```
//////////////////////////////////////////////////////////////////////
// Swift source code of main.swift Version 9.1.9 at greenchess.com //
//////////////////////////////////////////////////////////////////////

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

// the public user interface //
let GCUI:GCUserInterface = GCUserInterface()

// the greenchess implementation //
class GreenChess:NSView, @preconcurrency CAAnimationDelegate, NSWindowDelegate,
              NSApplicationDelegate, NSMenuItemValidation
{
   // the vertical y offset of the board view //
   var YOffset:Double = 105

   // the game database explorer slider value //
   var DataGameExpVal:Double = -1

   // the move rows list //
   var MoveRowsList:[AListRow] = []

   // the info rows list //
   var InfoRowsList:[AListRow] = []

   // the previous search string //
   var PrevSearch:String = ""

   // the game list data //
   var GameListData:[String] = []

   // the grid rows list //
   var GridRows:[String] = ["Wraparound 2 rows of boards in Grid View",
                            "Wraparound 3 rows of boards in Grid View"]

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

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

   // the main view image //
   var MainViewImage:CGImage = CoreGraphics().InitImage()

   // the sketch being drawn to //
   var Sketch:Sketching = Sketching()

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

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

   // the stockfish engine //
   nonisolated(unsafe) var Stockfish:StockfishEngine = StockfishEngine()

   // the game over dictionary //
   var GameOver:[Int:String] = [1:"Checkmate and Black Wins",
                                2:"Checkmate and White Wins", 3:"Stalemate",
                                4:"Draw by Threefold Repetition",
                                5:"Black Wins on Time", 6:"White Wins on Time",
                                7:"Draw by Insufficient Mating Material"]

   // the board and pieces data //
   var BPData:ChessBoardData = ChessBoardData()

   // the progress circle //
   var ProgressCircle:ProgressCircleLayer = ProgressCircleLayer()

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

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

   // the menu item states //
   var MenuItemStates:TheMenuItems = TheMenuItems()

   // the menu items dictionary to use //
   var MenuItems:Int = 0

   // the gallery review timer //
   var GalleryReviewTimer:Timer = Timer()

   // the game explorer timer //
   var GameExplorerTimer:Timer = Timer()

   // the stockfish engine timer //
   var StockfishTimer:Timer = Timer()

   // the snapshot image representation //
   var ImageBitmap:NSBitmapImageRep!

   // the gallery grid approximate board sizes //
   var GridSizes:[CGFloat] = [220, 220, 160]

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

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

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

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

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

   // the move list font //
   var MoveListFont:FontsSet = FontsSet()

   // the game info font //
   var GameInfoFont:FontsSet = FontsSet()

   // the game list font //
   var GameListFont:FontsSet = FontsSet()

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

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

   // the timed game color index //
   nonisolated(unsafe) var TimedGameColor:Int = 0

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

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

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

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

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

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

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

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

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

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

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

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

   // the saved position images //
   var PosImages:[Image] = []

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

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

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

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

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

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

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

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

   // the remaining time for white //
   nonisolated(unsafe) var RemainingWhiteTime:Int = 0

   // the remaining time for black //
   nonisolated(unsafe) var RemainingBlackTime:Int = 0

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

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

   // the total time for the timed game //
   nonisolated(unsafe) var TotalGameTime:Int = 0

   // the time left in the timed game //
   nonisolated(unsafe) var TimeLeft:Int = 0

   // the moving piece layer //
   let MovingPiece:AMovingPiece = AMovingPiece()

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

   // the message icon image //
   var MessageIcon:NSImage = CoreGraphics()
                             .CGImageToNSImage(ImageRenderer(content:
                             Image("message.fill", .blue)).cgImage!, Size:
                             500)

   // the validated icon image //
   var ValidatedIcon:NSImage = CoreGraphics()
                               .CGImageToNSImage(ImageRenderer(content:
                               Image("checkmark.message.fill", .green))
                               .cgImage!, Size:500)

   // the warning icon image //
   var WarningIcon:NSImage = CoreGraphics()
                             .CGImageToNSImage(ImageRenderer(content:
                             Image("exclamationmark.message.fill", .red))
                             .cgImage!, Size:500))

   // the string symbol that the pawn promoted to //
   var PromotionSymbol:String = ""

   // a move annotation string //
   var AnnotationString:String = ""

   // the return code from an alert //
   var ReturnCode:Int = 0

   // the panel for saving the game to the chosen filename and directory //
   var SavePanel:NSSavePanel = NSSavePanel()

   // the panel for opening a pgn game file //
   var OpenPanel:NSOpenPanel = NSOpenPanel()

   // the panel for appending a pgn game file //
   var AppendPanel:NSOpenPanel = NSOpenPanel()

   // a list of pgn moves for the pgn moves list data source //
   var PGNMoveList:[String] = []

   // the game info list //
   var GameInfoList:[String] = []

   // the stockfish principal variation line of play //
   nonisolated(unsafe) var StockfishPV:String = ""

   // the saved position data array //
   var SavedPositionData:[UnsafeMutableRawPointer] = []

   // a pgn move string for the pgn moves list data source //
   var PGNMove:String = ""

   // a pgn label string //
   var PGNLabel:String = ""

   // the selected fen piece from the fen layout for the board //
   var FENPiece:String = ""

   // the fen string //
   var FENString:String = ""

   // the load position window //
   var LoadPositionWindow:AWindow = AWindow()

   // the game gallery window //
   var GameGalleryWindow:AWindow = AWindow()

   // the enter fen window //
   var FENWindow:AWindow = AWindow()

   // the enter game info window //
   var EnterInfoWindow:AWindow = AWindow()

   // the timed game window //
   var TimedGameWindow:AWindow = AWindow()

   // the enter preferences window //
   var PreferencesWindow:AWindow = AWindow()

   // the main gui window //
   @IBOutlet var MainWindow:AWindow!

   // the about window //
   var AboutWindow:AWindow = AWindow()

   // the game list window //
   var GameListWindow:AWindow = AWindow()

   // the game explorer window //
   var GameExplorerWindow:AWindow = AWindow()

   // the game annotator window //
   var GameAnnotatorWindow:AWindow = AWindow()

   // the database explorer window //
   var DatabaseExplorerWindow:AWindow = AWindow()

   // the hide application menu item //
   @IBOutlet var HideMenuItem:NSMenuItem!

   // the hide info panel menu item //
   @IBOutlet var HideInfoMenuItem:NSMenuItem!

   // the large info panel menu item //
   @IBOutlet var LargeInfoMenuItem:NSMenuItem!

   // the small info panel menu item //
   @IBOutlet var SmallInfoMenuItem:NSMenuItem!

   // the chess engine opponent menu item //
   @IBOutlet var UCIEngineOpponentMenuItem:NSMenuItem!

   // the go stop menu item //
   @IBOutlet var GoStopMenuItem:NSMenuItem!

   // the game explorer menu item //
   @IBOutlet var GameExplorerMenuItem:NSMenuItem!

   // the timed game menu item //
   @IBOutlet var TimedGameMenuItem:NSMenuItem!

   // the human opponent menu item //
   @IBOutlet var HumanOpponentMenuItem:NSMenuItem!

   // the engine opponent menu item //
   @IBOutlet var EngineOpponentMenuItem:NSMenuItem!

   // the possible fen board layout characters //
   var FENCharacters:[String] = ["k", "q", "r", "b", "n", "p", "", "P", "N",
                                 "B", "R", "Q", "K"]

   // the piece dictionary //
   var PieceDict:[String:Int] = ["p":-1, "n":-2, "b":-3, "r":-4, "q":-5, "k":-6,
                                 "P":1, "N":2, "B":3, "R":4, "Q":5, "K":6]

   // the white piece symbol dictionary //
   var WhiteDict:[String:String] = ["K":"â™”", "Q":"â™•", "R":"â™–", "B":"â™—", "N":"â™˜"]

   // the black piece symbol dictionary //
   var BlackDict:[String:String] = ["K":"â™š", "Q":"â™›", "R":"â™œ", "B":"â™", "N":"â™ž"]

   // the annotation glyph codes //
   var Glyphs:[String] = ["$133", "$132", "$41", "$40", "$37", "$36", "$33",
                          "$32", "$23", "$22", "$19", "$18", "$17", "$16", "$15",
                          "$14", "$13", "$10", "$7", "$6", "$5", "$4", "$3",
                          "$2", "$1"]

   // the annotation glyph symbols //
   var GlyphSyms:[String] = ["â‡†", "â‡†", "â†‘", "â†‘", "â†’", "â†’", "âŸ³", "âŸ³", "â¨€", "â¨€",
                             "âˆ’ +", "+ âˆ’", "âˆ“", "Â±", "â©±", "â©²", "âˆž", "=", "â–¡",
                             "â‰", "â‰", "â‡", "â€¼", "?", "!"]

   // the about dictionary //
   var AboutDict:[Int:String] = [1:"greenchess.com/contact", 2:"greenchess.com",
                                 3:"www.gnu.org/licenses/gpl-3.0.en.html", 4:
                                 "greenchess.com/forum", 5:
                                 "greenchess.com/privacy", 6:
                                 "stockfishchess.org/download"]

   // the position pieces dictionary //
   var PosPieces:[Int:Int] = [2:0, 3:1, 4:2, 5:3, 6:4, 7:5, 8:6, 9:-1, 10:-2,
                             11:-3, 12:-4, 13:-5, 14:-6]

   // the enpassant squares label dictionary //
   var EnpassantSquares:[Int:String] = [1:"a3", 2:"b3", 3:"c3", 4:"d3", 5:"e3",
                                        6:"f3", 7:"g3", 8:"h3", 9:"a6", 10:"b6",
                                        11:"c6", 12:"d6", 13:"e6", 14:"f6", 15:
                                        "g6", 16:"h6"]

   // the enpassant squares selection dictionary //
   var EnpassantSelection:[String:Int] = ["-":0, "a3":1, "b3":2, "c3":3, "d3":4,
                                          "e3":5, "f3":6, "g3":7, "h3":8, "a6":9,
                                          "b6":10, "c6":11, "d6":12, "e6":13,
                                          "f6":14, "g6":15, "h6":16]

   // the application did finish launching //
   @MainActor func applicationDidFinishLaunching(_ Notif:Notification)
   {
      // refresh the table views //
      GCUI.RefreshViews(["MoveListTableView":false, "GameInfoTableView":false])
   }

   // initialize the app before finishes launching //
   @MainActor func applicationWillFinishLaunching(_ Notif:Notification)
   {
      // make the main window transparent to hide it //
      MainWindow.alphaValue = 0

      // initialize the stockfish chess engine //
      Stockfish.Initialize()

      // update opponent title to the current version of stockfish //
      UCIEngineOpponentMenuItem.title = String(Stockfish.GetEngineName()
                                               .prefix(12))

      // initialize the user interface //
      InitializeUI()

      // start a new chess game //
      StartNewGame("All")

      // refresh the table views //
      RefreshViews(["MoveListTableView":false, "GameInfoTableView":false])

      // if chose to autosave game data, extract the autosave game data //
      if Inf.Vals("AutosaveGame").VB {ExtractGameData(GameData, Tasks:"All")}

      // initialize the game clock //
      InitializeGameClock()

      // create a game timer that gets called ten times per second //
      RunLoop.current.add(Timer(timeInterval:0.1, target:self, selector:
                    #selector(UpdateGameClock), userInfo:nil, repeats:
                    true), forMode:RunLoop.Mode.common)

      // add a timer to remove extra edit menu items //
      RunLoop.current.add(Timer(timeInterval:0.5, target:self, selector:
                    #selector(RemoveEditItems), userInfo:nil, repeats:
                    false), forMode:RunLoop.Mode.common)

      // read the opening books on a background task //
      Task.detached(priority:.background) {self.ReadOpeningBooks()}

      // refresh the table views //
      RefreshViews(["MoveListTableView":false, "GameInfoTableView":false])
   }

   // override isopaque //
   @MainActor override var isOpaque:Bool {return true}

   // override accepts first responder //
   @MainActor override var acceptsFirstResponder:Bool {return true}

   // set the window background color //
   @MainActor func SetWindowBackgroundColor(_ RefreshBoard:Bool)
   {
      // get the current window background color //
      Inf.Update("VIPref", VBItems:["WindowBackgroundHue":GCUI
              .GetColorView("WindowColorView").GetCurrentHue()])

      // set the main window background color to the window texture //
      MainWindow.backgroundColor = WindowTexture.GetTexture(GCUI
                                    .GetColorView("WindowColorView")
                                    .GetCurrentHue(), Opacity:Inf
                                    .Vals("WindowTextureOpacity").VI)

      // save the window color to the board and piece compositor //
      GCUI.BPComp.WindowColor = MainWindow.backgroundColor

      // update the preference preview //
      UpdatePreview(RefreshBoard, Board:false, Diagram:false, Texture:true)

      // update the white time clock window color //
      GCUI.GetClockView("WhiteClockView").UpdateClockWindowColor(GCUI
           .GetColorView("WindowColorView").GetCurrentHue())

      // update the black time clock window color //
      GCUI.GetClockView("BlackClockView").UpdateClockWindowColor(GCUI
           .GetColorView("WindowColorView").GetCurrentHue())

      // update the time clock window preview color //
      GCUI.GetClockView("ClockView").UpdateClockWindowColor(GCUI
           .GetColorView("WindowColorView").GetCurrentHue())
   }

   // get the window sheet attachment point //
   @MainActor func window(_ Window:NSWindow, willPositionSheet Sheet:NSWindow,
                          using Rect:CGRect) -> CGRect
   {
      // set that the attached window sheet isn't movable //
      Sheet.isMovable = false

      // return the window sheet attachment point //
      return CGRect(x:0, y:740, width:Window.frame.size.width, height:0)
   }

   // the window did exit full screen //
   @MainActor func windowDidExitFullScreen(_ Notif:Notification)
   {
      // set that no longer in full screen mode //
      Inf.TurnOffVB("VB", Key:"FullScreen")

      // update the main window //
      UpdateMainWindow()
   }

   // the window will enter full screen mode //
   @MainActor func windowWillEnterFullScreen(_ Notif:Notification)
   {
      // set the info panel size to large //
      Resize(false, Small:false, Large:true)

      // set that in full screen mode //
      Inf.TurnOnVB("VB", Key:"FullScreen")
   }

   // the window did enter fullscreen //
   @MainActor func windowDidEnterFullScreen(_ Notif:Notification)
   {
      // update the main window //
      UpdateMainWindow()
   }

   // the window did miniaturize //
   @MainActor func windowDidMiniaturize(_ Notif:Notification)
   {
      // update the window open status //
      UpdateWindowOpenStatus(false)
   }

   // get the snapshot image data //
   @MainActor func SnapshotData() -> Data
   {
      // get the image bitmap //
      ImageBitmap = NSBitmapImageRep(cgImage:Inf.Vals("Diagramming").VB ?
                    Sketch.Snapshot() : BoardImage(Inf
                    .Obj("GameExplorerWindowOpen").VB ? Int(round(GCUI
                    .Obj("GameExpVal").VI * Double(GCUI.Attributes.Inf
                    .GetCount()))) : (Inf.Vals("InDatabaseExplorer").VB ?
                    Int(round(GCUI.Obj("DataExpMove").VI * Double(GCUI
                    .Attributes.Inf.GetCount()))) : -1), Size:1320, Update:
                    false))

      // set the image bitmap size //
      ImageBitmap.size = bounds.size

      // return the image bitmap as data //
      return ImageBitmap.representation(using:NSBitmapImageRep.FileType.jpeg,
                                        properties:[:])!
