   // get the piece cgimage //
   func GetPieceCGImage(_ Piece:Int) -> CGImage
   {
      // get the piece image if is a white or black piece //
      return Piece > 0 ? CGImages[Piece + PieceSet * 12 - 1] : CGImages[PieceSet * 12 - Piece + 5]
   }
}

// the menu items class //
class MenuItems
{
   // the index variable in a list //
   var Index:Int = 0

   // the dictionary count //
   var DictionaryCount:Int = 0

   // the menu item dictionaries //
   var Items:[[Int:Bool]] = []

   // the menu items initializer //
   init()
   {
      // for all menu items on //
      AddDictionary(Array(1...17))

      // for when engine is thinking //
      AddDictionary([1, 3, 14])

      // for when have an attached sheet //
      AddDictionary([1, 3])

      // for when in a panel //
      AddDictionary([3])

      // for when in the analysis mode //
      AddDictionary([1, 2, 3, 5, 6, 9, 11, 14, 16, 17])

      // for when in diagram mode //
      AddDictionary([1, 2, 3, 6, 16, 17])

      // for when in the position editor //
      AddDictionary([1, 2, 3, 6, 16, 17])

      // for when in timed game mode //
      AddDictionary([1, 2, 3, 5, 6, 16, 17])

      // the dictionary count //
      DictionaryCount = Items.count

      // loop over all the dictionaries //
      for Index in 0..<DictionaryCount
      {
         // go stop menu item button is false if human opponent, game over, or play timed game //
         Items.append(Items[Index].merging([14:false]) {(_, new) in new})
      }

      // loop over all the dictionaries //
      for Index in 0..<DictionaryCount * 2
      {
         // append the fullscreen version of the dictionary //
         Items.append(Items[Index].merging([3:false, 16:false]) {(_, new) in new})
      }
   }

   // add a dictionary to the menu item states //
   func AddDictionary(_ OnVals:[Int])
   {
      // append an empty dictionary //
      Items.append([:])

      // update the boolean status for the specified tags //
      for Index in 1...17 {Items[Items.count - 1].updateValue(OnVals.contains(Index), forKey:Index)}
   }

   // get the specified menu item state //
   func MenuItemState(_ Tag:Int, Dictionary:Int, FullScreen:Bool, HumanOrGameOverOrTimed:Bool) ->
                      Bool
   {
      // set the index to the current dictionary //
      Index = Dictionary

      // if human opponent, game over, or playing timed game, increment index by dictionary count //
      if HumanOrGameOverOrTimed {Index += DictionaryCount}

      // if fullscreen, increment index by two dictionary counts //
      if FullScreen {Index += 2 * DictionaryCount}

      // if the menu item tag is in the current dictionary //
      return (Items[Index][Tag] == nil) ? true : Items[Index][Tag]!
   }
}

// the chess board data class //
class ChessBoardData
{
   // all the captured pieces //
   var Captures:[Int] = []

   // square label functions //
   var SquareLabel:ASquareLabel = ASquareLabel()

   // the list of capture squares //
   var CaptureSquares:[String] = []

   // the list of threatened squares //
   var ThreatSquares:[String] = []

   // the board layout //
   var Layout:[[Int]] = Array(repeating:Array(repeating:0, count:8), count:8)

   // the save board layout //
   var SavedLayout:[[Int]] = Array(repeating:Array(repeating:0, count:8), count:8)

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

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

   // save the board layout //
   func SaveLayout() {SavedLayout = Layout}

   // restore the board layout //
   func RestoreLayout() {Layout = SavedLayout}

   // store the captured piece //
   func StoreCapture(_ Piece:Int)
   {
      // store the captured piece if there was one //
      if Piece != 0 {Captures.append(Piece)}

      // sort the captured pieces //
      Captures = Captures.sorted(by:>).filter {$0 > 0} + Captures.sorted().filter {$0 < 0}
   }

   // update the start and end squares for the move //
   func UpdateStartAndEndSquaresForMove(_ UCIMove:String, Dims:BoardDimensions, Rotated:Bool)
   {
      // set the start square for the move //
      Start.Set(SquareLabel.SquareFromUCI(UCIMove, Start:true, Dims:Dims, Rotated:Rotated))

      // set the end square for the move //
      End.Set(SquareLabel.SquareFromUCI(UCIMove, Start:false, Dims:Dims, Rotated:Rotated))
   }

   // update the capture and threaten highlights //
   func UpdateCaptureThreatLists(_ CaptureSquares:[String], ThreatSquares:[String])
   {
      // store the list of capture squares and threatened squares //
      (self.CaptureSquares, self.ThreatSquares) = (CaptureSquares, ThreatSquares)

      // remove the last element from the list of capture and threaten squares since empty //
      (_, _) = (self.CaptureSquares.popLast(), self.ThreatSquares.popLast())
   }

   // initialize the board layout //
   func InitializeLayout()
   {
      // initialize the board layout //
      Layout = Array(repeating:Array(repeating:0, count:8), count:8)

      // clear the captures //
      Captures.removeAll()
   }

   // return if the square is not equal to the end square for the move //
   func SquaresNotEqual() -> Bool {return !Start.Equals(End)}

   // set the start square to be equal to the end square of the move //
   func SquaresEqual() {Start.Set(End)}

   // update the board layout value for the specified square on the board //
   func BFU(_ Square:APoint, _ Value:Int)
   {
      // update the board layout value for the specified square on the board //
      Layout[max(min(Square.XI, 7), 0)][max(min(Square.YI, 7), 0)] = Value
   }

   // update the board layout value for the specified file and rank //
   func BFU(_ File:Int, _ Rank:Int, _ Value:Int)
   {
      // update the board layout value for the specified file and rank //
      Layout[max(min(File, 7), 0)][max(min(Rank, 7), 0)] = Value
   }

   // return the board layout value for the specified file and rank //
   func BFV(_ File:Int, _ Rank:Int) -> Int
   {
      // return the board layout value for the specified file and rank //
      return Layout[max(min(File, 7), 0)][max(min(Rank, 7), 0)]
   }

   // return the board layout value for the specified square on the board //
   func BFV(_ Square:APoint) -> Int
   {
      // return the board layout value for the specified square on the board //
      return Layout[max(min(Square.XI, 7), 0)][max(min(Square.YI, 7), 0)]
   }

   // return the board layout value at the end square of the move //
   func EndSquareValue() -> Int {return BFV(End)}

   // return the board layout value at the start square of the move //
   func StartSquareValue() -> Int {return BFV(Start)}

   // remove the pawn captured by en passant //
   func RemovePawnCapturedEnPassant() {BFU(End.XI, Start.YI, 0)}

   // update the board layout value at the end square of the move //
   func EndSquareValueUpdate(_ MovingPiece:Int) {BFU(End, MovingPiece)}

   // remove the start square piece //
   func RemoveStartSquarePiece() {BFU(Start.XI, Start.YI, 0)}

   // return if board layout value at specified rank and file is equal to specified value //
   func BFE(_ File:Int, _ Rank:Int, _ Value:Int) -> Bool
   {
      // return if board layout value at specified rank and file is equal to specified value //
      return Layout[max(min(File, 7), 0)][max(min(Rank, 7), 0)] == Value
   }