   }

   // copy the current board image into the pasteboard //
   @MainActor @IBAction func CopyBoardToPasteboardAction(_:AnyObject)
   {
      // clear the pasteboard contents //
      Pasteboard.clearContents()

      // set the pasteboard contents to the current snapshot data //
      Pasteboard.setData(SnapshotData(), forType:NSPasteboard.PasteboardType
                         .png)
   }

   // copy the current game to the pasteboard //
   @MainActor @IBAction func CopyGameToPasteboardAction(_:AnyObject)
   {
      // clear the pasteboard contents //
      Pasteboard.clearContents()

      // set the pasteboard contents to the current game pgn //
      Pasteboard.setString(GamePGN(), forType:NSPasteboard.PasteboardType
                           .string)
   }

   // the about window selection //
   @MainActor func AboutAction(_ Selection:Int)
   {
      // if the exit window button was clicked, close the window and reopen //
      if Selection == 0 {CloseSheet(AboutWindow); EndSheet(); return}

      // open the corresponding webpage //
      NSWorkspace.shared.open(URL(string:"https://\(AboutDict[Selection]!)")!)
   }

   // if the mouse entered the tracking area //
   @MainActor override func mouseEntered(with Event:NSEvent) {UpdateCursor()}

   // if the mouse exited the tracking area //
   @MainActor override func mouseExited(with Event:NSEvent) {UpdateCursor()}

   // update the window open status //
   @MainActor func UpdateWindowOpenStatus(_ Open:Bool)
   {
      // if the window is not currently open //
      if !Open
      {
         // set the hide menu item title to reopen //
         HideMenuItem.title = "Reopen"

         // set that the main window is currently not hidden //
         Inf.TurnOnVB("VB", Key:"MainWindowHidden")

         // exit the function //
         return
      }

      // make the app active //
      NSApp.activate(ignoringOtherApps:true)

      // make the main window visible //
      MainWindow.makeKeyAndOrderFront(nil)

      // set the hide menu item title to hide //
      HideMenuItem.title = "Hide"

      // set that the main window is not hidden //
      Inf.TurnOffVB("VB", Key:"MainWindowHidden")
   }

   // the application did become active //
   @MainActor func applicationDidBecomeActive(_ TheNotification:Notification)
   {
      // run on the main thread //
      Task {await MainActor.run
      {
         // update the window open status //
         UpdateWindowOpenStatus(true)

         // if in the game gallery view //
         if Inf.Vals("InGameGallery").VB
         {
            // refresh the game gallery cursor //
            GCUI.GetGalleryView("GalleryView").RefreshCursor()

            // refresh the game gallery slider //
            RefreshViews(["GallerySlider":false])
         }
         // not in the game gallery view //
         else
         {
            // update the cursor //
            UpdateCursor()
         }
      }}
   }

   // handle application being clicked on in dock //
   @MainActor func applicationShouldHandleReopen(_ Application:NSApplication,
                                                 hasVisibleWindows Flag:Bool) ->
                                                 Bool
   {
      // update the window open status //
      UpdateWindowOpenStatus(true)

      // return that main window was made visible //
      return true
   }

   // if is a valid file extension //
   @MainActor func ValidPGNFile(_ Filename:String) -> Bool
   {
      // turn on that doing query for invalid file //
      Inf.TurnOnVB("VB", Key:"QueryInvalidPGNFile")

      // return false if the filename doesn't have a document extension type //
      if ((Filename.components(separatedBy:".")).filter {$0 != ""}).count < 2
      {
         // alert the user if invalid file type //
         MakeAlert(["Okay"], Query:"Invalid File Type")

         // return that not a valid file extension //
         return false
      }

      // return false if not the specified file extension type //
      if (((Filename.components(separatedBy:".")).filter {$0 != ""}).last!)
         .lowercased() != "pgn"
      {
         // alert the user if invalid file type //
         MakeAlert(["Okay"], Query:"Invalid File Type")

         // return false //
         return false
      }

      // turn off that currently doing query for invalid file if file valid //
      Inf.TurnOffVB("VB", Key:"QueryInvalidPGNFile")

      // return that the file is valid //
      return true
   }

   // delegate to handle user clicking on pgn file to open application //
   @MainActor func application(_ Application:NSApplication, openFile Filename:
                                String) -> Bool
   {
      // update the window open status //
      UpdateWindowOpenStatus(true)

      // clear the selected square if needed //
      ClearSelectedSquareIfNeeded()

      // if don't want to open file under these conditions //
      if HaveAttachedSheet() || Inf.Vals("InPositionEditor").VB || MovingPiece
         .HaveMovingPiece() || Inf.Vals("EngineThinking").VB || MovingPiece
         .Animating() || Inf.Vals("PlayingEngineMatch").VB || Inf
         .Vals("InPanel").VB || Inf.Vals("InPanel").VB || Inf
         .Vals("ExploringGame").VB
      {
         // exit since don't want to open game file under these conditions //
         return true
      }

      // open the user selected game file //
      OpenGame(Filename)

      // return that done with function //
      return true
   }

   // run the core animation of piece moving on the board //
   @MainActor func RunCoreAnimation()
   {
      // set stop engines to false //
      Inf.TurnOffVB("VB", Key:"StopEngines")

      // if not undoing an illegal move //
      if MovingPiece.IsLegal()
      {
         // make the move on the board //
         MakeMoveOnBoard(MovingPiece.UCIMove)

         // save the pgn notation //
         GCUI.Attributes.Inf.SaveNotation("Moves", VBS:String(cString:
                                          BoardPGNLabel(Board)))

         // if move does not exist //
         if !Inf.Vals("MoveExists").VB
         {
            // reset the moving piece //
            MovingPiece.Reset()

            // refresh all //
            Refresh("All")

            // exit the function //
            return
         }

         // process the uci move //
         ProcessUCIMove()
      }

      // prepare the moving piece to be animated //
      MovingPiece.PrepareAnimation(BPData.Start, End:BPData.End, Rate:Inf
                                   .Vals("AnimationRate").VI)

      // refresh the view //
      Refresh("View")
   }

   // animation did stop delegate //
   @MainActor func animationDidStop(_ TheAnimation:CAAnimation, finished Flag:
                                    Bool)
   {
      // if need to stop the chess engines //
      if Inf.Vals("StopEngines").VB
      {
         // turn off stop engines //
         Inf.TurnOffVB("VB", Key:"StopEngines")

         // exit the function //
         return
      }

      // remove the animation layer from the super layer //
      MovingPiece.HideAnimation()

      // put the piece that moved onto the destination square //
      UpdateDestinationSquare()

      // play the sound of a piece thumping on the board //
      GCUI.GameSounds.Play("sound_thump")

      // if move is illegal, reset to legal and replay the moves //
      if MovingPiece.IsIllegal()
      {
         // reset the illegal move value //
         MovingPiece.ResetIllegal()

         // refresh all //
         Refresh("All")

         // exit the function //
         return
      }

      // update the chess board data //
      UpdateBoardData(GCUI.Attributes.Inf.GetCount())

      // refresh the view and tables //
      Refresh("ViewTables")

      // set that the chess engine is done thinking //
      Inf.TurnOffVB("VB", Key:"EngineThinking")

      // set the go stop menu item title to engine //
      GoStopMenuItem.title = "Engine"

      // if playing an engine match, make an engine match move //
      if Inf.Vals("PlayingEngineMatch").VB {MakeEngineMatchMove()}
   }

   // make an engine match move //
   @MainActor func MakeEngineMatchMove()
   {
      // update if the game is over //
      Inf.Update("VB", VBItems:["PlayingEngineMatch":!BoardGameOver(Board)])

      // if playing an engine match, move an engine move //
      if Inf.Vals("PlayingEngineMatch").VB {StartEngineMove()}
   }

   // override mouse down can move window //
   @MainActor override var mouseDownCanMoveWindow:Bool {return false}

   // the hide info panel menu item action //
   @MainActor @IBAction func HideInfoMenuItemAction(_:AnyObject)
   {
      // resize the window to hide panel //
      Resize(true, Small:false, Large:false)
   }

   // the small info panel menu item action //
   @MainActor @IBAction func SmallInfoMenuItemAction(_:AnyObject)
   {
      // resize the window to small //
      Resize(false, Small:true, Large:false)
   }

   // the large info panel menu item action //
   @MainActor @IBAction func LargeInfoMenuItemAction(_:AnyObject)
   {
      // resize the window to large //
      Resize(false, Small:false, Large:true)
   }

   // update the display game clock toggle button //
   @MainActor func UpdateDisplayGameClock(_ Key:String)
   {
      // toggle the hide game clock in panel button //
      Inf.ToggleVB("VBPref", Key:Key)

      // update the time clock hidden status //
      (GCUI.GetClockView("WhiteClockView").isHidden, GCUI
      .GetClockView("BlackClockView").isHidden) = (Inf
      .Vals("HideGameClockInPanel").VB, Inf.Vals("HideGameClockInPanel").VB)

      // update the main window //
      UpdateMainWindow()
   }

   // the resize the window panel //
   @MainActor func Resize(_ Hide:Bool, Small:Bool, Large:Bool)
   {
      // update the hide or the small or the large panel //
      Inf.Update("VBPref", VBItems:["HideInfoPanel":Hide, "SmallInfoPanel":
                 Small, "LargeInfoPanel":Large])

      // save the boolean preferences //
      Inf.SavePrefs(["VBPref"], VITags:[], VSTags:[])

      // update the main window //
      UpdateMainWindow()
   }

   // the window should close delegate //
   @MainActor func windowShouldClose(_ Window:NSWindow) -> Bool
   {
      // if is the main window, update the window open status //
      if Window == MainWindow {UpdateWindowOpenStatus(false)}

      // return that should close the main window //
      return true
   }

   // the custom windows to use when entering full screen //
   @MainActor func customWindowsToEnterFullScreen(for Window:NSWindow) ->
                                                   [NSWindow]?
   {
      // return the main window //
      return [MainWindow]
   }

   // start the custom animation when entering full screen //
   @MainActor func window(_ Window:NSWindow,
             startCustomAnimationToEnterFullScreenWithDuration Duration:
             TimeInterval) {}

   // the custom windows to use when exit full screen //
   @MainActor func customWindowsToExitFullScreen(for Window:NSWindow) ->
                                                  [NSWindow]?
   {
      // return the main window //
      return [MainWindow]
   }

   // start the custom animation when exiting full screen //
   @MainActor func window(_ Window:NSWindow,
             startCustomAnimationToExitFullScreenWithDuration Duration:
             TimeInterval) {}

   // update the main window //
   @MainActor func UpdateMainWindow()
   {
      // run on the main thread //
      Task {await MainActor.run
      {
         // set the window content size //
         MainWindow.setContentSize(CGSize(width:660 + BoolTo().CG(Inf
                   .Vals("SmallInfoPanel").VB) * 320 + BoolTo().CG(Inf
                   .Vals("LargeInfoPanel").VB) * 630, height:800))

         // center the main window //
         MainWindow.center()

         // update the window size menu item states and clock origins //
         (frame, HideInfoMenuItem.state, LargeInfoMenuItem.state,
         SmallInfoMenuItem.state, GCUI.GetClockView("WhiteClockView").frame
         .origin, GCUI.GetClockView("BlackClockView").frame.origin) = (CGRect(x:
         0, y:YOffset, width:660, height:660), UpdatedState(Inf
         .Vals("HideInfoPanel").VB), UpdatedState(Inf.Vals("LargeInfoPanel")
         .VB), UpdatedState(Inf.Vals("SmallInfoPanel").VB), CGPoint(x:205, y:2),
         CGPoint(x:350, y:2))

         // if have a small info panel //
         if Inf.Vals("SmallInfoPanel").VB
         {
            // update the frames for the small info panel //
            (GCUI.Obj("GameInfoTableView")!.Place, GCUI
            .Obj("MoveListTableView")!.Place) = (Placement(XP:670, YP:392.5,
            Width:300, Height:372.5), Placement(XP:670, YP:10, Width:300,
            Height:372.5))
         }
         // if have a large info panel //
         else if Inf.Vals("LargeInfoPanel").VB
         {
            // update the frames for the large info panel //
            (GCUI.Obj("GameInfoTableView")!.Place, GCUI
            .Obj("MoveListTableView")!.Place) = (Placement(XP:980, YP:10,
            Width:300, Height:755), Placement(XP:670, YP:10, Width:300, Height:
            755))
         }

         // initialize the mouse tracker //
         Tracker.Initialize(self, Margin:Dims.Border.Width, Bounds:bounds,
                            YOffset:YOffset)

         // refresh the views //
         RefreshViews(["GameInfoTableView":false, "MoveListTableView":false,
                       "PosEditorToolbar":false, "DiagramToolbar":false,
```

```
                "AnalysisToolbar":false, "ExitGameToolbar":false])

        // set the main window to be visible //
        MainWindow.alphaValue = 1

        // refresh the table views //
        GCUI.RefreshViews(["MoveListTableView":false, "GameInfoTableView":
                           false])
    })

    // refresh the table views //
    GCUI.RefreshViews(["MoveListTableView":false, "GameInfoTableView":false])
}

// intercept a key down event //
@MainActor func InterceptKeyDown(_ Event:NSEvent)
{
    // switch over the keycode //
    switch Event.keyCode
    {
        // if key pressed was the up or right arrow //
        case 124, 126 :

            // break out of switch if currently in a panel //
            if Inf.Vals("InPanel").VB {break}

            // if the database explorer window is visible //
            if IsAttached(DatabaseExplorerWindow)
            {
                // go right in the database explorer //
                GetDatabaseExplorerData(1)
            }
            // if game explorer window is visible and not reviewing a game //
            else if IsAttached(GameExplorerWindow) && !Inf.Vals("ReviewingGame")
                    .VB
            {
                // go right in the game explorer //
                GameExplorerMoveShift(1)
            }
            // if the load position window is visible //
            else if IsAttached(LoadPositionWindow)
            {
                // go right in the position images index //
                LoadPosIndex = min(PosImages.count - 1, LoadPosIndex + 1)

                // update the load position image //
                GCUI.Objs["LoadPosImage"]!.TheImage = PosImages[LoadPosIndex]
            }
            // if the game gallery window is visible and not reviewing //
            else if IsAttached(GameGalleryWindow) && !GCUI
                    .GetGalleryView("GalleryView").Inf.Vals("Reviewing").VB
            {
                // go right one frame in the game gallery //
                GameGalleryShift(1)
            }

        // if key pressed was the down or left arrow //
        case 123, 125 :

            // break out of switch if currently in a panel //
            if Inf.Vals("InPanel").VB {break}

            // if the database explorer window is visible //
            if IsAttached(DatabaseExplorerWindow)
            {
                // go left in the database explorer //
                GetDatabaseExplorerData(-1)
            }
            // if game explorer window is visible and not reviewing a game //
            else if IsAttached(GameExplorerWindow) && !Inf.Vals("ReviewingGame")
                    .VB
            {
                // go left in the game explorer //
                GameExplorerMoveShift(-1)
            }
            // if the load position window is visible //
            else if IsAttached(LoadPositionWindow)
            {
                // go left in the position images index //
                LoadPosIndex = max(LoadPosIndex - 1, 0)

                // update the load position image //
                GCUI.Objs["LoadPosImage"]!.TheImage = PosImages[LoadPosIndex]
            }
            // if the game gallery window is visible and not reviewing //
            else if IsAttached(GameGalleryWindow) && !GCUI
                    .GetGalleryView("GalleryView").Inf.Vals("Reviewing").VB
            {
                // go left one frame in the game gallery //
                GameGalleryShift(-1)
            }

        // if the spacebar was pressed //
        case 49 :

            // if in game explorer, review the game //
            if IsAttached(GameExplorerWindow) {ReviewGame()}

            // if in game gallery window, review the gallery //
            if IsAttached(GameGalleryWindow) {ReviewGallery()}

        // if the escape key was pressed //
        case 53 :

            // break out of switch if currently in a panel //
            if Inf.Vals("InPanel").VB {break}

            // if in fullscreen mode //
            if Inf.Vals("FullScreen").VB {MainWindow.toggleFullScreen(nil)}

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

// remove the extra edit items //
@MainActor @objc func RemoveEditItems()
{
    // assign the edit menu //
    let EditMenu = NSApplication.shared.mainMenu!.item(withTitle:"Edit")

    // exit if the edit menu doesn't exist //
    if EditMenu == nil {return}

    // loop over the edit menu items to remove //
    while EditMenu!.submenu!.numberOfItems > 9
    {
        // remove the edit menu item at the specified position //
        EditMenu!.submenu!.removeItem(at:9)
    }
}

// clear the selected square if need to //
@MainActor func ClearSelectedSquareIfNeeded()
{
    // if moving by selecting squares and have selected square //
    if GCUI.BPComp.SelectedSquare
    {
        // reset the moving piece //
        MovingPiece.Reset()

        // initialize the selected square //
        GCUI.BPComp.InitSelectedSquare()

        // refresh all //
        Refresh("All")
    }
}

// validate whether corresponding menu item should be active or inactive //
@MainActor func validateMenuItem(_ MenuItem:NSMenuItem) -> Bool
{
    // clear the selected square if needed //
    ClearSelectedSquareIfNeeded()

    // whether playing against human, game is over, or playing timed game //
    Inf.Update("VB", VBItems:["HumanOrGameOverOrTimed":Inf
              .Vals("HumanOpponent").VB || BoardGameOver(Board) || Inf
              .Vals("PlayingTimedGame").VB])

    // set the menu items dictionary //
    MenuItems = (Inf.Vals("InPanel").VB || MovingPiece.HaveMovingPiece() ||
                MovingPiece.Animating()) ? 3 : ((HaveAttachedSheet() || Inf
                .Vals("MainWindowHidden").VB) ? 2 : (Inf.Vals("Diagramming")
                .VB ? 5 : (Inf.Vals("InPositionEditor").VB ? 6 : (Inf
                .Vals("EngineThinking").VB ? 1 : (Inf.Vals("AnalyzingGame").VB
                ? 4 : (Inf.Vals("PlayingTimedGame").VB ? 7 : 0))))))

    // return the current menu item state //
    return MenuItemStates(MenuItem:MenuItem.tag, Dictionary:MenuItems,
           FullScreen:Inf.Vals("FullScreen").VB, HumanOrGameOverOrTimed:
           Inf.Vals("HumanOrGameOverOrTimed").VB)
}

// make a move on the board //
@MainActor func MakeMoveOnBoard(_ Move:String)
{
    // mark move as invalid and exit if not able to find the move //
    if !BoardFindMove(Board, Move)
    {
        // turn off the move exists value //
        Inf.TurnOffVB("VB", Key:"MoveExists")

        // exit the function //
        return
    }

    // make the move on the board //
    BoardMakeMove(Board, 0)

    // update the capture and threaten highlights //
    UpdateCaptureThreatenHighlightLists()

    // set that move exists and was able to be found from possible moves //
    Inf.TurnOnVB("VB", Key:"MoveExists")
}

// stop the stockfish engine from computing //
@MainActor @objc func StopStockfish() {Stockfish.WriteStockfishToStop()}

// start the engine thinking //
nonisolated func StartEngineThinking()
{
    // the dynamic search time //
    var DynamicSearchTime:Int = 0

    // not playing against a uci engine opponent //
    if !Inf.Vals("UCIEngineOpponent").VB
    {
        // if not playing a timed game //
        if !Inf.Vals("PlayingTimedGame").VB
        {
            // set the dynamic search time to the regular search time //
            DynamicSearchTime = Inf.Vals("SearchTime").VI
        }
        // if playing a timed game //
        else
        {
            // get the remaining time left //
            TimeLeft = TimedGameColor == 0 ? RemainingBlackTime :
                       RemainingWhiteTime

            // if less than five minutes left in game, set the search time //
            DynamicSearchTime = TimeLeft < 3000 ? TimeLeft / 200 + 10 :
                                TotalGameTime / 600
        }

        // get best move from the chess engine with max ply depth of 100 //
        MovingPiece.SaveUCIMove(String(cString:EngineFindBestMove(Engine,
                                Board, CInt(DynamicSearchTime), 100))))

        // exit function //
        return
    }

    // if multithreading stockfish engine, use maximum number of threads //
    if Inf.Vals("MultithreadUCIEngine").VB
    {
        // write maximum number of threads to use to the stockfish engine //
        Stockfish.WriteStockfishMaxThreads()
    }

    // if not multithreading the stockfish engine, use only one thread //
    if !Inf.Vals("MultithreadUCIEngine").VB
    {
        // write to just use a single thread to the stockfish engine //
        Stockfish.WriteStockfishSingleThread()
    }

    // set the uci engine difficulty option //
    Stockfish.WriteStockfishSkillLevel(Inf.Vals("UCIEngineDifficulty").VI)

    // set the uci engine hash option //
    Stockfish.WriteStockfishHashSize(Inf.Vals("UCIEngineHash").VI)

    // set the uci engine to use nnue //
    Stockfish.WriteStockfishUseNNUE()

    // write to stockfish current pos using the fen string and move list //
    Stockfish.WriteStockfishPositionFromFEN(GCUI.Attributes.FENString(),
              MoveList:GCUI.Attributes.Inf.GetVSList("Moves"))

    // write move command to stockfish engine with a max depth of 200 plies //
    Stockfish.WriteStockfishToMakeNewMove(200)

    // run on the main thread //
    Task {await MainActor.run
    {
        // set the stockfish timer //
        StockfishTimer = Timer.scheduledTimer(timeInterval:
                         Double(OnlyOnePossibleMove(Board) ? 1 : (Inf
                         .Vals("PlayingTimedGame").VB ? 5 : Inf
                         .Vals("SearchTime").VI)), target:self, selector:
                         #selector(StopStockfish), userInfo:nil, repeats:false)

        // add the stockfish timer to the main run loop //
        RunLoop.current.add(StockfishTimer, forMode:RunLoop.Mode.common)
    }}

    // get the best uci move from the stockfish engine //
    MovingPiece.SaveUCIMove(Stockfish.FindBestMove())

    // run on the main thread to invalidate the stockfish timer //
    Task {await MainActor.run {StockfishTimer.invalidate()}}

    // initialize the stockfish principle variation string to be empty //
    StockfishPV = ""

    // if principal variation exists in information string from stockfish //
    if (Stockfish.GetInfoString().range(of:" pv ") != nil) && Inf
       .Vals("ShowStockfishAnalysis").VB
    {
        // get the principal variation from the stockfish information string //
        StockfishPV = String(Stockfish.GetInfoString().suffix(from:(Stockfish
                      .GetInfoString().range(of:" pv ")?.upperBound)!))
    }
}

// compute a chess engine move //
nonisolated func ComputeEngineMove()
{
    // if first move in game from initial fen pos and not use opening book //
    if GCUI.Attributes.NoMoves() && GCUI.Attributes.IsDefaultFENAndWhite() &&
       !Inf.Vals("UseEcoBook").VB
    {
        // set the first move for white to be a standard opening //
        MovingPiece.SaveUCIMove("e2e4")
    }
    // not the very first move in a game from initial fen position //
    else
    {
        // if move exists for current board pos and using the opening book //
        if OpeningBookMoveExists(OpeningBook, BoardGetZobristHash(Board)) &&
           Inf.Vals("UseEcoBook").VB
        {
            // get the move from the chess opening book //
            MovingPiece.SaveUCIMove(String(cString:
                                    OpeningBookGetNextMove(OpeningBook)))

            // if can't find move from opening book, start engine thinking //
            if !BoardFindMove(Board, MovingPiece.UCIMove)
            {
                // start the engine thinking //
                StartEngineThinking()
            }
        }
        // not using the opening book or move not in the opening book //
        else
        {
            // start the chess engine thinking //
            StartEngineThinking()
        }
    }

    // run on the main thread //
    Task {await MainActor.run
    {
        // if need to stop the chess engines //
        if Inf.Vals("StopEngines").VB
        {
            // turn off the stop engines value //
            Inf.TurnOffVB("VB", Key:"StopEngines")

            // exit the function //
            return
        }

        // add the uci move to the move list //
        GCUI.Attributes.Inf.SaveNextVS("Moves", VS:MovingPiece.UCIMove)

        // store the autosave data //
        StoreAutoData("")

        // perform an engine move on the main thread //
        PerformEngineMove()
    }}
}

// redo the next move that had been made or undone //
@MainActor @IBAction func RedoLastMoveMenuItemAction(_:Any?Object)
{
    // redo to the next move //
    if GCUI.Attributes.Inf.RedoToNextVS() {Refresh("All"); StoreAutoData("")}
}

// update the destination square that the piece moves to //
@MainActor func UpdateDestinationSquare()
{
    // store captured piece if legal move and not query for pawn promotion //
    if !Inf.Vals("QueryPawnPromotion").VB && MovingPiece.IsLegal()
    {
        // store the captured piece at the specified square //
        BPData.StoreCapture(BoardIsReversed(Board) ? (BoardIsWhite(Board) ? 1
                            : -1) : BPData.EndSquareValue())
    }

    // put the piece that moved onto the destination square //
    BPData.EndSquareValueUpdate(MovingPiece.MovingPiece)

    // if the moving piece is illegal, reset the moving piece and exit //
    if MovingPiece.IsIllegal() {MovingPiece.Reset(); return}

    // perform an enpassant move if needed //
    if BoardIsEnpassant(Board) {BPData.RemovePawnCapturedEnPassant()}

    // if had a kingside castling move by white, move rook to correct pos //
    if MovingPiece.Castl("e1g1", Piece:6)
    {
        // move rook to the correct position //
        BPData.RFU(7, 0, 0)

        // move king to the correct position //
        BPData.RFU(5, 0, 4)
    }
    // if had a queenside castling move by white, move rook to correct pos //
    else if MovingPiece.Castl("e1c1", Piece:6)
    {
        // move rook to the correct position //
        BPData.RFU(0, 0, 0)

        // move king to the correct position //
        BPData.RFU(3, 0, 4)
    }
    // if had a kingside castling move by black, move rook to correct pos //
    else if MovingPiece.Castl("e8g8", Piece:-6)
    {
        // move rook to the correct position //
        BPData.RFU(7, 7, 0)

        // move king to the correct position //
        BPData.RFU(5, 7, -4)
    }
    // if had a queenside castling move by black, move rook to correct pos //
    else if MovingPiece.Castl("e8c8", Piece:-6)
    {
        // move rook to the correct position //
        BPData.RFU(0, 7, 0)

        // move king to the correct position //
        BPData.RFU(2, 7, -4)
    }

    // reset the piece moving to be none //
    MovingPiece.Reset()
}

// make an alert from the given buttons and the user query message //
@MainActor func MakeAlert(_ Buttons:[String], Query:String)
{
    // if already in a panel, return //
    if Inf.Vals("InPanel").VB {return}

    // start the sheet //
    StartSheet()

    // create the user alert //
    Alert = NSAlert()

    // loop over all the alert buttons //
    for TheButton in Buttons {Alert.addButton(withTitle:TheButton)}

    // set the alert icon //
    (Alert.messageText, Alert.icon) = (Query, Inf.Vals("QueryInvalidPGNFile")
    .VB ? WarningIcon : (Inf.Vals("InPositionEditor").VB ? MessageIcon :
    (Inf.Vals("InPositionEditor").VB ? (FENLayoutIsLegal() ? ValidatedIcon :
    WarningIcon) : (Inf.Vals("QuerySavePosition").VB ? Graphics
    .CGImageToNSImage(BoardImage(-1, Size:200, Update:false), Size:100) : (Inf
    .Vals("QueryLoadPosition").VB ? WarningIcon : MessageIcon)))))

    // run alert and reopen the previous window //
    Alert.beginSheetModal(for:CurrentWindow(), completionHandler:{Result ->
          Void in self.ReturnCode = Result.rawValue; self.ProcessAlert(); self
          .EndSheet()})
}

// return whether the specified window is attached or not //
@MainActor func IsAttached(_ Window:NSWindow) -> Bool
{
    // return whether the specified window is attached or not //
    return (MainWindow.attachedSheet != nil) ? MainWindow.attachedSheet ==
           Window : false
}

// initialize the game clock //
@MainActor func InitializeGameClock()
{
    // update the game time for white and black //
    (ElapsedWhiteTime, ElapsedBlackTime) = (TimedGameOffset, TimedGameOffset)

    // refresh the clocks //
    (GCUI.GetClockView("WhiteClockView").RefreshAll, GCUI
    .GetClockView("BlackClockView").RefreshAll) = (true, true)
}

// update the game clock //
@MainActor @objc func UpdateGameClock()
{
    // update the progress //
    ProgressCircle.UpdateProgress()

    // if clock preview needs to be refreshed, update time clock preview //
    if GCUI.GetClockView("ClockView").RefreshAll
    {
        // update the time clock preview //
        GCUI.GetClockView("ClockView").UpdateClock(0)
    }

    // if querying user that timed game is over, return //
    if Inf.Vals("QueryTimedGameOver").VB {return}

    // if in pos editor, diagramming, in gallery, or have attached sheet //
    if Inf.Vals("InPositionEditor").VB || Inf.Vals("Diagramming").VB ||
       HaveAttachedSheet()
    {
        // for all the clocks //
        for TheClock in [GCUI.GetClockView("WhiteClockView"), GCUI
            .GetClockView("BlackClockView")]
        {
            // hide the clock //
            (TheClock.isHidden, TheClock.alphaValue) = (true, 0)
        }

        // exit the function since don't want to update the game clock //
        return
    }

    // for all the clocks //
    for TheClock in [GCUI.GetClockView("WhiteClockView"), GCUI
        .GetClockView("BlackClockView")]
    {
        // update the clock hidden status //
        (TheClock.isHidden, TheClock.alphaValue) = (Inf
        .Vals("HideGameClockInPanel").VB, Inf.Vals("HideGameClockInPanel").VB
        ? 0 : 1)
    }

    // if the game clock has started //
    if Inf.Vals("StartClock").VB
    {
        // if it is currently white's turn //
        if (BoardIsWhite(Board) && !(MovingPiece.Animating() || MovingPiece
           .IsIllegal())) || (BoardIsBlack(Board) && MovingPiece.Animating()
           && MovingPiece.IsWhite())
        {
            // increment the time for white //
            ElapsedWhiteTime += 1
        }
        // if it is currently black's turn //
        else
        {
            // increment the time for black //
            ElapsedBlackTime += 1
        }
    }

    // if playing a timed game //
    if Inf.Vals("PlayingTimedGame").VB
    {
        // calculate the time remaining for white and black //
        (RemainingWhiteTime, RemainingBlackTime) = (216000 - ElapsedWhiteTime,
        216000 - ElapsedBlackTime)

        // if at max time by white for game, show that white lost on time //
        if ElapsedWhiteTime >= 216000 {GameState = 5}

        // if at max time by black for game, show that black lost on time //
        if ElapsedBlackTime >= 216000 {GameState = 6}

        // if playing a timed game and current game state is nonzero //
        if GameState > 0
        {
            // set that querying user that timed game is now over //
            Inf.TurnOnVB("VB", Key:"QueryTimedGameOver")

            // if the current game state exists in the game status dictionary //
            if GameOver[GameState] != nil
            {
                // show an alert for the current game state //
                MakeAlert(["Okay"], Query:GameOver[GameState]!)