   // return if board layout value at specified rank and file not equal to specified value //
   func BFNE(_ File:Int, _ Rank:Int, _ Value:Int) -> Bool
   {
      // return if board layout value at specified rank and file not equal to specified value //
      return Layout[max(min(File, 7), 0)][max(min(Rank, 7), 0)] != Value
   }

   // return if board layout absolute value at rank and file is equal to specified value //
   func BFAE(_ File:Int, _ Rank:Int, _ Value:Int) -> Bool
   {
      // return if board layout absolute value at rank and file is equal to specified value //
      return abs(Layout[max(min(File, 7), 0)][max(min(Rank, 7), 0)]) == Value
   }

   // store the chess board data //
   func Store(_ BPData:ChessBoardData)
   {
      // store the board data values //
      (Layout, CaptureSquares, ThreatSquares, Captures) = (BPData.Layout, BPData.CaptureSquares,
      BPData.ThreatSquares, BPData.Captures)

      // store the start square for the move from the board and pieces data //
      Start.Set(BPData.Start)

      // store the end square for the move from the board and pieces data //
      End.Set(BPData.End)
   }
}

// the class for a point //
class APoint
{
   // the integer value of x //
   var XI:Int = 0

   // the integer value of y //
   var YI:Int = 0

   // the cgfloat value of x //
   var XF:CGFloat = 0

   // the cgfloat value of y //
   var YF:CGFloat = 0

   // the x coordinate of the square center in display points //
   var XC:CGFloat = 0

   // the y coordinate of the square center in display points //
   var YC:CGFloat = 0

   // the bounded x location of the mouse in display points //
   var XL:CGFloat = 0

   // the bounded y location of the mouse in display points //
   var YL:CGFloat = 0

   // check if point is equal to the current point //
   func Equals(_ Point:APoint) -> Bool {return (XI == Point.XI) && (YI == Point.YI)}

   // get the cgpoint of the xl and yl values //
   func GetCGPoint() -> CGPoint {return CGPoint(x:XL, y:YL)}

   // get the board coordinates of the current location //
   func GetCoord() -> CGPoint {return CGPoint(x:XC, y:YC)}

   // the initializer //
   init() {}

   // the initializer //
   init(_ X:Int, Y:Int) {XI = X; YI = Y; XF = CGFloat(X); YF = CGFloat(Y)}

   // set the point from the x and y square indices //
   func Set(_ X:Int, Y:Int, Dims:BoardDimensions, Rotated:Bool)
   {
      // update the indices and float values //
      (XI, YI, XF, YF) = (max(min(X, 7), 0), max(min(Y, 7), 0), CGFloat(X), CGFloat(Y))

      // set the x and y coordinates of the square center in display points //
      (XC, YC) = (CGFloat(Rotated ? 7 - XI : XI) * Dims.Square.Width + Dims.Square.Bounds.midX,
      CGFloat(Rotated ? 7 - YI : YI) * Dims.Square.Width + Dims.Square.Bounds.midX)
   }

   // set the point from the x and y coordinates and if rotated or not //
   func Set(_ Location:CGPoint, Dims:BoardDimensions, Rotated:Bool)
   {
      // set the bounded x and y locations of the mouse in display points //
      (XL, YL) = (min(max(Location.x, Dims.Square.Bounds.midX), Dims.Board.Width - Dims.Square
      .Bounds.midX), min(max(Location.y, Dims.Square.Bounds.midX), Dims.Board.Width - Dims.Square
      .Bounds.midX))

      // set the x and y indices of the possible square value from 0 to 7 //
      (XI, YI) = (max(min(Int((XL - Dims.Border.Width) / Dims.Square.Width), 7), 0), max(min(Int((YL
      - Dims.Border.Width) / Dims.Square.Width), 7), 0))

      // set the x and y coordinates of the square center in display points //
      (XC, YC) = (CGFloat(XI) * Dims.Square.Width + Dims.Square.Bounds.midX, CGFloat(YI) * Dims
      .Square.Width + Dims.Square.Bounds.midX)

      // if the board is rotated //
      if Rotated {XI = 7 - XI; YI = 7 - YI}

      // save the cgfloat value of xi and yi //
      (XF, YF) = (CGFloat(XI), CGFloat(YI))
   }

   // set the point to the given point //
   func Set(_ P:APoint)
   {
      // initialize the variables to zero //
      (XI, YI, XF, YF, XC, YC, XL, YL) = (P.XI, P.YI, P.XF, P.YF, P.XC, P.YC, P.XL, P.YL)
   }
}

// the game info list data source implementation //
class GameInfoDataSource:NSObject, NSTableViewDataSource
{
   // a list of game info //
   var GameInfoList:[NSAttributedString] = []

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

   // refresh the game info list //
   func RefreshGameInfoList(_ SavedGameInfoList:[String], GameInfoFont:FontsSet)
   {
      // clear the game info list //
      GameInfoList.removeAll()

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

      // loop over the game info list //
      for TheInfo in SavedGameInfoList {if !TheInfo.isEmpty
      {
         // append the attributed move string //
         GameInfoList.append(GameInfoFont.GetString("\n" + TheInfo + "\n", White:false))

         // append the row height //
         Heights.append(GameInfoFont.GetStringHeight(GameInfoList.last!, Width:250))
      }}
   }

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

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

// the pgn moves list data source implementation //
class PGNMovesDataSource:NSObject, NSTableViewDataSource
{
   // the board data list //
   var DataList:[ChessBoardData] = [ChessBoardData()]

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

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

   // get the chess board data at the given row //
   func GetData(_ Row:Int) -> ChessBoardData {return DataList[min(Row, DataList.count - 1)]}

   // get the text for the pgn moves table view row //
   func GetViewText(_ Row:Int) -> NSAttributedString {return PGNS[min(Row, PGNS.count - 1)]}

   // update the board data //
   func Update(_ Row:Int, BPData:ChessBoardData)
   {
      // while data bounds exceeded by the row //
      while Row >= DataList.count {DataList.append(ChessBoardData())}

      // update the board data //
      DataList[Row].Store(BPData)
   }

   // refresh the pgn moves list //
   func RefreshPGNMovesList(_ SavedPGNS:[String], MoveListFont:FontsSet)
   {
      // clear the move list //
      PGNS.removeAll()

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

      // loop over the moves list //
      for TheMove in SavedPGNS
      {
         // append the attributed move string //
         PGNS.append(MoveListFont.GetString("\n" + TheMove, White:false))

         // append the row height //
         Heights.append(MoveListFont.GetStringHeight(PGNS.last!, Width:250) + 25)
      }
   }

   // return the row height //
   func RowHeight(_ Row:Int) -> CGFloat {return Heights[min(Row, Heights.count - 1)]}

   // return the last row number //
   func LastRow() -> Int {return max(0, PGNS.count - 1)}

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

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

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

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

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

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

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

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

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

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

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

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

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

      // remove the last object in the game list since it is empty //
      _ = List.popLast()

      // set the list of game indices //
      Indices = Array(0..<Int(List.count))

      // save the original game list and indices //
      (ListOriginal, IndicesOriginal) = (List, Indices)
   }

   // condense the game list related to the current search string //
   func CondenseGameList(_ SearchString:String)
   {
      // reset the list if the search string is empty //
      if SearchString.isEmpty {(List, Indices) = (ListOriginal, IndicesOriginal); return}