            }
            // current game state doesn't exist in the game status dictionary //
            else
            {
                // set query timed game over to false //
                Inf.TurnOffVB("VB", Key:"QueryTimedGameOver")
            }
        }
    }

    // update white time clock if the time changed or need to refresh all //
    if (ElapsedWhiteTime != GCUI.GetClockView("WhiteClockView").ElapsedTime)
       || GCUI.GetClockView("WhiteClockView").RefreshAll
    {
        // update the white time clock //
        GCUI.GetClockView("WhiteClockView").UpdateClock(ElapsedWhiteTime)
    }

    // update black time clock if the time changed or need to refresh all //
    if (ElapsedBlackTime != GCUI.GetClockView("BlackClockView").ElapsedTime)
       || GCUI.GetClockView("BlackClockView").RefreshAll
    {
        // update the black time clock //
        GCUI.GetClockView("BlackClockView").UpdateClock(ElapsedBlackTime)
    }
}

// start a new game //
@MainActor func StartNewGame(_ Tasks:String)
{
    // if uci engine loaded and playing against uci engine, tell new game //
    if Inf.Vals("UCIEngineOpponent").VB {Stockfish.WriteStockfishNewGame()}

    // set that initially haven't started the clock //
    Inf.TurnOffVB("VB", Key:"StartClock")

    // initialize variables to empty //
    (GameState, RemainingWhiteTime, RemainingBlackTime, TimedGameOffset,
    TotalGameTime) = (0, 0, 0, 0, 0)

    // set the initial moving piece to none //
    MovingPiece.Reset()

    // initialize the selected squares to false //
    GCUI.BPComp.InitSelectedSquare()

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

    // initialize the game attributes //
    GCUI.Attributes.Initialize()

    // initialize the board //
    InitializeBoard()

    // initialize the transposition table //
    EngineInitializeTransTable(Engine)

    // refresh all the views //
    Refresh(Tasks)
}

// initialize the chessboard //
@MainActor func InitializeBoard()
{
    // set the initial layout of the pieces on the chessboard //
    SetInitialLayout()

    // initialize the chessboard with the initial fen strings //
    InitializeBoardAttributes(Board)

    // set start square to end square so can tell when to highlight a move //
    BPData.SquaresEqual()

    // update the capture and threaten highlights //
    UpdateCaptureThreatenHighlightLists()

    // update the chess board data for the starting board position //
    UpdateBoardData(0)
}

// check whether specified enpassant square from position editor is valid //
@MainActor func CheckEnpassantSquare()
{
    // set the FEN enpassant square from position editor isn't valid //
    Inf.TurnOffVB("VB", Key:"ValidFENEnpassantSquare")

    // if size of the FEN EP square is of a valid size for square position //
    if GCUI.Attributes.Inf.Vals("FENEnpassantSquare").VS.count == 2
    {
        // get file and rank from the FEN enpassant target square location //
        (File, Rank) = (SquareLabel.FileIndex(GCUI.Attributes.Inf
        .Vals("FENEnpassantSquare").VS.first!), SquareLabel.RankIndex(GCUI
        .Attributes.Inf.Vals("FENEnpassantSquare").VS.last!))

        // if the file for the enpassant square location is within the board //
        if (0...7).contains(File)
        {
            // check if the enpassant square specified is a valid one //
            if ((Rank == 2) && BPData.RFE(File, 3, 1) && BPData.RFE(File, 2, 0)
               && BPData.RFE(File, 1, 0) && GCUI.Attributes.BlackStarts()) ||
               ((Rank == 5) && BPData.RFE(File, 4, -1) && BPData.RFE(File, 5, 0)
               && BPData.RFE(File, 6, 0) && GCUI.Attributes.WhiteStarts())
            {
                // set that the enpassant square specified is a valid one //
                Inf.TurnOnVB("VB", Key:"ValidFENEnpassantSquare")
            }
        }
    }

    // if FEN enpassant square specified by user is not valid, set to empty //
    if !Inf.Vals("ValidFENEnpassantSquare").VB
    {
        // update the enpassant square //
        GCUI.Attributes.Inf.Update("FENPref", VSItems:["FENEnpassantSquare":
                                   "-"])
    }
}

// determine if the fen layout from the position editor is legal or not //
@MainActor func FENLayoutIsLegal() -> Bool
{
    // initially set that the fen layout from the position editor is legal //
    Inf.TurnOnVB("VB", Key:"LegalFENLayout")

    // initialize number of white and black pieces that were promoted //
    (PromotedWhitePieces, PromotedBlackPieces) = (0, 0)

    // convert the current board layout to Forsyth-Edwards notation //
    ConvertLayoutToFEN()

    // create the castling rights in terms of FEN notation //
    CreateCastlingRights()

    // check whether FEN enpassant target square specified is valid or not //
    CheckEnpassantSquare()

    // initialize different piece types in fen layout from position editor //
    PiecesInFENLayout = Array(repeating:0, count:13)

    // loop over all the files on the board //
    for File in 0...7 {for Rank in 0...7
    {
        // increment the pieces of each type //
        PiecesInFENLayout[BPData.RFV(File, Rank) + 6] += 1
    }}

    // loop over all the files to see if have any illegally placed pawns //
    for File in 0...7 {if BPData.RFAE(File, 0, 1) || BPData.RFAE(File, 7, 1)
    {
        // set the legal fen layout to false //
        Inf.TurnOffVB("VB", Key:"LegalFENLayout")
    }}

    // set max num of pieces that could be promoted by white and black //
    (AllowedWhitePromotedPieces, AllowedBlackPromotedPieces,
    PromotedWhitePieces, PromotedBlackPieces) = (8 - PiecesInFENLayout[7], 8
    - PiecesInFENLayout[5], max(0, PiecesInFENLayout[8] - 2) + max(0,
    PiecesInFENLayout[9] - 2) + max(0, PiecesInFENLayout[10] - 2) + max(0,
    PiecesInFENLayout[11] - 1), max(0, PiecesInFENLayout[4] - 2) + max(0,
    PiecesInFENLayout[2] - 2) + max(0, PiecesInFENLayout[3] - 2) + max(0,
    PiecesInFENLayout[4] - 2))

    // if white or black promoted more than possible //
    if (PromotedWhitePieces > AllowedWhitePromotedPieces) ||
       (PromotedBlackPieces > AllowedBlackPromotedPieces) ||
       (PiecesInFENLayout[0] != 1) || (PiecesInFENLayout[12] != 1) ||
       (PiecesInFENLayout[5] > 8) || (PiecesInFENLayout[7] > 8)
    {
        // set that current fen layout from edited board position is illegal //
        Inf.TurnOffVB("VB", Key:"LegalFENLayout")
    }

    // if the fen layout still has a possibility of being a legal position //
    if Inf.Vals("LegalFENLayout").VB
    {
        // initialize the king captured test board with fen color //
        InitializeBoardAttributes(KingCapturedTestBoard)

        // if king could be captured in fen layout position //
        if BoardKingIsCaptured(KingCapturedTestBoard)
        {
            // set that fen layout is not legal //
            Inf.TurnOffVB("VB", Key:"LegalFENLayout")
        }
    }

    // return whether current fen layout from board position is legal //
    return Inf.Vals("LegalFENLayout").VB
}

// initialize the board attributes //
@MainActor func InitializeBoardAttributes(_ InitializingBoard:
                                          UnsafeMutableRawPointer)
{
    // initialize board attributes with fen info from game attributes //
    BoardInitializeAttributes(InitializingBoard, GCUI.Attributes.Inf
                              .Vals("FENLayout").VS, GCUI.Attributes.Inf
                              .Vals("FENCastlingRights").VS, GCUI.Attributes
                              .Inf.Vals("FENColor").VS, GCUI.Attributes.Inf
                              .Vals("FENEnpassantSquare").VS)
}

// the total pieces on the board //
@MainActor func NumPiecesOnBoard() -> Int
{
    // initialize total pieces on the board to zero //
    TotalPieces = 0

    // loop over all the files on the board //
    for File in 0...7 {for Rank in 0...7
    {
        // increment the total number of pieces on the board //
        if BPData.RFNE(File, Rank, 0) {TotalPieces += 1}
    }}

    // return the total number of pieces on the board //
    return TotalPieces
}

// save and exit the position editor //
@MainActor func PositionEditorSave()
{
    // if the board currently has no pieces on it //
    if NumPiecesOnBoard() == 0
    {
        // exit the position editor //
        ExitPositionEditor()

        // exit the function //
        return
    }

    // set that currently querying if want to save edited board position //
    Inf.TurnOnVB("VB", Key:"QuerySaveEditedPosition")

    // if the fen position from the position editor is legal //
    if FENLayoutIsLegal()
    {
        // make the query if want to save the edited position alert //
        MakeAlert(["Save", "Edit", "Exit"], Query:"Position Validated")
    }
    // the fen position from the position editor is illegal //
    else
    {
        // tell that edited board position is illegal and need to fix it //
        MakeAlert(["Edit", "Exit"], Query:"Invalid Position")
    }
}

// convert the edited board position to a fen string representation //
@MainActor func ConvertLayoutToFEN()
{
    // initialize the fen layout string //
    GCUI.Attributes.Inf.ClearVS("FENPref", Key:"FENLayout")

    // loop over all the ranks on the board //
    for Rank in (0...7).reversed()
    {
        // initialize the count of the empty squares //
        Count = 0

        // loop over all the files on the board //
        for File in 0...7
        {
            // get the piece type at the current position on the board //
            PieceType = BPData.RFV(File, Rank)

            // if have empty square, increment count of empty squares //
            if PieceType == 0 {Count += 1}

            // if had empty squares and encountered piece or end of the rank //
            if (Count > 0) && ((PieceType != 0) || ((PieceType == 0) && (File
               == 7)))
            {
                // append the number of empty squares to the fen layout string //
                GCUI.Attributes.Inf.AddTo("FENPref", Key:"FENLayout", AddOn:
                                          "\(Count)")

                // reset the empty square count to zero //
                Count = 0
            }

            // if the piece is valid //
            if (abs(PieceType) <= 6) && (PieceType != 0)
            {
                // add the corresponding fen layout character //
                GCUI.Attributes.Inf.AddTo("FENPref", Key:"FENLayout", AddOn:
                                          FENCharacters[PieceType + 6])
            }

            // if at end of a rank and not on first rank, add forward slash //
            if (File == 7) && (Rank != 0)
            {
                // add a slash to the fen layout key value //
                GCUI.Attributes.Inf.AddTo("FENPref", Key:"FENLayout", AddOn:"/")
            }
        }
    }
}

// initialize the app preferences //
@MainActor func InitializePreferences()
{
    // get the boolean and integer preferences //
    Inf.GetPrefs(["VBPref"], VITags:["VIPref"], VSTags:["VSPref"])

    // initialize the control buttons and time clock hidden status //
    (GCUI.GetClockView("WhiteClockView").isHidden, GCUI
    .GetClockView("BlackClockView").isHidden, GCUI.GameSounds.SoundOff) =
    (Inf.Vals("HideGameClockInPanel").VB, Inf.Vals("HideGameClockInPanel").VB,
    Inf.Vals("SoundOff").VB)

    // read in the move list font //
    MoveListFont.Read(27, TheScale:1, Tag:"MoveList")

    // read in the game info font //
    GameInfoListFont.Read(27, TheScale:1, Tag:"GameInfo")

    // read in the game list font //
    GameListFont.Read(27, TheScale:1, Tag:"GameList")

    // initialize the window background color view //
    GCUI.GetColorView("WindowColorView").Initialize(Inf
        .Vals("WindowBackgroundHue").VI)

    // set the window background color //
    SetWindowBackgroundColor(false)

    // read in preferences used to create the board and piece composites //
    GCUI.BPComp.Read(WindowTexture)

    // read in the frames movie preferences //
    GCUI.FramesMovie.ReadPrefs()

    // store the game clock color set //
    GCUI.GetColorView("ClockColorView").Initialize(Inf.Vals("GameClockHue")
                                                   .VI)

    // update the clock color //
    SetGameClockColor()

    // store the game clock background color set //
    GCUI.GetColorView("ClockSurfaceColorView").Initialize(Inf
        .Vals("GameClockBackgroundHue").VI)

    // set the game clock background color //
    SetGameClockBackgroundColor()

    // update the clock background visibility //
    UpdateClockVisibilityForBackground()

    // update the progress circle color //
    ProgressCircle.UpdateCircleColor(GCUI.GetColorView("WindowColorView")
                                     .GetCurrentHue())

    // set the opponent to play against with greenchess as the default //
    UpdateOpponent(Inf.Vals("Opponent").VS)

    // loop over all the saved positions //
    for PositionListIndex in 1...25
    {
        // if the saved game position exists //
        if !GCUI.Attributes.Initialize("\(PositionListIndex)")
        {
            // convert the current game to game data //
            ConvertToGameData()

            // create new element at the end of saved positions data array //
            SavedPositionData.append(DataNew())

            // copy game data to the end of the saved positions data array //
            DataCopy(SavedPositionData.last!, GameData)
        }
    }

    // get the game attributes //
    _ = GCUI.Attributes.Initialize("")

    // if do autosave of game, convert autosave data to game data to store //
    if Inf.Vals("AutosaveGame").VB {ConvertToGameData()}
}

// store the autosave data //
@MainActor func StoreAutoData(_ Suffix:String)
{
    // if not autosave data or a playing a game and not saving pos, return //
    if !Inf.Vals("AutosaveGame").VB || (Inf.Vals("ExploringGame").VB && !Inf
       .Vals("QuerySavePosition").VB)
    {
        // exit the function since won't store the data to preferences //
        return
    }

    // store the autosave data for the game attributes //
    GCUI.Attributes.StoreAutoData(Suffix)
}

// set the initial board layout and check for errors //
@MainActor func SetInitialLayout()
{
```

```
    // compute the initial board layout //
    ComputeLayout()

    // if fen string not valid, set fen defaults and compute initial layout //
    if !Inf.Vals("ValidFENString").VB
    {
      // set the fen default values //
      GCUI.Attributes.SetFENDefaults()

      // compute the layout //
      ComputeLayout()
    }

    // if fen layout not legal, set fen defaults and compute initial layout //
    if !FENLayoutIsLegal()
    {
      // set the fen default values //
      GCUI.Attributes.SetFENDefaults()

      // compute the layout //
      ComputeLayout()
    }
  }
}

// compute the initial board layout //
@MainActor func ComputeLayout()
{
  // set that valid fen string is initially true //
  Inf.TurnOnVB("VB", Key:"ValidFENString")

  // set that initially have no moving piece at beginning of game //
  MovingPiece.Reset()

  // initialize the board layout //
  BPData.InitializeLayout()

  // set the initial rank and file values on the board //
  (Rank, File) = (7, 0)

  // loop over the entire fen layout string //
  for Index in 0..<GCUI.Attributes.Inf.Vals("FENLayout").VS.count
  {
    // if the fen layout was corrupted and out of range //
    if !(0...7).contains(Rank) || !(0...7).contains(File)
    {
      // set the valed fen string to false //
      Inf.TurnOffVB("VB", Key:"ValidFENString")

      // exit the loop //
      break
    }

    // get the current piece from the fen string //
    FENPiece = GCUI.Attributes.Inf.GetC("FENLayout", Index:Index)

    // if the piece is in the piece dictionary //
    if PieceDict[FENPiece] != nil
    {
      // add the corresponding piece to the layout //
      BPData.RFU(File, Rank, PieceDict[FENPiece]!)

      // increment the file //
      File += 1
    }
    // if fen piece is just a count of how many empty squares there are //
    else if SquareLabel.IsRank(FENPiece.first!)
    {
      // loop over the count of empty squares in the fen string //
      for _ in 0..<Int(FENPiece)!