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

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

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

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

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

   // refresh the game list //
   func RefreshGameList(_ GameListFont:FontsSet)
   {
      // clear the attributed game list //
      AttributedList.removeAll()

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

      // loop over the game list //
      for TheGame in List
      {
         // append the attributed game string //
         AttributedList.append(GameListFont.GetString("\n" + TheGame + "\n", White:false))

         // append the row height //
         Heights.append(GameListFont.GetStringHeight(AttributedList.last!, Width:605))
      }
   }

   // get the specified row attributed string //
   func GetRow(_ Row:Int) -> NSAttributedString {return AttributedList[Row]}

   // get the selected row in the table that is highlighted blue //
   func SelectedRow(_ Row:Int, GameListFont:FontsSet) -> NSAttributedString
   {
      // return the attributed string for the selected row with white font //
      return GameListFont.GetString(AttributedList[Row].string, White:true)
   }

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

// the time clock display //
class TimeClockDisplay:NSView
{
   // the second hand origins //
   var SecondsOrigin:[CGPoint] = []

   // the second hand endpoints //
   var SecondsEndpoint:[CGPoint] = []

   // the minute hand endpoint //
   var MinutesEndpoint:CGPoint = CGPoint()

   // the hour hand endpoint //
   var HoursEndpoint:CGPoint = CGPoint()

   // the start point //
   var StartPoint:CGPoint = CGPoint()

   // the end point //
   var EndPoint:CGPoint = CGPoint()

   // the draw rectangles for the second hand //
   var RectsSec:[CGRect] = []

   // the clock face layer //
   var ClockFaceLayer:CGLayer?

   // the clock face hours path //
   var ClockFaceHoursPath:CGMutablePath = CGMutablePath()

   // the clock face minutes path //
   var ClockFaceMinutesPath:CGMutablePath = CGMutablePath()

   // the total elapsed time //
   var ElapsedTime:Int = 0

   // whether the clock face needs to be fully refreshed //
   var RefreshAll:Bool = true

   // radians for a clock face //
   var Radians:CGFloat = 0

   // the cosine of the radians //
   var Cosine:CGFloat = 0

   // the sine of the radians //
   var Sine:CGFloat = 0

   // the radius of the clock //
   var Radius:CGFloat = 0

   // the diameter of the clock //
   var Diameter:CGFloat = 0

   // the blended colors //
   var BlendedColors:[NSColor] = []

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

   // the background color //
   var BackgroundColor:NSColor = NSColor()

   // the border color //
   var BorderColor:NSColor = NSColor()

   // the bezel color //
   var BezelColor:NSColor = NSColor()

   // the blended background color //
   var BlendedBackgroundColor:NSColor = NSColor()

   // the background color opacity //
   var BackgroundColorOpacity:CGFloat = 0

   // the darkened clock background color //
   var DarkenedColor:NSColor = NSColor()

   // the lightened clock background color //
   var LightenedColor:NSColor = NSColor()

   // the view bounds //
   var ViewBounds:CGRect = CGRect()

   // the bezel bounds //
   var BezelBounds:CGRect = CGRect()

   // clock circles to color //
   var Circles:[CGRect] = []

   // the core graphics functions //
   let Graphics:CoreGraphics = CoreGraphics()

   // update the clock window color //
   func UpdateClockWindowColor(_ Color:NSColor) {WindowColor = Color; RefreshAll = true}

   // update the clock background color //
   func UpdateClockBackgroundColor(_ Color:NSColor) {BackgroundColor = Color; RefreshAll = true}

   // update the clock border color //
   func UpdateClockBorderColor(_ Color:NSColor) {BorderColor = Color; RefreshAll = true}

   // update the clock background opacity //
   func UpdateClockBackgroundOpacity(_ Opacity:Int)
   {
      // update the clock background color opacity and set refresh all to true //
      (BackgroundColorOpacity, RefreshAll) = (CGFloat(Opacity) / 100, true)
   }

   // create the blended colors //
   func CreateBlendedColors()
```

```
      // set the blended background color //
      BlendedBackgroundColor = WindowColor.blended(withFraction:BackgroundColorOpacity, of:
                                                   BackgroundColor)!

      // clear the list of blended colors //
      BlendedColors.removeAll()

      // set the clock colors //
      (LightenedColor, DarkenedColor, BezelColor) = (BlendedBackgroundColor.blended(withFraction:
      0.05, of:NSColor.white)!, BlendedBackgroundColor.blended(withFraction:0.05, of:NSColor
      .black)!, BlendedBackgroundColor.blended(withFraction:0.1, of:NSColor.white)!)

      // loop over the number of circles to color //
      for Index in 0..<Circles.count
      {
         // append a color blend from the darkened and lightened colors //
         BlendedColors.append(DarkenedColor.blended(withFraction:min(abs(tan(CGFloat(Index * 10))),
                              1), of:LightenedColor)!)
      }
   }

   // initialize the clock //
   func Initialize()
   {
      // allow concurrent drawing, and set clock radius and diameter //
      (canDrawConcurrently, Radius, Diameter) = (true, frame.size.width, frame.size.width * 2)

      // translate the clock origin to be the center of the view //
      translateOrigin(to:CGPoint(x:Radius + 0.5, y:Radius + 0.5))

      // set the view and bezel bounds //
      (ClockFaceLayer, ViewBounds, BezelBounds) = (Graphics.CreateLayer(Diameter, Height:Diameter),
      CGRect(x:-Radius, y:-Radius, width:Diameter, height:Diameter), CGRect(x:5 - Radius, y:5 -
      Radius, width:Diameter - 10, height:Diameter - 10))

      // translate the clock face origin to the middle of the layer //
      ClockFaceLayer.context?.translateBy(x:Radius, y:Radius)

      // loop over all the seconds in the clock face //
      for Index in 0...59
      {
         // set the current radians //
         Radians = CGFloat(Index) * CGFloat.pi / 30

         // set the current cosine and sine //
         (Cosine, Sine) = (cos(Radians), sin(Radians))

         // set the start and end points of the tick mark //
         (StartPoint, EndPoint) = (CGPoint(x:0.93 * Radius * Cosine, y:0.93 * Radius * Sine),
         CGPoint(x:0.97 * Radius * Cosine, y:0.97 * Radius * Sine))

         // if at an hour mark //
         if (Index % 5) == 0
         {
            // move the current hour mark to the beginning position //
            ClockFaceHoursPath.move(to:StartPoint)

            // add a line to create the current hour mark //
            ClockFaceHoursPath.addLine(to:EndPoint)
         }
         // at a minute mark //
         else
         {
            // move the current minute mark to the beginning position //
            ClockFaceMinutesPath.move(to:StartPoint)

            // add a line to create the current minute mark //
            ClockFaceMinutesPath.addLine(to:EndPoint)
         }
      }

      // loop over all the circles to color //
      for Index in (1...Int(Radius)).reversed()
      {
         // append a clock circle to color //
         Circles.append(CGRect(x:-Index, y:-Index, width:Index * 2, height:Index * 2))
      }

      // loop over all the possible second hand positions //
      for Index in 0...599
      {
         // set the current radians //
         Radians = CGFloat.pi * CGFloat(Index) / 300