      {
        // if the fen layout was corrupted and out of range //
        if !(0...7).contains(Rank) || !(0...7).contains(File)
        {
          // set the valid fen string to false //
          Inf.TurnOffVB("VB", Key:"ValidFENString")

          // exit the loop //
          break
        }

        // set the current layout to an empty square //
        BPData.RFU(File, Rank, 0)

        // increment the file //
        File += 1
      }
    }

    // if reached the end of a rank //
    if File == 8 {(File, Rank) = (0, Rank - 1)}
  }

  // check the fen start move number in the fen string //
  GCUI.Attributes.CheckFENStartMoveNumber()

  // initialize the castles //
  InitializeCastles()

  // check the color to move in the fen string //
  GCUI.Attributes.CheckColorToMove()
}

// initialize the castles //
@MainActor func InitializeCastles()
{
  // turn off the castles //
  Inf.Update("VB", VBItems:["WhiteKingsideCastle":false,
            "WhiteQueensideCastle":false, "BlackKingsideCastle":false,
            "BlackQueensideCastle":false])

  // loop over the castling rights //
  for Index in 0..<GCUI.Attributes.Inf.Vals("FENCastlingRights").VS.count
  {
    // switch over the castling character //
    switch GCUI.Attributes.Inf.GetC("FENCastlingRights", Index:Index)
    {
      // set white kingside castle to true //
      case "K" : Inf.TurnOnVB("VB", Key:"WhiteKingsideCastle")

      // set white queenside castle to true //
      case "Q" : Inf.TurnOnVB("VB", Key:"WhiteQueensideCastle")

      // set black kingside castle to true //
      case "k" : Inf.TurnOnVB("VB", Key:"BlackKingsideCastle")

      // set black queenside castle to true //
      case "q" : Inf.TurnOnVB("VB", Key:"BlackQueensideCastle")

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

// initialize the user interface //
@MainActor func InitializeUI()
{
  // initialize the preferences window //
  PreferencesWindow.Update(660, Height:685)

  // initialize the enter game info window //
  EnterInfoWindow.Update(660, Height:685)

  // initialize the load position window //
  LoadPositionWindow.Update(655, Height:685)

  // initialize the timed game window //
  TimedGameWindow.Update(550, Height:100)

  // initialize the enter fen window //
  FENWindow.Update(660, Height:100)

  // initialize the game list window //
  GameListWindow.Update(660, Height:685)

  // initialize the game explorer window //
  GameExplorerWindow.Update(660, Height:685)

  // initialize the game annotator window //
  GameAnnotatorWindow.Update(660, Height:685)

  // initialize the database explorer window //
  DatabaseExplorerWindow.Update(660, Height:685)

  // initialize the about window //
  AboutWindow.Update(660, Height:525)

  // initialize the game gallery window //
  GameGalleryWindow.Update(660, Height:685)

  // add the colors and clocks views and tables to the screen //
  GCUI.AddNSViewsTables(self)

  // update the boolean values //
  Inf.Update("VB", VBItems:["StopEngines":false, "HumanOrGameOverOrTimed":
      false, "MainWindowHidden":false, "GameExplorerWindowOpen":false,
      "StartClock":false, "FullScreen":false, "InPanel":false,
      "ValidFENString":true, "PlayingEngineMatch":false,
      "ValidFENEnpassantSquare":false, "LegalFENLayout":false, "MoveExists":
      false, "EditingNonemptyPosition":false, "InPositionEditor":false,
      "InDatabaseExplorer":false, "QuerySavePosition":false,
      "QueryLoadPosition":false, "EnteringPreferences":false,
      "InGameGallery":false, "QueryPawnPromotion":false,
      "QueryStartNewGame":false, "QueryGameOver":false,
      "QuerySavedEditedPosition":false, "QueryQuitApplication":false,
      "EngineThinking":false, "HumanOpponent":false, "EngineOpponent":false,
      "UCIEngineOpponent":false, "Diagramming":false, "ReviewingGame":false,
      "ExploringGame":false, "AnnotatingGame":false, "PlayingTimedGame":
      false, "EnteringFEN":false, "AnalyzingGame":false,
      "WillSaveEditedPosition":false, "WhiteKingsideCastle":false,
      "WhiteQueensideCastle":false, "BlackKingsideCastle":false,
      "BlackQueensideCastle":false, "QueryInvalidPGNFile":false])

  // update the string default preferences //
  Inf.Update("VSPref", VSItems:["Opponent":"GreenChess"])

  // update the boolean value defaults for preferences //
  Inf.Update("VBPref", VBItems:["HideIllustration":false,
      "HideGameClockInPanel":false, "MultithreadUCIEngine":false,
      "UseEcoBook":false, "AutosaveGame":false, "ShowStockfishAnalysis":
      false, "HideInfoPanel":false, "SmallInfoPanel":false,
      "LargeInfoPanel":true, "SoundsOff":false])

  // update the integer default, min, and max values for preferences //
  Inf.Update("VIPref", VDefMinMaxItems:["WindowBackgroundHue":[64, 0,
      GCUI.GetClockView("WindowColorView").MaxHue()],
      "WindowTextureOpacity":[0, 0, 100], "IllustrationSize":[200, 150,
      265], "GameClockHue":[27, 0, GCUI.GetClockView("ClockColorView")
      .MaxHue()], "GameClockBackgroundHue":[82, 0, GCUI
      .GetClockView("ClockSurfaceColorView").MaxHue()],
      "ClockBackgroundVisibility":[30, 0, 100], "NumberOfGridRows":[3, 2,
      3], "AnimationRate":[75, 1, 100], "GalleryAnimationRate":[75, 1, 100],
      "UCIEngineDifficulty":[20, 0, 20], "UCIEngineHash":[16, 16, 512],
      "SavedPositionSlot":[1, 1, 5], "SearchTime":[5, 5, 120],
      "PreferencesSelection":[1, 1, 9]])

  // set the current tracking area //
  addTrackingArea(NSTrackingArea(rect:Dims.InnerBounds.Bounds, options:
              [.mouseEnteredAndExited, .mouseMoved, .activeAlways],
              owner:self, userInfo:nil))

  // set button, titles, responders, concurrent drawing //
  (MainWindow.acceptsMouseMovedEvents, MainWindow
  .allowsConcurrentViewDrawing, GameGalleryWindow
  .allowsConcurrentViewDrawing, MainWindow.initialFirstResponder,
  canDrawConcurrently, GoStopMenuItem.title, HideMenuItem.title) = (true,
  true, true, self, true, "Engine", "Hide")

  // add the progress circle to the board //
  ProgressCircle.AddLayer(self)

  // initialize the white time clock //
  GCUI.GetClockView("WhiteClockView").Initialize()

  // initialize the black time clock //
  GCUI.GetClockView("BlackClockView").Initialize()

  // initialize the time clock preview //
  GCUI.GetClockView("ClockView").Initialize()

  // initialize the moving piece attributes //
  MovingPiece.Initialize()

  // initialize all the app preferences //
  InitializePreferences()

  // refresh the board //
  Refresh("Board")

  // update the main window //
  UpdateMainWindow()

  // initialize the sketch //
  Sketch.Initialize(self as NSView, Width:Dims.Board.Width.Ret, Height:Dims
              .Board.Width.Ret, Length:Dims.Board.Width.Ret, YOffset:
              YOffset, Margin:Dims.Border.BoundsRet.midX)

  // add the swiftui views to the screen //
  GCUI.AddSwiftUIViews()

  // refresh the position editor buttons //
  RefreshPosEditorButtons(false)
}

// refresh the position editor buttons //
@MainActor func RefreshPosEditorButtons(_ Visible:Bool)
{
  // run on the main thread //
  Task {await MainActor.run
  {
    // update the visibility of the position editor buttons //
    GCUI.UpdateVisibility(["PosEditorToolbar", "WhiteCastling",
                  "BlackCastling", "EPSquare", "ColorToMove"],
                  Visible:Visible)
  }}
}

// update the mouse cursor //
@MainActor func UpdateCursor()
{
  // if cursor location is off board or attached sheet or query to quit //
  if !Tracker.InBounds() || HaveAttachedSheet() || Inf
    .Vals("QueryQuitApplication").VB
  {
    // set the mouse to the arrow cursor //
    Task {await MainActor.run {NSCursor.arrow.set()}}

    // exit function //
    return
  }

  // update the sketch cursor if in diagramming mode //
  if Inf.Vals("Diagramming").VB {Sketch.Tracker.RefreshCursor(); return}

  // if in the position editor //
  if Inf.Vals("InPositionEditor").VB {PosCursor.set(); return}

  // if not have moving piece and not animating //
  if MovingPiece.HaveMovingPiece() && !MovingPiece.Animating()
  {
    // set the cursor to the pointing hand //
    Task {await MainActor.run {NSCursor.pointingHand.set()}}
  }
  // for all other cases //
  else
  {
    // set the mouse to the open hand cursor //
    Task {await MainActor.run {NSCursor.openHand.set()}}
  }
}

// make an edit on the position editor board //
@MainActor func MakePositionEdit()
{
  // set the moving piece to be the position editor moving piece //
  MovingPiece.Set(PosMovingPiece)

  // add the piece to the chosen square on the board //
  BPData.RFU(MovingPiece.Location, MovingPiece.MovingPiece)

  // reset the piece moving to be none //
  MovingPiece.Reset()

  // refresh the view //
  Refresh("View")
}

// override mouse moved //
@MainActor override func mouseMoved(with Event:NSEvent) {UpdateCursor()}

// override mouse up //
@MainActor override func mouseUp(with Event:NSEvent) {UpdateCursor()}

// override mouse dragged //
@MainActor override func mouseDragged(with Event:NSEvent) {UpdateCursor()}

// function to intercept a mouse down event //
@MainActor override func mouseDown(with Event:NSEvent)
{
  // if shouldn't be accepting any mouse down input from the user //
  if !Tracker.InBounds() || Inf.Vals("EngineThinking").VB || MovingPiece
    .Animating() || (BoardGameOver(Board) && !Inf.Vals("InPositionEditor")
    .VB)
  {
    // return since shouldn't be accepting mouse down input from user //
    return
  }

  // if currently diagramming //
  if Inf.Vals("Diagramming").VB
  {
    // update the mouse cursor type //
    UpdateCursor()

    // start to sketch a diagram //
    Sketch.BeginPath()

    // return since done with diagramming //
    return
  }

  // update the moving piece location //
  MovingPiece.UpdateLoc(Tracker.Location())

  // if currently in the position editor //
  if Inf.Vals("InPositionEditor").VB {MakePositionEdit(); return}

  // if haven't selected a square //
  if !GCUI.BPComp.SelectedSquare
  {
    // the piece that the user has chosen to move //
    MovingPiece.Set(BPData.RFV(MovingPiece.Location))

    // if the piece selected to move is currently the color to move next //
    if MovingPiece.PieceCantMove(Board) {MovingPiece.Reset(); return}

    // set that clock has started //
    Inf.TurnOnVB("VB", Key:"StartClock")

    // set the start square of the uci move //
    MovingPiece.SetUCIMoveStartSquare()
  }

  // if currently moving the piece by selecting squares //
  if GCUI.BPComp.Inf.Vals("MoveBySelectingSquares").VB
  {
    // if haven't selected a square yet //
    if !GCUI.BPComp.SelectedSquare
    {
      // set that have selected a square //
      GCUI.BPComp.SetSelectedSquare()

      // refresh the view //
      Refresh("View")

      // erase the piece from starting square since it is moving now //
      BPData.RFU(MovingPiece.Location, 0)
    }
    // have selected a square //
    else
    {
      // reset selected squares to false //
      GCUI.BPComp.InitSelectedSquare()

      // do the mouse up tasks //
      TheMouseUp()
    }

    // exit function since done //
    return
  }

  // update the mouse cursor type //
  UpdateCursor()

  // unhide the moving piece layer contents //
  MovingPiece.UnHideMovingPiece()

  // erase the piece from the starting square since it is moving now //
  BPData.RFU(MovingPiece.Location, 0)

  // refresh the view //
  Refresh("View")

  // perform the mouse piece movements //
  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 == .leftMouseDragged {MovingPiece.UpdateLocAndPos(Tracker
        .Location())} else {TheMouseUp(); Done.pointee = true}})
}

// the engine match menu item action //
@MainActor @IBAction func EngineMatchMenuItemAction(_:AnyObject)
{
  // set that clock has started //
  Inf.TurnOnVB("VB", Key:"StartClock")

  // set the current engine to uci chess engine //
  UpdateOpponent("Stockfish")

  // make an engine match move to have the uci engine self play //
  MakeEngineMatchMove()
}

// start a move by the chess engine //
@MainActor func StartEngineMove()
{
  // set that engine is currently thinking //
  Inf.TurnOnVB("VB", Key:"EngineThinking")

  // set the go stop menu item to stop //
  GoStopMenuItem.title = "Stop"

  // start the progress circle animation //
  ProgressCircle.StartProgressAnimation()

  // compute the engine move on a background task //
  Task.detached(priority:.background) {self.ComputeEngineMove()}

  // set stop engines to false //
  Inf.TurnOffVB("VB", Key:"StopEngines")
}

// function for a mouse up //
@MainActor func TheMouseUp()
{
  // hide the moving piece layer contents //
  MovingPiece.HideMovingPiece(Tracker.Location())

  // update the start and end squares for the move //
  BPData.UpdateStartAndEndSquareForMove(MovingPiece.UCIMove, Dims:Dims,
      Rotated:GCUI.BPComp.Inf.Vals("RotateBoard").VB)

  // if illegal move //
  if !(BoardFindMove(Board, MovingPiece.UCIMove) || BoardFindMove(Board,
    MovingPiece.UCIMove + "q"))
  {
    // set the move to be illegal //
    MovingPiece.SetIllegal()

    // run the core animation //
    RunCoreAnimation()

    // exit function since had an illegal move //
    return
  }

  // if the moving piece is a pawn and can promote //
  if MovingPiece.PawnCanPromote()
  {
    // update the moving piece to the new location on the board //
    UpdateDestinationSquare()

    // refresh the view //
    Refresh("View")

    // set that currently querying for a pawn promotion //
    Inf.TurnOnVB("VB", Key:"QueryPawnPromotion")

    // alert user as to which type of piece they want to promote pawn to //
    MakeAlert(["Queen", "Rook", "Bishop", "Knight"], Query:
          "Promote Pawn To")

    // exit the function //
    return
  }

  // finish up the move //
  FinishUpMove()
}

// finish up the move //
@MainActor func FinishUpMove()
{
  // add current move to the move list //
  GCUI.Attributes.Inf.SaveNextVS("Moves", VS:MovingPiece.UCIMove)

  // store the autosave data //
  StoreAutoData("")

  // make the move on the board //
  MakeMoveOnBoard(MovingPiece.UCIMove)

  // save the notation //
  GCUI.Attributes.Inf.SaveNotation("Moves", VNS:String(cString:
                  BoardPGNLabel(Board)))

  // update the moving piece to the new location on the board //
  UpdateDestinationSquare()

  // update the chess board data //
  UpdateBoardData(GCUI.Attributes.Inf.GetCount())

  // refresh the view and tables //
  Refresh("ViewTables")

  // play the sound of a thump //
  GCUI.GameSounds.Play("sound_thump")

  // set the query pawn promotion value to false //
  Inf.TurnOffVB("VB", Key:"QueryPawnPromotion")

  // if playing against chess engine and game not over, start engine move //
  if !Inf.Vals("HumanOpponent").VB && !BoardGameOver(Board)
  {
    // start an engine move //
    StartEngineMove()
  }
}

// update the stockfish analysis toggle button //
@MainActor func UpdateStockfishAnalysis(_ Key:String)
{
  // toggle the update stockfish analysis key //
  Inf.ToggleVB("VBPref", Key:Key)

  // refresh all //
  Refresh("All")
}

// update if will have game sounds off //
@MainActor func UpdateGameSoundsOff(_ Key:String)
{
  // update the games sounds off option //
  Inf.ToggleVB("VBPref", Key:Key)

  // update the game sounds off value //
  GCUI.GameSounds.SoundOff = Inf.Vals("SoundOff").VB
}

// override the draw function //
@MainActor override func draw(_ Rect:CGRect)
{
  // if currently diagramming, update the sketch to the screen //
  if Inf.Vals("Diagramming").VB
  {
    // update the screen with the image //
    Sketch.SketchImages.UpdateScreen(bounds)

    // set that no longer displaying to the screen //
    Sketch.Displaying = false
  }
  // if currently not diagramming and the main view image exists //
  else
  {
    // draw the board and pieces image to the view //
    NSGraphicsContext.current?.cgContext.draw(MainViewImage, in:bounds,
                        byTiling:false)

    // update the cursor //
    UpdateCursor()
  }
}

// the snapshot tasks //
@MainActor func SnapshotTasks()
{
  // start the sheet //
  StartSheet()

  // create the save panel //
  SavePanel = NSSavePanel()

  // set the allowed content, file name, prompt, and message //
  (SavePanel.allowedContentTypes, SavePanel.nameFieldStringValue, SavePanel
  .prompt, SavePanel.message) = ([.jpeg], "GreenChessSnapshot",
  "Save Image", "Save Snapshot of Current View to JPEG File")
}

// the movie tasks //
@MainActor func MovieTasks()
{
  // start the sheet //
  StartSheet()

  // create the save panel //
  SavePanel = NSSavePanel()

  // set the allowed content, file name, prompt, and message //
  (SavePanel.allowedContentTypes, SavePanel.nameFieldStringValue, SavePanel
  .prompt, SavePanel.message) = ([.quickTimeMovie], "GreenChessMovie",
  "Save Movie", "Save Current Game to QuickTime Movie File")
}

// the take a board snapshot action //
@MainActor @IBAction func BoardSnapshotAction(_:AnyObject)
{
  // the take a board snapshot action //
  BoardSnapshotAction()
}

// the take a board snapshot action //
@MainActor func BoardSnapshotAction()
{
  // do the snapshot tasks //
  SnapshotTasks()

  // run the panel to save the current board snapshot as jpeg file //
  SavePanel.beginSheetModal(for:CurrentWindow(), completionHandler:
        {Result -> Void in if Result == NSApplication.ModalResponse.OK
        {try? self.SnapshotData().write(to:URL(fileURLWithPath:self
        .SavePanel.url!.path), options:[.atomic])}; self.EndSheet()})
}

// get the board image //
@MainActor func BoardImage(_ Index:Int, Size:CGFloat, Update:Bool) -> Image
{
  // return the board image //
  return Image(BoardImage(Index, Size:Size, Update:Update), scale:2,
        orientation:Image.Orientation.up, label:Text(""))
}

// get the board image //
@MainActor func BoardImage(_ Index:Int, Size:CGFloat, Update:Bool) -> CGImage
{
  // return the board image //
  return GCUI.BPComp.Recompute(Size, BPData:(Index == -1) ? BPData : GCUI
      .PGNMovesData.GetData(Index), InPositionEditor:Inf
      .Vals("InPositionEditor").VB, RankAndFile:MovingPiece.Location,
      UpdateBoard:Update, Info:false)
}

// change to currently playing against stockfish uci chess engine //
@MainActor @IBAction func UCIEngineOpponentMenuItemAction(_:AnyObject)
{
  // update to the stockfish opponent //
  UpdateOpponent("Stockfish")
}

// change to currently playing against the built-in chess engine //
@MainActor @IBAction func EngineOpponentMenuItemAction(_:AnyObject)
{
  // update to the greenchess opponent //
  UpdateOpponent("GreenChess")
}

// change to currently playing against a human //
@MainActor @IBAction func HumanOpponentMenuItemAction(_:AnyObject)
{
  // update to the human opponent //
  UpdateOpponent("Human")
}

// save a chess game to the given filename //
@MainActor func SaveGame(_ Filename:String)
{
  // convert the game to a game data object //
  ConvertToGameData()

  // write the game in pgn notation to the chosen filename //
  DataWritePGN(GameData, Filename, GCUI.Attributes.BlackStarts())
}

// construct pgn string representing game //
@MainActor func GamePGN() -> String
{
  // init number of move plies and add header information to pgn string //
  (Plies, PGN) = (BoolTo().Num(GCUI.Attributes.BlackStarts()), GCUI
  .Attributes.GetPGNGameHeader())

  // loop over all moves in the move list //
  for Index in 0..<GCUI.Attributes.Inf.GetCount()
  {
    // if it is first move and black to make first move from fen info //
    if (Plies == 1) && GCUI.Attributes.BlackStarts()
    {
      // add to pgn string that black moving first as from fen game //
      PGN += GCUI.Attributes.Inf.Vals("FENStartMoveNumber").VS + "... "
    }

    // increment plies //
    Plies += 1

    // if starting a new move number, add move number to pgn string //
    if (Plies % 2) == 1
    {
      // add move number to the pgn string //
      PGN += "\((Plies - 1) / 2 + GCUI.Attributes.FENStart()). "
    }

    // add move to pgn string //
    PGN += GCUI.Attributes.Inf.Vals(Index).VNS

    // if move annotation not empty, add move annotation to pgn string //
    if !GCUI.Attributes.Inf.VLSIsEmpty(Index)
    {
      // append the annotation string //
      PGN += " " + GCUI.Attributes.Inf.Vals(Index).VLS
    }

    // add newline if added another five moves or end of move list //
    PGN += ((Plies % 10) == 0) || (Index == (GCUI.Attributes.Inf.GetCount()
        - 1)) ? "\n" : " "
  }

  // if the move list was empty, store an * to represent no moves in game //
  if GCUI.Attributes.NoMoves() {PGN += "*\n"}

  // append result and newline to pgn string //
  PGN += GCUI.Attributes.Inf.Vals("EndResult").VS + "\n"

  // return the game pgn //
  return PGN
}

// convert the game to a game data object //
@MainActor func ConvertToGameData()
{
  // initialize the game data //
  DataInitializeAttributes(GameData)

  // store the game header data //
  GCUI.Attributes.StoreGameHeaderData(GameData)

  // initialize the game information board //
  InitializeBoardAttributes(Board)

  // loop over all moves in the move list //
  for Index in 0..<GCUI.Attributes.Inf.GetCount()
  {
    // make the move on the data board //
    MakeMoveOnBoard(GCUI.Attributes.Inf.Vals(Index).VS)

    // if the move exists //
    if Inf.Vals("MoveExists").VB
    {
      // store the current move and annotation into the game data //
      DataStoreMove(GameData, BoardPGNLabel(Board), GCUI.Attributes.Inf
              .Vals(Index).VLS)
    }
  }
}

// append the pgn game to a currently existing pgn game //
@MainActor func AppendGame(_ Filename:String)
{
  // exit function if not a valid pgn file //
  if !ValidPGNFile(Filename) {return}

  // convert the current game to data format //
  ConvertToGameData()

  // append the game in pgn notation to the chosen filename //
  DataAppendPGN(GameData, Filename, GCUI.Attributes.BlackStarts())
}

// read the opening books //
nonisolated func ReadOpeningBooks()
{
  // the opening book game list //
  let OpeningBookGameList:UnsafeMutableRawPointer = DatabaseNew()

  // the opening board //
  let OpeningBoard:UnsafeMutableRawPointer = BoardNew()

  // the opening book move list //
  var OpeningBookMoveList:[String] = []

  // the zobrist hash //
  var ZobristHash:CUnsignedLongLong = 0

  // initialize the chess opening book //
  OpeningBookInitializeAttributes(OpeningBook)

  // loop over the list of opening books //
  for TheBook in ["eco", "SemiBenoni"]
  {
    // read in the opening book pgn database from the main bundle //
    DatabaseReadGameCollection(OpeningBookGameList, Bundle.main
                  .path(forResource:TheBook, ofType:"pgn")!)

    // loop over the entire opening book game list //
    for BookIndex in 0..<Int(DatabaseGetDataListSize(OpeningBookGameList))
    {
      // initialize the opening book board //
      BoardInitializeAttributes(OpeningBoard, GCUI.Attributes.Inf
                    .Vals("FENLayout").VDS, "KQkq", "w", "-")

      // get move list for the specified game in opening book game list //
      OpeningBookMoveList = String(cString:
                 DatabaseGetGameDataUCI(OpeningBookGameList,
                 CInt(BookIndex))).components(separatedBy:" ")

      // loop over move list for game in opening book //
      for MoveListCountIndex in 0..<min(8, OpeningBookMoveList.count)
      {
        // get zobrist hash of opening book board before move is made //
        ZobristHash = BoardGetZobristHash(OpeningBoard)

        // find the move on the opening book board //
        BoardFindMove(OpeningBoard,
                OpeningBookMoveList[MoveListCountIndex])

        // make the move on the opening book board //
        BoardMakeMove(OpeningBoard, 0)

        // store next move and zobrist hash in the chess opening book //
        OpeningBookStoreOpeningBookData(OpeningBook, ZobristHash,
                          BoardUCILabel(OpeningBoard))
      }
    }
  }
}

// open the game //
@MainActor func OpenGame(_ Filename:String)
{
  // exit function is not a valid pgn file //
  if !ValidPGNFile(Filename) {return}

  // initialize the game data //
  DataInitializeAttributes(GameData)

  // read in a list of pgn games //
  DatabaseReadGameCollection(GameList, Filename)

  // if there were valid games read in to the game database //
  if DatabaseGetDataListSize(GameList) > 0
  {
    // store the game list into the game list data //
    GameListData = String(cString:DatabaseGetGameDescriptions(GameList))
                .components(separatedBy:"**********")

    // remove the last element of the game list since it is empty //
    _ = GameListData.popLast()

    // set the table list view values to publish //
    (GCUI.Objs["DataExpGame"]!.Val, GCUI.Objs["DataExpGame"]!.MaxV, GCUI
    .Objs["DataExpMove"]!.Val, GCUI.Objs["DataExpMove"]!.MaxV, GCUI
    .Objs["GameListTableView"]!.GameList, GCUI.Objs["GameListTableView"]!
    .Selection, GCUI.Objs["GameListTableView"]!.Entered, PrevSearch) = (0,
    1, 1, 1, GameListData, 0, "", "")

    // start the search the game list //
    StartSearch()
  }
}

// the game list toolbar action //
@MainActor func GameListAction(_ Selection:Int)
{
  // exit the list without loading selected game //
  if Selection == 0 {CloseListWindow(); ResetGameData(); return}

  // load the game from the list //
  LoadGameFromList("GameListTableView")
}

// refresh or load the game from the game list //
@MainActor func RefreshOrLoadFromList(_ Key:String, Entry:String, Load:Bool)
{
  // if need to load the selected game in the list //
  if Load {LoadGameFromList(Key); return}

  // return if the search string hasn't changed //
  if PrevSearch == GetString().FromTextField(Entry).lowercased() {return}

  // save the search string //
  PrevSearch = GetString().FromTextField(Entry).lowercased()

  // run on the main thread //
  Task {await MainActor.run
  {
    // condense the search list to values matching the search string //
    GCUI.Objs[Key]!.GameList = GetString().FromTextField(Entry).isEmpty ?
                    GameListData : GameListData.filter
                    {$0.lowercased().contains(GetString()
                    .FromTextField(Entry).lowercased())}

    // reset the selected row to be zero //
    GCUI.Objs[Key]!.Selection = 0

    // refresh the views //
    RefreshViews(["GameListTableView":false, "ListToolbar":GCUI.Objs[Key]!
            .GameList.isEmpty])
  }}
}

// load the selected game from the game list window //
@MainActor func LoadGameFromList(_ Key:String)
{
  // run on the main thread //
  Task {await MainActor.run
  {
    // reset the list toolbar selection to zero //
    GCUI.Objs["ListToolbar"]!.Selection = 0

    // close the game list window //
    CloseListWindow()

    // start a new game //
    StartNewGame("")

    // extract and display the game information from the game data list //
    ExtractGameData(DatabaseGetGameData(GameData, GameList, CInt(GCUI
            .Objs[Key]!.GameListIndex())), Tasks:"All")

    // store the autosave data //
    StoreAutoData("")

    // initialize the game clock //
    InitializeGameClock()
  }}
}

// save the current board layout for before go into the position editor //
@MainActor func SaveBoardLayout()
{
  // save the board layout //
  BPData.SaveLayout()

  // save the fen preference values //
  GCUI.Attributes.Inf.SaveValues(["FENPref"])
}

// reset the board layout to the original setup //
@MainActor func ResetBoardLayout()
{
  // reset the board layout //
```

```
      BPData.RestoreLayout()

      // restore the saved fen string //
      GCUI.Attributes.Inf.RestoreSavedValues(["FENPref"])

      // set moving piece to zero //
      MovingPiece.Reset()
   }

// play timed game action //
@MainActor func PlayTimedGameAction(_ Selection:Int)
{
   // switch over the selection //
   switch Selection
   {
      // exit the timed game window //
      case 0 : CloseSheet(TimedGameWindow)

      // play the timed game from the user selections //
      case 1 :

         // update the exit timed game toolbar visibility //
         GCUI.UpdateVisibility(["ExitGameToolbar"], Visible:true)

         // start a new game //
         StartNewGame("All")

         // set that currently playing a timed game and to start the clock //
         Inf.Update("VB", VBItems:["PlayingTimedGame":true, "StartClock":
                    true])

         // save the use eco book value //
         Inf.SaveValue("VBPref", Key:"UseEcoBook")

         // set that currently using opening book //
         Inf.TurnOnVB("VBPref", Key:"UseEcoBook")

         // update the opponent type //
         UpdateOpponent((TimedGameOpponent == 0) ? "GreenChess" :
                        ((TimedGameOpponent == 1) ? "Human" : "Stockfish"))

         // calculate the total game time //
         TotalGameTime = 9000 * (TimedGameTime + 1)

         // calc timed game offset from 6 o'clock pos and remaining times //
         (TimedGameOffset, RemainingWhiteTime, RemainingBlackTime) = (216000
         - TotalGameTime, TotalGameTime, TotalGameTime)

         // initialize the game clock //
         InitializeGameClock()

         // close the timed game window //
         CloseSheet(TimedGameWindow)

         // if user plays black against engine, start engine move //
         if (TimedGameColor == 1) && (Inf.Vals("HumanOpponent").VB
         {
            // start the engine move //
            StartEngineMove()
         }

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

// the enter fen string action //
@MainActor func EnterFENAction(_ Selection:Int)
{
   // close the fen window //
   CloseSheet(FENWindow)

   // if the fen string is not empty //
   if !FENString.isEmpty
   {
      // save the current board layout //
      SaveBoardLayout()

      // extract the entered fen string //
      GCUI.Attributes.ExtractEnteredFENString(FENString)

      // set the initial layout //
      SetInitialLayout()

      // send the fen to convert to the position editor //
      StartPosEditor()
   }

   // set entering fen to false //
   Inf.TurnOffVB("VB", Key:"EnteringFEN")
}

// update the position editor //
@MainActor func UpdatePosEditor(_ Key:String, Selection:Int)
{
   // switch over the key //
   switch Key
   {
      // update the white castling selection //
      case "WhiteCastling" : Inf.Update("VB", VBItems:["WhiteKingsideCastle":
                             (Selection == 1 || Selection == 3) ? true :
                             false, "WhiteQueensideCastle": (Selection == 2
                             || Selection == 3) ? true : false])

      // update the black castling selection //
      case "BlackCastling" : Inf.Update("VB", VBItems:["BlackKingsideCastle":
                             (Selection == 1 || Selection == 3) ? true :
                             false, "BlackQueensideCastle": (Selection == 2
                             || Selection == 3) ? true : false])

      // update the enpassant square selection //
      case "EPSquare" : GCUI.Attributes.Inf.Update("FENPref", VSItems:
                        ["FENEnpassantSquare":EnpassantSquares[Selection] !=
                        nil ? EnpassantSquares[Selection]! : "-"])

      // update the color to move selection //
      case "ColorToMove" : GCUI.Attributes.Inf.Update("FENPref", VSItems:
                           ["FENColor":(Selection == 0) ? "w" : "b"])