         // set the cosine and sine //
         (Cosine, Sine) = (cos(Radians), sin(Radians))

         // append an origin of the second hand //
         SecondsOrigin.append(CGPoint(x:5 * Sine, y:5 * Cosine))

         // append an endpoint of the second hand //
         SecondsEndpoint.append(CGPoint(x:44 * Sine, y:44 * Cosine))

         // append a rectangle to draw the second hand in //
         RectsSec.append(CGRect(x:min(SecondsOrigin.last!.x, SecondsEndpoint.last!.x) - 1, y:
                  min(SecondsOrigin.last!.y, SecondsEndpoint.last!.y) - 1, width:abs(SecondsEndpoint
                  .last!.x - SecondsOrigin.last!.x) + 2, height:abs(SecondsEndpoint.last!.y -
                  SecondsOrigin.last!.y) + 2))
      }
   }

   // draw the clock //
   override func draw(_ Rect:CGRect)
   {
      // return if the clock face layer doesn't exist //
      if ClockFaceLayer == nil {return}

      // draw the clock face layer into the current context //
      NSGraphicsContext.current?.cgContext.draw(ClockFaceLayer, in:bounds)

      // set the stroke color //
      NSGraphicsContext.current?.cgContext.setStrokeColor(BorderColor.cgColor)

      // set the line type to round //
      NSGraphicsContext.current?.cgContext.setLineCap(CGLineCap.round)

      // set the line width to 4 //
      NSGraphicsContext.current?.cgContext.setLineWidth(4)

      // start the hour hand from the origin //
      NSGraphicsContext.current?.cgContext.move(to:CGPoint.zero)

      // add a line for the hour hand //
      NSGraphicsContext.current?.cgContext.addLine(to:HoursEndpoint)

      // stroke the hour hand //
      NSGraphicsContext.current?.cgContext.strokePath()

      // set the line width to 2 //
      NSGraphicsContext.current?.cgContext.setLineWidth(2)

      // start the minute hand from the origin //
      NSGraphicsContext.current?.cgContext.move(to:CGPoint.zero)

      // add a line for the minute hand //
      NSGraphicsContext.current?.cgContext.addLine(to:MinutesEndpoint)

      // stroke the minute hand //
      NSGraphicsContext.current?.cgContext.strokePath()

      // set the line width to 1 //
      NSGraphicsContext.current?.cgContext.setLineWidth(1)

      // start the second hand from the origin //
      NSGraphicsContext.current?.cgContext.move(to:SecondsOrigin[ElapsedTime % 600])

      // add a line for the second hand //
      NSGraphicsContext.current?.cgContext.addLine(to:SecondsEndpoint[ElapsedTime % 600])

      // stroke the second hand //
      NSGraphicsContext.current?.cgContext.strokePath()
   }

   // update the clock face layer //
   func UpdateClockFaceLayer()
   {
      // clear the clock face layer //
      ClockFaceLayer.context?.clear(ViewBounds)

      // create the blended colors //
      CreateBlendedColors()

      // loop over all the clock circles to draw //
      for Index in 0..<Circles.count
      {
         // set the current fill color from the list of blended colors //
         ClockFaceLayer.context?.setFillColor(BlendedColors[Index].cgColor)

         // draw a circle in the given ellipse //
         ClockFaceLayer.context?.fillEllipse(in:Circles[Index])
      }

      // set the fill color to the border color //
      ClockFaceLayer.context?.setFillColor(BorderColor.cgColor)

      // draw the center of the clock face //
      ClockFaceLayer.context?.fillEllipse(in:CGRect(x:-10, y:-10, width:20, height:20))

      // set the stroke color to the bezel color //
      ClockFaceLayer.context?.setStrokeColor(BezelColor.cgColor)

      // set the line width for the bezel //
      ClockFaceLayer.context?.setLineWidth(10)

      // draw the bezel //
      ClockFaceLayer.context?.strokeEllipse(in:BezelBounds)

      // set the stroke color to the border color //
      ClockFaceLayer.context?.setStrokeColor(BorderColor.cgColor)

      // set the line type to round //
      ClockFaceLayer.context?.setLineCap(CGLineCap.round)

      // set the line width for the hour marks //
      ClockFaceLayer.context?.setLineWidth(4)

      // add the hour marks to the clock face //
      ClockFaceLayer.context?.addPath(ClockFaceHoursPath)

      // stroke the hour marks //
      ClockFaceLayer.context?.strokePath()

      // set the line width for the minute marks //
      ClockFaceLayer.context?.setLineWidth(2)

      // add the minute marks to the clock face //
      ClockFaceLayer.context?.addPath(ClockFaceMinutesPath)

      // stroke the minute marks //
      ClockFaceLayer.context?.strokePath()
   }

   // update the clock //
   func UpdateClock(_ Time:Int)
   {
      // update the elapsed time and radians //
      (ElapsedTime, Radians) = (Time, CGFloat.pi * CGFloat(Time) / 18000)

      // set the minute hand endpoint //
      MinutesEndpoint = CGPoint(x:39 * sin(Radians), y:39 * cos(Radians))

      // set the radians for the hour hand //
      Radians = CGFloat.pi * CGFloat(ElapsedTime) / 216000

      // set the hours hand endpoint //
      HoursEndpoint = CGPoint(x:25 * sin(Radians), y:25 * cos(Radians))

      // if the entire clock needs to be refreshed //
      if RefreshAll
      {
         // update the clock face layer //
         UpdateClockFaceLayer()

         // update the entire display //
         display(bounds)

         // set refresh to false //
         RefreshAll = false
      }
      // if entire clock doesn't need to be refreshed //
      else
      {
         // update the display periodically for the minute hand //
         if (ElapsedTime % 50) == 0
         {
            // update the display for the minute hand //
            display(CGRect(x:min(0, MinutesEndpoint.x) - 1, y:min(0, MinutesEndpoint.y) - 1,
                    width:abs(MinutesEndpoint.x) + 2, height:abs(MinutesEndpoint.y) + 2))
         }

         // update the second hand //
         display(RectsSec[ElapsedTime % 600])
      }
   }
}

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

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

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

// the game gallery view //
class Gallery:NSView
{
   // the frames and movie creator //
   @IBOutlet var FramesMovie:FramesMovieCreator!

   // the sketch of a diagram //
   var Sketch:Sketching = Sketching()

   // the information dictionary //
   var Inf:Information = Information()

   // the game sound effects //
   @IBOutlet var GameSounds:GameSoundEffects!

   // the number of layers in the gallery //
   var Layers:Int = 0

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

   // the snapshot image representation //
   var SnapshotImageRep:NSBitmapImageRep!

   // the tracking area //
   var TrackingArea:NSTrackingArea!

   // the frame size //
   var FrameSize:CGFloat = 0

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

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

   // the current position in the gallery //
   var CurrentPosition:Int = 0

   // the core graphics functions //
   let Graphics:CoreGraphics = CoreGraphics()

   // override the draw function //
   override func draw(_ Rect:CGRect)
   {
      // if have an image ready, update the sketch image to the screen //
      if Sketch.Displaying {Sketch.SketchImages.UpdateScreen(bounds); Sketch.Displaying = false}
   }

   // refresh the cursor //
   func RefreshCursor() {Sketch.Tracker.RefreshCursor()}

   // update the gallery color //
   func UpdateGalleryColor(_ Color:String)
   {
      // play the game sound of a click //
      GameSounds.Play("click")

      // set that currently diagramming //
      Inf.TurnOnVB("VB", Key:"Diagramming")

      // set the current diagram cursor //
      Sketch.UpdateColor(Color)
   }

   // if the mouse entered the tracking area //
   override func mouseEntered(with Event:NSEvent) {RefreshCursor()}

   // if the mouse exited the tracking area //
   override func mouseExited(with Event:NSEvent) {RefreshCursor()}

   // if the mouse moved in the tracking area //
   override func mouseMoved(with Event:NSEvent) {RefreshCursor()}

   // override mouse up //
   override func mouseUp(with Event:NSEvent) {RefreshCursor()}

   // override mouse dragged //
   override func mouseDragged(with Event:NSEvent) {RefreshCursor()}

   // override mouse down //
   override func mouseDown(with Event:NSEvent)
   {
      // if not reviewing and diagramming //
      if !Inf.Vals("Reviewing").VB && Inf.Vals("Diagramming").VB
      {
         // play the sound of a marker pen starting to write //
         GameSounds.Play("marker")

         // begin the sketch path //
         Sketch.BeginPath()
      }
   }

   // take a snapshot of the gallery //
   func GallerySnapshot() -> NSBitmapImageRep
   {
      // get the snapshot image representation from the context //
      SnapshotImageRep = NSBitmapImageRep(cgImage:Sketch.Snapshot())

      // set the snapshot image representation size //
      SnapshotImageRep.size = bounds.size

      // return the bitmap image rep //
      return SnapshotImageRep
   }

   // go left in the game gallery //
   func GoLeft()
   {
      // update the current position //
      UpdateCurrentPosition(max(0, CurrentPosition - 1))

      // update the game gallery slider to the current position //
      Board.GameGallerySlider.intValue = Int32(CurrentPosition)
   }

   // update offset //
   func UpdateOffset(_ Offset:CGFloat) {Sketch.UpdateOffset(Offset); RefreshCursor()}

   // go right in the game gallery //
   func GoRight()
   {
      // if the current position is less than the maximum position //
      if CurrentPosition < FramesMovie.MaxPosition
      {
         // increment the current position by one //
         CurrentPosition += 1

         // if reached the maximum position and reviewing the gallery //
         if (CurrentPosition == FramesMovie.MaxPosition) && Inf.Vals("Reviewing").VB
         {
            // stop the gallery review timer //
            Board.GalleryReviewTimer.invalidate()

            // set that no longer reviewing //
            Inf.TurnOffVB("VB", Key:"Reviewing")

            // set the gallery buttons to state true //
            Board.GalleryTabs.UpdateState(true)
         }
      }
      // if already at maximum position and reviewing gallery //
      else if Inf.Vals("Reviewing").VB
      {
         // set the current position to zero //
         CurrentPosition = 0
      }

      // update the game gallery slider to the current value //
      Board.GameGallerySlider.intValue = Int32(CurrentPosition)

      // update the gallery offset //
      UpdateOffset(FramesMovie.GetX(CurrentPosition))
   }

   // write a gallery movie //
   func WriteMovie(_ Path:URL)
   {
      // write a gallery movie //
      FramesMovie.WriteMovie(Sketch.Diagram, VidWidth:Width, VidHeight:Height, Path:Path, Layered:
                             true)
   }

   // erase the gallery //
   func EraseGallery()
   {
      // set that no longer reviewing the gallery //
      Inf.TurnOffVB("VB", Key:"Reviewing")

      // remove the tracking area //
      removeTrackingArea(TrackingArea)

      // clear the sketch information //
      Sketch.ClearInfo()
   }

   // update the gallery current position //
   func UpdateCurrentPosition(_ CurrentPosition:Int)
   {
      // update the current position in the gallery //
      self.CurrentPosition = CurrentPosition

      // update the gallery offset //
      UpdateOffset(FramesMovie.GetX(CurrentPosition))
   }

   // override isopaque //
   override var isOpaque: Bool {return true}

   // override accepts first responder //
   override var acceptsFirstResponder: Bool {return true}

   // initialize the gallery //
   func Initialize()
   {
      // set diagramming and reviewing to false //
      Inf.Update("VR", VBItems:["Diagramming":false, "Reviewing":false])

      // update the frame size to retina, current position, and height and width //
      (canDrawConcurrently, FrameSize, CurrentPosition, Height, Width) = (true, FrameSize
      * 2, 0, frame.size.height * 2, frame.size.width * 2)

      // update the gallery button states //
      Board.GalleryTabs.UpdateState(true)

      // create the frame set //
      FramesMovie.Create(FrameSize, Rows:Layers, FramesWide:Int(Width / FrameSize))

      // initialize the sketch //
      Sketch.Initialize(self as NSView, Width:Width, Height:Height, Length:CGFloat(FramesMovie
              .TotalFrames) * FrameSize, VidTime:50, Margin:2)

      // draw the frame set into the sketch background //
      Sketch.Background.context?.draw(FramesMovie.Layer, at:CGPoint(x:0, y:0))

      // reset the frames movie layer //
      FramesMovie.Layer = Graphics.CreateLayer(1, Height:1)

      // set the tracking area //
      TrackingArea = NSTrackingArea(rect:bounds, options:[.mouseEnteredAndExited, .mouseMoved,
                                    .activeAlways], owner:self, userInfo:nil)

      // add the current tracking area to the view //
      addTrackingArea(TrackingArea)
   }
}

// the colors view //
class ColorsView:NSView
{
   // the colors view unique identifier //
   var Identifier:Int = 0

   // the hue layer //
   var HueLayer:CGLayer!

   // the current hue rectangle //
   var CurrentRect:CGRect = CGRect()

   // the color hues //
   var Hues:[NSColor] = []

   // the current hue //
   var CurrentHue:Int = 0

   // the previous hue //
   var PreviousHue:Int = -1

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

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

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

   // the core graphics functions //
   let Graphics:CoreGraphics = CoreGraphics()

   // the maximum hue index //
   func MaxHue() -> Int {return 122}

   // get the hue index value //
   func GetHueValue() -> Int {return CurrentHue}

   // the current hue //
   func GetCurrentHue() -> NSColor {return Hues[CurrentHue]}

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

      // compute the current hue //
      ComputeCurrentHue()

      // if the previous hue equals the current hue //
      if PreviousHue == CurrentHue {return} else {PreviousHue = CurrentHue}

      // switch over the colors view unique identifier //
      switch Identifier
      {
         // set the board labels font color //
         case 0 : Board.UpdateBoardLabelsFont(self)

         // set the game clock border color //
         case 1 : Board.SetGameClockColor()

         // set the game clock background color //
         case 2 : Board.SetGameClockBackgroundColor()

         // set game info font color //
         case 3 : Board.UpdateGameInfoFont(self)

         // set the move list font color //
         case 4 : Board.UpdateMoveListFont(self)

         // set the game list font color //
         case 5 : Board.UpdateGameListFont(self)

         // set the window background color //
         case 6 : Board.UpdateWindowBackgroundColor()

         // the default switch case //
         default : break
      }

      // update the view //
      display(bounds)
   }

   // override mouse down //
   override func mouseDown(with Event:NSEvent) {UpdateCurrentHue(Event)}