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

// update the timed game info //
@MainActor func UpdateTimedGameInfo(_ Key:String, Selection:Int)
{
   // set the timed game color //
   TimedGameColor = GCUI.Objs["TimedGameColor"]!.Selection

   // set the timed game opponent //
   TimedGameOpponent = GCUI.Objs["TimedGameOpponent"]!.Selection

   // set the timed game time //
   TimedGameTime = GCUI.Objs["TimedGameTime"]!.Selection
}

// the load position action //
@MainActor func LoadPositionAction(_ Selection:Int)
{
   // switch over the selection //
   switch Selection
   {
      // exit the load position window and close the load position window //
      case 0 : CloseSheet(LoadPositionWindow)

      // load the selected position //
      case 1 :

         // start a new game //
         StartNewGame("")

         // extract and display the game information from game data list //
         ExtractGameData(SavedPositionData[LoadPosIndex], Tasks:"All")

         // Initialize the game clock //
         InitializeGameClock()

         // store the autosave data //
         StoreAutoData("")

         // close the load position window //
         CloseSheet(LoadPositionWindow)

      // go left in the load position window //
      case 2 :

         // go left in the position images index //
         LoadPosIndex = max(LoadPosIndex - 1, 0)

         // set the load position image to the selected index //
         GCUI.Objs["LoadPosImage"]!.TheImage = PosImages[LoadPosIndex]

      // go right in the load position window //
      case 3 :

         // go right in the position images index //
         LoadPosIndex = min(PosImages.count - 1, LoadPosIndex + 1)

         // set the load position image to the selected index //
         GCUI.Objs["LoadPosImage"]!.TheImage = PosImages[LoadPosIndex]

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

// the game annotator action //
@MainActor func GameAnnotatorAction(_ Selection:Int)
{
   // switch over the selection //
   switch Selection
   {
      // exit the game annotator //
      case 0 :

         // set that no longer annotating the game //
         Inf.TurnOffVB("VB", Key:"AnnotatingGame")

         // close the game annotator window sheet //
         CloseSheet(GameAnnotatorWindow)

         // store the autosave data //
         StoreAutoData("")

         // refresh all the views //
         Refresh("All")

      // go left one move in the game annotator //
      case 1: GCUI.Objs["AnnotatorMove"]!.Val = max(0, GCUI
              .Objs["AnnotatorMove"]!.Val - 1 / Double((GCUI.Attributes.Inf
              .GetCount() <= 1) ? 1 : GCUI.Attributes.Inf.GetCount() - 1))

      // go right one move in the game annotator //
      case 2: GCUI.Objs["AnnotatorMove"]!.Val = min(1, GCUI
              .Objs["AnnotatorMove"]!.Val + 1 / Double((GCUI.Attributes.Inf
              .GetCount() <= 1) ? 1 : GCUI.Attributes.Inf.GetCount() - 1))

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

// the position editor action //
@MainActor func PositionAction(_ Selection:Int)
{
   // switch over the selection //
   switch Selection
   {
      // the exit button //
      case 0 :

         // refresh the game explorer toolbar //
         RefreshViews(["GameExpToolbar":Inf.Vals("ReviewingGame").VB])

         // exit the position editor //
         ExitPositionEditor()

      // save the edited position //
      case 1 : PositionEditorSave()

      // set the cursor to the piece currently being placed in the editor //
      case 2...14 : SetCursor(PosPieces[Selection])

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

// the game analysis action //
@MainActor func AnalysisAction(_ Selection:Int)
{
   // update the visibility of the game analysis toolbar //
   GCUI.UpdateVisibility(["AnalysisToolbar"], Visible:false)

   // exit the game anlysis or timed game mode //
   ExitAnalysisOrTimedGame()
}

// exit the the timed game action //
@MainActor func ExitGameAction(_ Selection:Int)
{
   // update the visibility of the exit timed game toolbar //
   GCUI.UpdateVisibility(["ExitGameToolbar"], Visible:false)

   // exit the game analysis or timed game mode //
   ExitAnalysisOrTimedGame()
}

// the board diagram action //
@MainActor func DiagramAction(_ Selection:Int)
{
   // switch over the selection //
   switch Selection
   {
      // exit the board diagram mode //
      case 0 :

         // refresh the game explorer toolbar //
         RefreshViews(["GameExpToolbar":Inf.Vals("ReviewingGame").VB])

         // update the visibility of the diagram toolbar //
         GCUI.UpdateVisibility(["DiagramToolbar"], Visible:false)

         // the start game explorer menu item action //
         StartGameExplorer()

      // take the board snapshot action //
      case 1 : BoardSnapshotAction()

      // refresh the board sketch //
      case 2 : Sketch.Refresh(true)

      // update the sketch path //
      case 3 : Sketch.UndoPath()

      // update the marker drawing color //
      case 4...7 :

         // set that currently diagramming //
         Inf.TurnOnVB("VB", Key:"Diagramming")

         // set the current diagram cursor //
         Sketch.UpdateColor(Selection + 1)

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

// the game gallery action //
@MainActor func GameGalleryAction(_ Selection:Int)
{
   // switch over the selection //
   switch Selection
   {
      // exit the game gallery //
      case 0 :

         // invalidate the game gallery review time //
         GalleryReviewTimer.invalidate()

         // erase the game gallery view //
         GCUI.GetGalleryView("GalleryView").EraseGallery()

         // set that no longer in the game gallery //
         Inf.TurnOffVB("VB", Key:"InGameGallery")

         // close the gallery window sheet //
         CloseSheet(GameGalleryWindow)

      // save the gallery snapshot //
      case 1 :

         // do the snapshot tasks //
         SnapshotTasks()

         // save the game gallery snapshot //
         SavePanel.beginSheetModal(for:GameGalleryWindow, completionHandler:
                   {Result ->Void in if Result == NSApplication.ModalResponse
                   .OK {try? GCUI.GetGalleryView("GalleryView")
                   .GallerySnapshot().representation(using:NSBitmapImageRep
                   .FileType.jpeg, properties:[:])!.write(to:
                   URL(fileURLWithPath:self.SavePanel.url!.path), options:
                   [.atomic])}; self.EndSheet()})

      // save the gallery movie //
      case 2 :

         // do the movie tasks //
         MovieTasks()

         // save the gallery movie //
         SavePanel.beginSheetModal(for:CurrentWindow(), completionHandler:
                   {Result -> Void in if Result == NSApplication
                   .ModalResponse.OK {GCUI.GetGalleryView("GalleryView")
                   .WriteMovie(self.SavePanel.url!)}; self.EndSheet()})

      // clear the gallery marker paths //
      case 3 : GCUI.GetGalleryView("GalleryView").Sketch.Refresh(true)

      // undo a marker pen path in the gallery //
      case 4 : GCUI.GetGalleryView("GalleryView").Sketch.UndoPath()

      // set the marker pen color //
      case 5...8 : GCUI.GetGalleryView("GalleryView")
                   .UpdateGalleryColor(Selection)

      // go left one frame in the game gallery //
      case 9 : GameGalleryShift(-1)

      // review the game gallery //
      case 10 : ReviewGallery()

      // go right one frame in the game gallery //
      case 11 : GameGalleryShift(1)

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

// the game explorer action //
@MainActor func GameExplorerAction(_ Selection:Int)
{
   // switch over the selection //
   switch Selection
   {
      // exit the game explorer //
      case 0 :

         // invalidate the game explorer review timer //
         GameExplorerTimer.invalidate()

         // set game explorer window open, exploring, reviewing to false //
         Inf.Update("VB", VBItems:["GameExplorerWindowOpen":false,
                    "ExploringGame":false, "ReviewingGame":false])

         // close the game explorer window sheet //
         CloseSheet(GameExplorerWindow)

         // refresh all the views //
         Refresh("All")

      // create a board snapshot //
      case 1 : BoardSnapshotAction()

      // create a movie //
      case 2 :

         // do the movie tasks //
         MovieTasks()

         // start the save panel query for saving a game in movie format //
         SavePanel.beginSheetModal(for:GameExplorerWindow, completionHandler:
                   {Result ->Void in if Result == NSApplication.ModalResponse
                   .OK {GCUI.FramesMovie.WriteMovie(self.Graphics
                   .InitLayer(), VidWidth:1320, VidHeight:1320, Path:self
                   .SavePanel.url!, Layered:false)}; self.EndSheet()})

      // enter the game analysis mode //
      case 3 :

         // update the game analysis toolbar visibility //
         GCUI.UpdateVisibility(["AnalysisToolbar"], Visible:true)

         // set that game explorer window closed and analysing game //
         Inf.Update("VB", VBItems:["GameExplorerWindowOpen":false,
                    "AnalysingGame":true])

         // close the game explorer window sheet //
         CloseSheet(GameExplorerWindow)

         // save the game subset //
         SaveGameSubset()

      // enter the position editor //
      case 4 :

         // set explorer window false, analysing, editing nonempty to true //
         Inf.Update("VB", VBItems:["GameExplorerWindowOpen":false,
                    "AnalysingGame":true, "EditingNonemptyPosition":true])

         // close the game explorer window sheet //
         CloseSheet(GameExplorerWindow)

         // save the game subset //
         SaveGameSubset()

         // enter the position editor //
         StartPosEditor()

      // enter the board diagram mode //
      case 5 :

         // set the current marker color selection to red //
         GCUI.Objs["DiagramToolbar"]!.Selection = 4

         // update the board diagram toolbar //
         GCUI.UpdateVisibility(["DiagramToolbar"], Visible:true)

         // update explorer window to not open and that diagramming //
         Inf.Update("VB", VBItems:["GameExplorerWindowOpen":false,
                    "Diagramming":true])

         // set the diagram cursor to red highlighter //
         Sketch.UpdateColor(5)

         // close the game explorer window sheet //
         CloseSheet(GameExplorerWindow)

         // save the game subset //
         SaveGameSubset()

         // refresh the sketch //
         Sketch.Refresh(true)

      // go left in the game explorer slider //
      case 6 : GameExplorerMoveShift(-1)

      // review the game //
      case 7 : ReviewGame()

      // go right in the game explorer slider //
      case 8 : GameExplorerMoveShift(1)

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

// refresh the views //
@MainActor func RefreshViews(_ Views:[String:Bool])
{
   // refresh the views on main thread //
   Task {await MainActor.run {GCUI.RefreshViews(Views)}}
}

// the database explorer action //
@MainActor func DatabaseExplorerAction(_ Selection:Int)
{
   // switch over the selection //
   switch Selection
   {
      // create a board snapshot //
      case 2 : BoardSnapshotAction(); return

      // if go left in game database explorer //
      case 3 : GetDatabaseExplorerData(-1); return

      // if go right in game database explorer //
      case 4 : GetDatabaseExplorerData(1); return

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

   // turn off that in the game database epxlorer //
   Inf.TurnOffVB("VB", Key:"InDatabaseExplorer")

   // close the database explorer sheet //
   CloseSheet(DatabaseExplorerWindow)

   // switch over the selection //
   switch Selection
   {
      // if exit window, reset the game data //
      case 0 : ResetGameData()

      // if loading a game, initialize the game clock and refresh  //
      case 1 : InitializeGameClock(); Refresh("All"); StoreAutoData("")

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

// enter the database explorer //
@MainActor @IBAction func DatabaseExplorerMenuItemAction(_:AnyObject)
{
   // set that currently in the database explorer //
   Inf.TurnOnVB("VB", Key:"InDatabaseExplorer")

   // refresh the views //
   RefreshViews(["DataExpGame":GameListData.isEmpty, "DataExpMove":
                GameListData.isEmpty, "DataExpToolbar":GameListData.isEmpty])

   // backup the game attributes //
   GCUI.Attributes.Inf.BackupValues(["FENPref", "TagsPref", "ListsPref"],
                                    Tag:"Moves")

   // reset the game database explorer slider value //
   DataGameExpVal = -1

   // get the data to display //
   GetDatabaseExplorerData(0)

   // update the database game explorer image //
   UpdateExplorerImage("DataExpMove")

   // open the database explorer window sheet //
   OpenSheet(DatabaseExplorerWindow)
}

// open the main window sheet //
@MainActor func OpenSheet(_ Sheet:NSWindow)
{
   // run on the main thread //
   Task {await MainActor.run
   {
      // open the window sheet //
      CurrentWindow().beginSheet(Sheet, completionHandler:nil)
   }}
}

// edit the current board position //
@MainActor @IBAction func CurrentPositionEditorMenuItemAction(_:AnyObject)
{
   // set that editing nonempty board position //
   Inf.TurnOnVB("VB", Key:"EditingNonemptyPosition")

   // start the position editor //
   StartPosEditor()
}

// create an initial board layout position for analysis //
@MainActor @IBAction func PositionEditorMenuItemAction(_:AnyObject)
{
   // start the position editor //
   StartPosEditor()
}

// start the position editor //
@MainActor func StartPosEditor()
{
   // set that currently in the position editor //
   Inf.TurnOnVB("VB", Key:"InPositionEditor")

   // set the position editor piece selection to white pawn //
   GCUI.Objs["PosEditorToolbar"]!.Selection = 3

   // refresh the position editor buttons //
   RefreshPosEditorButtons(true)

   // if not entering fen //
   if !Inf.Vals("EnteringFEN").VB
   {
      // save the current board layout //
      SaveBoardLayout()

      // if starting with an empty position //
      if !Inf.Vals("EditingNonemptyPosition").VB
      {
         // set the default fen string //
         GCUI.Attributes.SetFENDefaults()

         // assign a blank layout //
         BPData.InitializeLayout()
      }
      // starting from a current position //
      else
      {
         // editing the current position //
         GCUI.Attributes.EditingCurrentPosition(Board)
      }
   }

   // initialize the castles //
   InitializeCastles()

   // set the white castling picker selection //
   GCUI.Objs["WhiteCastling"]!.Selection = (Inf.Vals("WhiteKingsideCastle")
                                           .VB && Inf
                                           .Vals("WhiteQueensideCastle").VB)
                                           ? 3 : (Inf
                                           .Vals("WhiteKingsideCastle").VB ?
                                           1 : (Inf
                                           .Vals("WhiteQueensideCastle").VB
                                           ? 2 : 0))

   // set the black castling picker selection //
   GCUI.Objs["BlackCastling"]!.Selection = (Inf.Vals("BlackKingsideCastle")
                                           .VB && Inf
                                           .Vals("BlackQueensideCastle").VB)
                                           ? 3 : (Inf
                                           .Vals("BlackKingsideCastle").VB ?