   // override mouse dragged //
   override func mouseDragged(with Event:NSEvent) {UpdateCurrentHue(Event)}

   // override mouse up //
   override func mouseUp(with Event:NSEvent) {UpdateCurrentHue(Event)}

   // compute the current hue //
   func ComputeCurrentHue() {CurrentHue = min(max(Int((Mouse.x - 30) / Width), 0), Hues.count - 1)}

   // the draw rect function //
   override func draw(_ Rect:CGRect)
   {
      // draw the hue layer into the current context //
      NSGraphicsContext.current?.cgContext.draw(HueLayer, in:bounds)

      // get the current rectangle based on the current hue //
      CurrentRect = (CurrentHue == 0) ? CGRect(x:0.5, y:1, width:Width, height:38) : ((CurrentHue ==
                    Hues.count - 1) ? CGRect(x:CGFloat(CurrentHue) * Width - 0.5, y:1, width:Width,
                    height:38) : CGRect(x:CGFloat(CurrentHue) * Width, y:1, width:Width, height:38))

      // set the fill hue to the current hue //
      Hues[CurrentHue].setFill()

      // fill the current hue rectangle with the current hue //
      NSBezierPath(rect:CurrentRect).fill()

      // set the border color //
      NSColor.black.set()

      // draw the current hue rectangle border //
      NSBezierPath(rect:CurrentRect).stroke()
   }

   // initialize the color object //
   func Initialize(_ ChosenHue:Int)
   {
      // allow concurrent drawing //
      canDrawConcurrently = true

      // loop over all the hues //
      for Index in 0...119
      {
         // switch over the index //
         switch Index
         {
            // append black to the hues //
            case 0 : Hues.append(NSColor.black)

            // append light gray to the hues //
            case 40 : Hues.append(NSColor.lightGray)

            // append white to the hues //
            case 80 : Hues.append(NSColor.white)

            // the default case //
            default : break
         }

         // append a new hue //
         Hues.append(NSColor(hue:CGFloat(Index % 40) / 39, saturation:0.75, brightness:
                     CGFloat(Index) / 160 + 0.25, alpha:1))
      }

      // create the hue layer //
      HueLayer = Graphics.CreateLayer(2 * frame.size.width, Height:2 * frame.size.height)

      // set the current hue, hue bar width, and set current graphics context to the hue layer //
      (CurrentHue, Width, NSGraphicsContext.current) = (ChosenHue, frame.size.width / CGFloat(Hues
      .count), NSGraphicsContext(cgContext:HueLayer.context!, flipped:false))

      // loop over all the hues //
      for Index in 0..<Hues.count
      {
         // set the current fill hue //
         Hues[Index].setFill()

         // draw the current hue rectangle //
         NSBezierPath(rect:CGRect(x:2 * Width * CGFloat(Index), y:20, width:2 * Width, height:40))
                      .fill()
      }

      // set the border color //
      NSColor.white.set()

      // draw the hue layer border //
      NSBezierPath(rect:CGRect(x:1, y:20, width:2 * (frame.size.width - 1), height:40)).stroke()

      // set that view needs display //
      display(bounds)
   }
}

// the game attributes class //
class GameAttributes:NSObject
{
   // the pgn tags controls //
   @IBOutlet var PGNTags:PGNTagsController!

   // the user preferences to store and retrieve from //
   var Prefs:UserPreferences = UserPreferences()

   // standard fen string components //
   let FENComponents:[String] = ["w", "KQkq", "-", "0", "1"]

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

   // the information dictionary //
   var Inf:Information = Information()

   // the move offset //
   func MoveOffset() -> Int {return BlackStarts() ? 1 : 0}

   // save the game subset with the specified end value //
   func SaveSubset(_ End:Int)
   {
      // backup the values for the specified tags //
      Inf.BackupValues(["FENPref", "TagsPref", "ListsPref"], Tag:"Moves")

      // save the game subset up the the specified end //
      Inf.SaveSubset(End)
   }

   // check to see if there are no moves left in the list //
   func NoMoves() -> Bool {return Inf.GetCount() == 0}

   // compress the moves //
   func Compress(_ Type:String)
   {
      // compress moves if is for the main moves or for the analysis moves //
      if Type == "Moves" {Inf.Compress("Moves")} else {Inf.CompressAnalysis("Moves")}
   }

   // check if the current fen layout is the default layout and currently white to move //
   func IsDefaultFENAndWhite() -> Bool
   {
      // return if the current fen layout is the default layout and currently white to move //
      return Inf.VSIsDefault("FENLayout") && Inf.VSIsDefault("FENColor")
   }

   // save the fen string //
   func SaveFENString(_ FENStringArray:[String])
   {
      // if the fen array is not of size six, set to fen defaults and return //
      if FENStringArray.count != 6 {SetFENDefaults(); return}

      // save the fen items into the info dictionary //
      Inf.Update("FENPref", VSItems:["FENLayout":FENStringArray[0], "FENColor":FENStringArray[1]
                 .lowercased(), "FENCastlingRights":FENStringArray[2], "FENEnpassantSquare":
                 FENStringArray[3].lowercased(), "FENStartMoveNumber":FENStringArray[5]])
   }

   // set the fen defaults //
   func SetFENDefaults() {Inf.VSToDefaults(["FENPref"])}

   // check the color to move //
   func CheckColorToMove()
   {
      // if the fen color is not white or black //
      if !Inf.VSIsEqual("FENColor", VS:"w") && !Inf.VSIsEqual("FENColor", VS:"b")
      {
         // update the fen color to be white //
         Inf.Update("FENPref", VSItems:["FENColor":"w"])
      }
   }

   // get the fen start move number //
   func FENStart() -> Int {return Inf.VSToNum("FENStartMoveNumber")}

   // check the fen start move number //
   func CheckFENStartMoveNumber()
   {
      // check to make sure the fen start move number is bounded //
      Inf.Update("FENPref", VSItems:["FENStartMoveNumber":String(min(max(Inf
                 .VSToNum("FENStartMoveNumber"), 1), 1000))])
   }

   // check if the fen string has white moving first //
   func WhiteStarts() -> Bool {return Inf.VSIsEqual("FENColor", VS:"w")}

   // check if the fen string has black moving first //
   func BlackStarts() -> Bool {return Inf.VSIsEqual("FENColor", VS:"b")}

   // editing the current position //
   func EditingCurrentPosition(_ Board:UnsafeMutableRawPointer)
   {
      // save fen components //
      Inf.Update("FENPref", VSItems:["FENEnpassantSquare":"-", "FENCastlingRights":"-", "FENColor":
                 BoardIsBlack(Board) ? "b" : "w"])
   }

   // swap the color to move //
   func SwapColorToMove()
   {
      // save the fen color into the info dictionary //
      Inf.Update("FENPref", VSItems:["FENColor":(Inf.VSIsEqual("FENColor", VS:"w") ? "b" : "w")])
   }

   // extract the entered fen string //
   func ExtractEnteredFENString(_ EnteredFENString:String)
   {
      // parse the entered fen string to convert //
      EnteredFENStringArray = (EnteredFENString.components(separatedBy:" ")).filter {$0 != ""}

      // loop over all the standard fen components //
      for Index in 0..<FENComponents.count {if EnteredFENStringArray.count == (Index + 1)
      {
         // append the corresponding fen string component to the entered fen string array //
         EnteredFENStringArray.append(FENComponents[Index])
      }}

      // save the entered fen string //
      SaveFENString(EnteredFENStringArray)
   }

   // process the game data //
   func ProcessGameData(_ GameData:UnsafeMutableRawPointer)
   {
      // save the pgn tags from the game data //
      Inf.Update("TagsPref", VSItems:["WhiteName":DStng(DataWhitePlayer(GameData)), "BlackName":
          DStng(DataBlackPlayer(GameData)), "TimeControl":DStng(DataGetTimeControl(GameData)),
          "Opening":DStng(DataGetOpening(GameData)), "Section":DStng(DataGetSection(GameData)),
          "BoardNumber":DStng(DataGetBoardNumber(GameData)), "WhiteElo":
          DStng(DataGetWhiteElo(GameData)), "BlackElo":DStng(DataGetWhiteElo(GameData)),
          "WhiteUSCF":DStng(DataGetWhiteUSCF(GameData)), "BlackUSCF":
          DStng(DataGetBlackUSCF(GameData)), "Annotator":DStng(DataGetAnnotator(GameData)), "ECO":
          DStng(DataGetECO(GameData)), "SiteLocation":DStng(DataGetSite(GameData)), "EventTitle":
          DStng(DataGetEvent(GameData)), "RoundNumber":DStng(DataGetRound(GameData)), "EndResult":
          DStng(DataGetResult(GameData)),"DateGamePlayed":DStng(DataGetDate(GameData))])

      // save the parsed fen string //
      SaveFENString(String(validatingUTF8:DataGetFENString(GameData))!.components(separatedBy:" "))

      // parse the moves list string into the information dictionary //
      Inf.Parse("Moves", VSs:String(validatingUTF8:DataGetGameDataUCI(GameData))!, VLSs:
               String(validatingUTF8:DataGetGameDataAnnotations(GameData))!, VSSeparator:" ",
               VLSSeparator:"*****")
   }

   // process the analysis string //
   func ProcessAnalysis(_ AnalysisString:String)
   {
      // parse the analysis string into the information dictionary //
      Inf.Parse("Moves", VSs:AnalysisString, Separator:" ")
   }

   // get the fen string //
   func FENString() -> String
   {
      // return the fen string //
      return Inf.Vals("FENLayout").VS + " " + Inf.Vals("FENColor").VS + " " + Inf
             .Vals("FENCastlingRights").VS + " " + Inf.Vals("FENEnpassantSquare").VS + " 0 " +
             Inf.Vals("FENStartMoveNumber").VS
   }

   // store the game header data in the game data object //
   func StoreGameHeaderData(_ GameData:UnsafeMutableRawPointer)
   {
      // store the game header data in the game data oject //
      DataStoreGameHeader(GameData, Inf.Vals("WhiteName").VS, Inf.Vals("BlackName").VS, Inf
                          .Vals("SiteLocation").VS, Inf.Vals("EventTitle").VS, Inf
                          .Vals("RoundNumber").VS, Inf.Vals("EndResult").VS, Inf
                          .Vals("DateGamePlayed").VS, Inf.Vals("TimeControl").VS, Inf
                          .Vals("WhiteElo").VS, Inf.Vals("BlackElo").VS, Inf.Vals("WhiteUSCF").VS,
                          Inf.Vals("BlackUSCF").VS, Inf.Vals("Annotator").VS, Inf.Vals("ECO").VS,
                          Inf.Vals("Opening").VS, Inf.Vals("Section").VS, Inf.Vals("BoardNumber").VS,
                          FENString())
   }

   // convert the extracted game data information to a string //
   func DStng(_ Convert:UnsafePointer<CChar>) -> String
   {
      // return the extracted game data information as a string //
      return String(validatingUTF8:Convert) != nil ? String(validatingUTF8:Convert)! : ""
   }

   // get the pgn string from the text field //
   func PGNStr(_ TextField:NSTextField) -> String
   {
      // return the pgn string from the text field //
      return GetString().FromTextField(TextField.stringValue)
   }

   // update the pgn tags from the enter game info window //
   func UpdateFromEnterGameInfoWindow()
   {
      // save the pgn tags from the pgn tag text fields //
      Inf.Update("TagsPref", VSItems:["WhiteName":PGNStr(PGNTags.WhiteNameTextField), "BlackName":
          PGNStr(PGNTags.BlackNameTextField), "SiteLocation":PGNStr(PGNTags.SiteLocationTextField),
          "EventTitle":PGNStr(PGNTags.EventTitleTextField), "RoundNumber":PGNStr(PGNTags
          .RoundNumberTextField), "TimeControl":PGNStr(PGNTags.TimeControlTextField), "Opening":
          PGNStr(PGNTags.OpeningTextField), "Section":PGNStr(PGNTags.SectionTextField),
          "BoardNumber":PGNStr(PGNTags.BoardNumberTextField), "WhiteElo":PGNStr(PGNTags
          .WhiteEloTextField), "BlackElo":PGNStr(PGNTags.BlackEloTextField), "WhiteUSCF":
          PGNStr(PGNTags.WhiteUSCFTextField), "BlackUSCF":PGNStr(PGNTags.BlackUSCFTextField),
          "Annotator":PGNStr(PGNTags.AnnotatorTextField), "ECO":PGNStr(PGNTags.ECOTextField),
          "DateGamePlayed":PGNTags.GetDateGamePlayed()])

      // clear the end result value //
      Inf.ClearVS("TagsPref", Key:"EndResult")

      // if white won the game //
      if PGNTags.Inf.Vals("WhiteWonGame").VB {Inf.Update("TagsPref", VSItems:["EndResult":"1-0"])}

      // if black won the game //
      if PGNTags.Inf.Vals("BlackWonGame").VB {Inf.Update("TagsPref", VSItems:["EndResult":"0-1"])}

      // if the game was drawn //
      if PGNTags.Inf.Vals("DrawnGame").VB {Inf.Update("TagsPref", VSItems:["EndResult":"1/2-1/2"])}
   }

   // get the pgn game file header tag for the specified key //
   func GetTag(_ Key:String) -> String
   {
      // return the pgn game file header tag for the specified key //
      return "[" + Inf.Vals(Key).VLS + " \"" + Inf.Vals(Key).VS + "\"]\n"
   }

   // get the pgn game file header string //
   func GetPGNGameHeader() -> String
   {
      // get game header information for pgn //
      return GetTag("EventTitle") + GetTag("SiteLocation") + GetTag("DateGamePlayed") +
             GetTag("RoundNumber") + GetTag("WhiteName") + GetTag("BlackName") + GetTag("EndResult")
             + GetTag("TimeControl") + GetTag("WhiteElo") + GetTag("BlackElo") + GetTag("WhiteUSCF")
             + GetTag("BlackUSCF") + GetTag("Annotator") + GetTag("ECO") + GetTag("Opening") +
             GetTag("Section") + GetTag("BoardNumber") + "[SetUp \"1\"]\n" + "[FEN \"" + FENString()
             + "\"]\n\n"
   }

   // get the game information list //
   func GetInfoList() -> [String]
   {
      // return the game information list from the pgn tags //
      return AppTag("EventTitle") + AppTag("SiteLocation") + AppTag("DateGamePlayed") +
             AppTag("RoundNumber") + AppTag("WhiteName") + AppTag("BlackName") + AppTag("EndResult")
             + AppTag("TimeControl") + AppTag("WhiteElo") + AppTag("BlackElo") + AppTag("WhiteUSCF")
             + AppTag("BlackUSCF") + AppTag("Annotator") + AppTag("ECO") + AppTag("Opening") +
             AppTag("Section") + AppTag("BoardNumber")
   }

   // append a tag to the game info list //
   func AppTag(_ Key:String) -> [String]
   {
      // return the label and value or an empty array //
      return !Inf.VSIsEmpty(Key) ? [Inf.Vals(Key).VLS + ":  " + Inf.Vals(Key).VS] : [""]
   }

   // initialize to empty attributes //
   func Initialize()
   {
      // set the default fen strings //
      Inf.VSToDefaults(["FENPref"])

      // clear the pgn tags and the lists //
      Inf.ClearVS(["TagsPref", "ListsPref"])

      // clear all the indices //
      Inf.ClearAllKeyIndices()

      // save the current date to use if no other date is available //
      Inf.Update("TagsPref", VSItems:["DateGamePlayed":PGNTags.GetCurrentDate()])
   }

   // override the initializer //
   override init()
   {
      // add the FEN information to the information dictionary //
      Inf.Update("FENPref", VDSItems:["FENLayout":"rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR",
                "FENColor":"w", "FENCastlingRights":"KQkq", "FENEnpassantSquare":"-",
                "FENStartMoveNumber":"1"])

      // add the pgn tag information to the information dictionary //
      Inf.Update("TagsPref", VSItems:["WhiteName":"White", "BlackName":"Black", "SiteLocation":
```

```
   Inf.Update("TagsPref", VSItems:["WhiteName":"White", "BlackName":"Black", "SiteLocation":
              "Site", "EventTitle":"Event", "RoundNumber":"Round", "EndResult":"Result",
              "DateGamePlayed":"Date", "TimeControl":"TimeControl", "Opening":"Opening",
              "Section":"Section", "BoardNumber":"Board", "WhiteElo":"WhiteElo", "BlackElo":
              "BlackElo", "WhiteUSCF":"WhiteUSCF", "BlackUSCF":"BlackUSCF", "Annotator":
              "Annotator", "ECO":"ECO"])

   // add the move list string and the annotation list string to the information dictionary //
   Inf.Update("ListsPref", VSItems:["MoveListString":"", "AnnotationListString":""])
}

// initialize the game attributes for the given tag //
func Initialize(_ Suffix:String) -> Bool
{
   // get all the preferences //
   Inf.GetPrefs(["FENPref", "TagsPref", "ListsPref"], Suffix:Suffix)

   // parse the move list string and the annotation list string into info items in dictionary //
   Inf.Parse("Moves", VSKey:"MoveListString", VLSKey:"AnnotationListString", VSSeparator:
            "**********", VLSSeparator:"**********")

   // return if the move list string is empty and is the default fen layout
   return Inf.VSIsEmpty("MoveListString") && Inf.VSIsDefault("FENLayout")
}

// store the autosave data //
func StoreAutosaveData(_ Suffix:String)
{
   // concatenate the move and annotations into lists to store into the info dictionary //
   Inf.ConcatenateLists("ListsPref", VSKey:"MoveListString", VLSKey:"AnnotationListString")

   // save all the preferences for the specified tags //
   Inf.SavePrefs(["FENPref", "TagsPref", "ListsPref"], Suffix:Suffix)
   }
}

// the core graphics class //
class CoreGraphics
{
   // a rectangle //
   var Rectangle:CGRect = CGRect()

   // the base address for use in pixel buffers //
   var BaseAddress:UnsafePointer<UInt8>? = nil

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

   // the context //
   var Context:CGContext!

   // convert an image to a pixel buffer //
   func ImageToBuffer(_ Image:CGImage) -> CVPixelBuffer?
   {
      // get the base address of the cgimage data and reset the pixel buffer //
      (BaseAddress, PixelBuffer) = (CFDataGetBytePtr((Image.dataProvider?.data)!), nil)

      // draw the image into the cv pixel buffer //
      CVPixelBufferCreateWithBytes(nil, Image.width, Image.height, kCVPixelFormatType_32ARGB,
                                   UnsafeMutableRawPointer(mutating:BaseAddress!), Image.bytesPerRow,
                                   {ReleaseContext, BaseAddress in let ContextData =
                                   Unmanaged<CFData>.fromOpaque(ReleaseContext!); ContextData
                                   .release()}, Unmanaged<CFData>.passRetained((Image.dataProvider?
                                   .data)!).toOpaque(), nil, &PixelBuffer)

      // return the pixel buffer //
      return PixelBuffer
   }

   // create the context //
   func CreateContext(_ Width:CGFloat, Height:CGFloat) -> CGContext
   {
      // return a context for the given width and height //
      return CGContext(data:nil, width:Int(Width), height:Int(Height), bitsPerComponent:8,
                       bytesPerRow:0, space:CGColorSpaceCreateDeviceRGB(), bitmapInfo:
                       CGImageAlphaInfo.premultipliedFirst.rawValue)!
   }

   // create the layer //
   func CreateLayer(_ Width:CGFloat, Height:CGFloat) -> CGLayer
   {
      // return a layer for the given width and height //
      return CGLayer(CreateContext(Width, Height:Height), size:CGSize(width:Width, height:Height),
                     auxiliaryInfo:nil)!
   }

   // get a cgimage from an nsimage //
   func NSImageToCGImage(_ Image:NSImage, Size:CGFloat) -> CGImage
   {
      // create the context for the given image size //
      Context = CreateContext(Size, Height:Size)

      // draw the image into the context //
      Context.draw(Image.cgImage(forProposedRect:nil, context:nil, hints:nil)!, in:CGRect(x:0, y:0,
                   width:Context.width, height:Context.height), byTiling:false)

      // return the cgimage //
      return Context.makeImage()!
   }

   // create a square or circle cgimage //
   func CreateShape(_ Width:CGFloat, Fraction:CGFloat, Color:NSColor, Square:Bool) -> CGImage
   {
      // create the context from the given width //
      Context = CreateContext(Width, Height:Width)

      // translate the context origin to the center of the context //
      Context.translateBy(x:Width * 0.5, y:Width * 0.5)

      // set the fill color for the square //
      Context.setFillColor(Color.cgColor)

      // the rectangle for the square or circle //
      Rectangle = CGRect(x:-Width * 0.5 * Fraction, y:-Width * 0.5 * Fraction, width:Width *
                         Fraction, height:Width * Fraction)

      // if is for a square shape //
      if Square {Context.fill(Rectangle)} else {Context.fillEllipse(in:Rectangle)}

      // return a cgimage from the context //
      return Context.makeImage()!
   }

   // convert a cgimage to an nsimage for the specified size //
   func CGImageToNSImage(_ Image:CGImage, Size:CGFloat) -> NSImage
   {
      // return an nsimage from a cgimage from the given image and size //
      return NSImage(cgImage:Image, size:CGSize(width:Size, height:Size))
   }
}
```