                                           1 : (Inf
                                           .Vals("BlackQueensideCastle").VB
                                           ? 2 : 0))

   // set the en passant square picker selection //
   GCUI.Objs["EPSquare"]!.Selection = (EnpassantSelection[GCUI.Attributes.Inf
                                      .Vals("FENEnpassantSquare").VS] != nil)
                                      ? EnpassantSelection[GCUI.Attributes
                                      .Inf.Vals("FENEnpassantSquare").VS]! :
                                      0

   // set the color to move picker selection //
   GCUI.Objs["ColorToMove"]!.Selection = (GCUI.Attributes.Inf
                                         .Vals("FENColor").VS == "w") ? 0 : 1

   // clear the moving piece //
   MovingPiece.Reset()

   // lighten the table views //
   (GCUI.Objs["GameInfoTableView"]!.ObjView.alphaValue, GCUI
   .Objs["MoveListTableView"]!.ObjView.alphaValue) = (0.25, 0.25)

   // save edited position to false //
   Inf.Update("VB", VBItems:["WillSaveEditedPosition":false])

   // set the position cursor to a white pawn //
   SetCursor(1)

   // refresh the view and tables //
   Refresh("ViewTables")
}

// create the castling rights for the fen notation string //
@MainActor func CreateCastlingRights()
{
   // initialize the fen notation string castling rights //
   GCUI.Attributes.Inf.ClearVS("FENPref", Key:"FENCastlingRights")

   // if white has right to kingside castle and correct king and rook pos //
   if Inf.Vals("WhiteKingsideCastle").VB && BPData.RFE(4, 0, 6) && BPData
      .RFE(7, 0, 4)
   {
      // append to fen castling rights that white able to castle kingside //
      GCUI.Attributes.Inf.AddTo("FENPref", Key:"FENCastlingRights", AddOn:
                                "K")
   }

   // if white has right to queenside castle and correct king and rook pos //
   if Inf.Vals("WhiteQueensideCastle").VB && BPData.RFE(4, 0, 6) && BPData
      .RFE(0, 0, 4)
   {
      // append to fen castling rights that white able to castle queenside //
      GCUI.Attributes.Inf.AddTo("FENPref", Key:"FENCastlingRights", AddOn:
                                "Q")
   }

   // if black has right to kingside castle and correct king and rook pos //
   if Inf.Vals("BlackKingsideCastle").VB && BPData.RFE(4, 7, -6) && BPData
      .RFE(7, 7, -4)
   {
      // append to fen castling rights that black able to castle kingside //
      GCUI.Attributes.Inf.AddTo("FENPref", Key:"FENCastlingRights", AddOn:
                                "k")
   }

   // if black has right to queenside castle and correct king and rook pos //
   if Inf.Vals("BlackQueensideCastle").VB && BPData.RFE(4, 7, -6) && BPData
      .RFE(0, 7, -4)
   {
      // append to fen castling rights that black able to castle queenside //
      GCUI.Attributes.Inf.AddTo("FENPref", Key:"FENCastlingRights", AddOn:
                                "q")
   }

   // if have no fen castling rights, set that have no castling rights //
   if GCUI.Attributes.Inf.VSIsEmpty("FENCastlingRights")
   {
      // set fen castling rights to - to show no current castling rights //
      GCUI.Attributes.Inf.Update("FENPref", VSItems:["FENCastlingRights":
                                 "-"])
   }
}

// exit the position editor //
@MainActor func ExitPositionEditor()
{
   // if will be able to save the edited position //
   if Inf.Vals("WillSaveEditedPosition").VB
   {
      // save the fen layout from the edited board position //
      ConvertLayoutToFEN()

      // save the edited board layout //
      SaveBoardLayout()

      // start a new game //
      StartNewGame("")

      // reset the board layout to saved layout //
      ResetBoardLayout()

      // initalize the game clock //
      InitializeGameClock()

      // store the autosave data //
      StoreAutoData("")
   }
   // not able to save the edited board position //
   else
   {
      // reset board to the original layout //
      ResetBoardLayout()
   }

   // set entering fen and position editor values to false //
   Inf.Update("VB", VBItems:["InPositionEditor":false,
              "WillSaveEditedPosition":false, "EditingNonemptyPosition":false,
              "EnteringFEN":false])

   // refresh all the views //
   Refresh("All")

   // set table view alpha values to one //
   (GCUI.Objs["GameInfoTableView"]!.ObjView.alphaValue, GCUI
   .Objs["MoveListTableView"]!.ObjView.alphaValue) = (1, 1)

   // refresh the position editor buttons //
   RefreshPosEditorButtons(false)

   // if currently in game analysis mode //
   if Inf.Vals("AnalyzingGame").VB
   {
      // update the visibility of the game analysis toolbar //
      GCUI.UpdateVisibility(["AnalysisToolbar"], Visible:true)
   }
}
// the append game to existing pgn file menu item action //
@MainActor @IBAction func AppendGameMenuItemAction(_:AnyObject)
{
   // start the sheet //
   StartSheet()

   // create the append panel //
   AppendPanel = NSOpenPanel()

   // set the append panel to allow all file types //
   AppendPanel.allowsOtherFileTypes = true

   // set the append file panel prompt and message //
   (AppendPanel.prompt, AppendPanel.message) = ("Append PGN",
   "Append Current Game to PGN File")

   // run the append panel as a sheet on the main window //
   AppendPanel.beginSheetModal(for:CurrentWindow(), completionHandler:
               {Result -> Void in if Result == NSApplication.ModalResponse.OK
               {self.AppendGame(self.AppendPanel.url!.path)}; self
               .EndSheet()})
}

// the open game menu item action //
@MainActor @IBAction func OpenGameMenuItemAction(_:AnyObject)
{
   // start the sheet //
   StartSheet()

   // create an open panel //
   OpenPanel = NSOpenPanel()

   // set the open panel to allow all file types //
   OpenPanel.allowsOtherFileTypes = true

   // set the open file allowed prompt and message //
   (OpenPanel.prompt, OpenPanel.message) = ("Open PGN", "Open PGN Game File")

   // run the open panel as a sheet on the main window //
   OpenPanel.beginSheetModal(for:CurrentWindow(), completionHandler:{Result
             -> Void in if Result == NSApplication.ModalResponse.OK {self
             .OpenGame(self.OpenPanel.url!.path)}; self.EndSheet()})
}

// save the current game to pgn file menu item action //
@MainActor @IBAction func SaveGameMenuItemAction(_:AnyObject)
{
   // start the sheet //
   StartSheet()

   // create a save panel //
   SavePanel = NSSavePanel()

   // set the allowed content type, file name, prompt, and message //
   (SavePanel.allowedContentTypes, SavePanel.nameFieldStringValue, SavePanel
   .prompt, SavePanel.message) = ([UTType(filenameExtension:"pgn",
   conformingTo:.text)!], "GreenChessGame.pgn", "Save PGN",
   "Save Current Game to PGN File")

   // run the panel to save the current game in pgn format //
   SavePanel.beginSheetModal(for:CurrentWindow(), completionHandler:{Result
             -> Void in if Result == NSApplication.ModalResponse.OK {self
             .SaveGame(self.SavePanel.url!.path)}; self.EndSheet()})
}

// reset the game data //
@MainActor func ResetGameData()
{
   // restore the backup values for the specified tags //
   GCUI.Attributes.Inf.RestoreBackupValues(["FENPref", "TagsPref",
                                           "ListsPref"], Tag:"Moves")

   // refresh all the views //
   Refresh("All")

   // store the autosave data //
   StoreAutoData("")
}

// save the game subset //
@MainActor func SaveGameSubset()
{
   // save the game subset for the game explorer slider val //
   GCUI.Attributes.SaveSubset(Int(round(GCUI.Objs["GameExpMove"]!.Val *
                              Double(GCUI.Attributes.Inf.GetCount()))))

   // refresh all the views //
   Refresh("All")
}

// extract and display information from the game data object //
@MainActor func ExtractGameData(_ GameData:UnsafeMutableRawPointer, Tasks:
                               String)
{
   // process the game data //
   GCUI.Attributes.ProcessGameData(GameData)

   // remove the invalid moves and update the notation //
   RemoveInvalidMovesAndUpdateNotation("Moves")

   // refresh the tasks //
   Refresh(Tasks)
}

// close the game list window //
@MainActor func CloseListWindow()
{
   // close the game list window //
   CloseSheet(GameListWindow)

   // let the game list table view know to exit //
   GCUI.Objs["GameListTableView"]!.Command = "End"
}

// close the specified sheet //
@MainActor func CloseSheet(_ Sheet:NSWindow)
{
   // end the specified sheet attached to the main window //
   MainWindow.endSheet(Sheet)

   // close the specified sheet //
   Sheet.close()
}

// exit the preferences window //
@MainActor func ExitPreferencesWindow()
{
   // close the preferences window sheet //
   CloseSheet(PreferencesWindow)

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

   // save the move list font preferences //
   MoveListFont.SavePreferences()

   // save the game info font preferences //
   GameInfoFont.SavePreferences()

   // save the game list font preferences //
   GameListFont.SavePreferences()

   // save the board and piece composite preferences //
   GCUI.BPComp.SavePreferences()

   // save the frames movie preferences //
   GCUI.FramesMovie.SavePrefs()

   // set that no longer entering in preferences //
   Inf.TurnOffVB("VB", Key:"EnteringPreferences")

   // set the white and black time clocks to refresh all //
   (GCUI.GetClockView("WhiteClockView").RefreshAll, GCUI
   .GetClockView("BlackClockView").RefreshAll) = (true, true)

   // update the white time clock //
   GCUI.GetClockView("WhiteClockView").UpdateClock(ElapsedWhiteTime)

   // update the black time clock //
   GCUI.GetClockView("BlackClockView").UpdateClock(ElapsedBlackTime)

   // update the progress circle color //
   ProgressCircle.UpdateCircleColor(GCUI.GetColorView("WindowColorView")
                                    .GetCurrentHue())

   // refresh the board and tables or all //
   Refresh(Inf.Vals("InPositionEditor").VB ? "BoardTables" : "All")
}

// exit the enter game info window //
@MainActor func ExitInfoWindow(_ Val:Int)
{
   // close the game info sheet //
   CloseSheet(EnterInfoWindow)

   // refresh all //
   Refresh("All")
```

```
   // store the autosave data //
   StoreAutoData("")
}

// launch the gallery //
@MainActor func LaunchGallery(_ NumberOfLayers:Int, ApproxSize:CGFloat)
{
   // set that in the game gallery //
   Inf.TurnOnVB("VB", Key:"InGameGallery")

   // set toolbar selection, slider value, max value, frame, and layers //
   (GCUI.Objs["GalleryToolbar"]!.Selection, GCUI.Objs["GallerySlider"]!.Val,
   GCUI.Objs["GallerySlider"]!.MaxV, GCUI.GetGalleryView("GalleryView")
   .Layers, GCUI.GetGalleryView("GalleryView").FrameSize, GCUI
   .Objs["GallerySlider"]!.Place, GCUI.Objs["GallerySlider"]!.ObjView.frame)
   = (0, 0, 1, NumberOfLayers, MainWindow.frame.size.width /
   CGFloat(Int(MainWindow.frame.size.width / ApproxSize)), Placement(XP:15,
   YP:6, Width:MainWindow.frame.size.width - 30.0, Height:30), Placement(XP:
   15, YP:6, Width:MainWindow.frame.size.width - 30.0, Height:30).Frame)

   // set current size of game gallery window to fit size of main window //
   GameGalleryWindow.setContentSize(CGSize(width:MainWindow.frame.size.width,
                    height:GCUI.GetGalleryView("GalleryView").FrameSize *
                    CGFloat(GCUI.GetGalleryView("GalleryView").Layers) +
                    70))

   // set frame for gallery view //
   GCUI.GetGalleryView("GalleryView").frame = CGRect(x:0, y:40, width:
   MainWindow.frame.size.width, height:GCUI.GetGalleryView("GalleryView")
   .FrameSize * CGFloat(GCUI.GetGalleryView("GalleryView").Layers))

   // refresh the gallery toolbar //
   RefreshViews(["GalleryToolbar":GCUI.GetGalleryView("GalleryView").Inf
              .Vals("Reviewing").VB])

   // initialize the game gallery view //
   GCUI.GetGalleryView("GalleryView").Initialize()

   // refresh the sketch to the screen //
   GCUI.GetGalleryView("GalleryView").Sketch.Refresh(true)

   // open the game gallery window //
   OpenSheet(GameGalleryWindow)
}

// the game gallery grid menu item action //
@MainActor @IBAction func GalleryGridMenuItemAction(_:AnyObject)
{
   // launch the gallery for the grid view //
   LaunchGallery(Inf.Vals("NumberOfGridRows").VI, ApproxSize:GridSizes[Inf
              .Vals("NumberOfGridRows").VI - 1])
}

// the game gallery menu item action //
@MainActor @IBAction func GameGalleryMenuItemAction(_:AnyObject)
{
   // launch the gallery for one row //
   LaunchGallery(1, ApproxSize:GridSizes[0])
}

// update the game list font //
@MainActor func UpdateGameListFont(_ Key:String, Selection:Int)
{
   // update the game list font //
   GameListFont.UpdateFont(Key, Selection:Selection)
}

// update the move list font //
@MainActor func UpdateMoveListFont(_ Key:String, Selection:Int)
{
   // update the move list font //
   MoveListFont.UpdateFont(Key, Selection:Selection)

   // refresh the move list font type //
   RefreshMoveListFont()
}

// update the game info font //
@MainActor func UpdateGameInfoFont(_ Key:String, Selection:Int)
{
   // update the game info font //
   GameInfoFont.UpdateFont(Key, Selection:Selection)

   // refresh the game info list font type //
   RefreshGameInfoFont()
}

// update the board labels font //
@MainActor func UpdateBoardLabelsFont(_ Key:String, Selection:Int)
{
   // update the board labels font //
   GCUI.BPComp.BoardLabelsFont.UpdateFont(Key, Selection:Selection)

   // refresh the board //
   Refresh("Board")
}

// set the game clock background color //
@MainActor func SetGameClockBackgroundColor()
{
   // set game clock background color from the clock background color view //
   Inf.Update("VIPref", VIItems:["GameClockBackgroundHue":GCUI
              .GetColorView("ClockSurfaceColorView").GetHueValue()])

   // loop over all the clocks //
   for TheClock in [GCUI.GetClockView("WhiteClockView"), GCUI
       .GetClockView("BlackClockView"), GCUI.GetClockView("ClockView")]
   {
      // update the clock background color //
      TheClock.UpdateClockBackgroundColor(GCUI
              .GetColorView("ClockSurfaceColorView").GetCurrentHue())
   }
}

// set the game clock color //
@MainActor func SetGameClockColor()
{
   // set the game clock color from the clock color view //
   Inf.Update("VIPref", VIItems:["GameClockHue":GCUI
             .GetColorView("ClockColorView").GetHueValue()])

   // loop over all the clocks //
   for TheClock in [GCUI.GetClockView("WhiteClockView"), GCUI
       .GetClockView("BlackClockView"), GCUI.GetClockView("ClockView")]
   {
      // update the clock border color //
      TheClock.UpdateClockBorderColor(GCUI.GetColorView("ClockColorView")
              .GetCurrentHue())
   }
}

// set the number of gallery grid rows //
@MainActor func SetNumberOfGridRows(_ Key:String, Selection:Int)
{
   // set the number of gallery grid rows //
   Inf.Update("VIPref", VIItems:[Key:Selection + 2])
}

// the preferences window action //
@MainActor func PreferencesAction(_ Selection:Int)
{
   // exit the preferences window is click on close button //
   if Selection == 0 {ExitPreferencesWindow(); return}

   // update the current preferences selection //
   Inf.Update("VIPref", VIItems:["PreferencesSelection":Selection])

   // update the board previews //
   UpdatePreviews(true, Board:true, Diagram:true, Texture:true)

   // update the visible toolbar selection buttons //
   GCUI.UpdateVisibility(GCUI.PrefsBar[Selection].3)
}

// the preferences menu item action //
@MainActor @IBAction func PreferencesMenuItemAction(_:AnyObject)
{
   // set that currently entering in preferences //
   Inf.TurnOnVB("VB", Key:"EnteringPreferences")

   // update the current preferences selection //
   PreferencesAction(Inf.Vals("PreferencesSelection").VI)

   // refresh the preferences toolbar //
   RefreshViews(["PrefsToolbar":false])

   // begin the preferences sheet //
   OpenSheet(PreferencesWindow)
}

// search the game list menu item action //
@MainActor @IBAction func SearchGameListMenuItemAction(_:AnyObject)
{
   // start the game list search //
   StartSearch()
}

// start the game list search //
@MainActor func StartSearch()
{
   // update the game list view //
   (GCUI.Objs["GameListTableView"]!.FontType, GCUI.Objs["GameListTableView"]!
   .Command, GCUI.Objs["GameListTableView"]!.TextColor) = (GameListFont
   .FontType, "Start", Color(GCUI.GetColorView("GameListColorView")
   .GetCurrentHue()))

   // refresh the views //
   RefreshViews(["GameListTableView":GameListData.isEmpty, "ListToolbar":
              GCUI.Objs["GameListTableView"]!.GameList.isEmpty])

   // backup the game attributes //
   GCUI.Attributes.Inf.BackupValues(["FENPref", "TagsPref", "ListsPref"],
                                   Tags:"Moves")

   // open the game list window sheet //
   OpenSheet(GameListWindow)
}

// the load position menu item action //
@MainActor @IBAction func LoadPositionMenuItemAction(_:AnyObject)
{
   // if the saved position data is empty //
   if SavedPositionData.isEmpty
   {
      // set query load position to true //
      Inf.TurnOnVB("VB", Key:"QueryLoadPosition")

      // let user know that there are no saved position //
      MakeAlert(["Okay"], Query:"No Saved Boards")

      // exit the function //
      return
   }

   // refresh the load position toolbar //
   RefreshViews(["LoadPosToolbar":true])

   // save the current game //
   GCUI.Attributes.SaveSubset(GCUI.Attributes.Inf.GetCount())

   // clear the position images //
   PosImages.removeAll()

   // loop over the saved position data //
   for PositionData in SavedPositionData
   {
      // extract the position data //
      ExtractGameData(PositionData, Tasks:"Replay")

      // append the corresponding position image for the data //
      PosImages.append(BoardImage(0, Size:1310, Update:false))
   }

   // reset the game data to the original values //
   ResetGameData()

   // update the load position image //
   GCUI.Objs["LoadPosImage"]!.TheImage = PosImages[LoadPosIndex]

   // begin the load position window sheet //
   OpenSheet(LoadPositionWindow)
}

// the save position menu item action //
@MainActor @IBAction func SavePositionMenuItemAction(_:AnyObject)
{
   // set query save position to true //
   Inf.TurnOnVB("VB", Key:"QuerySavePosition")

   // convert the current game to game data //
   ConvertToGameData()

   // create a new element at the end of the saved positions data array //
   SavedPositionData.append(DataNew())

   // copy the game data to the end of the saved positions data array //
   DataCopy(SavedPositionData[SavedPositionData.count - 1], GameData)

   // store the autosave data to the saved position slot //
   StoreAutoData(Inf.IntStr("SavedPositionSlot"))

   // if autosaving game data //
   if Inf.Vals("AutosaveGame").VB
   {
      // increment the saved position slot index //
      Inf.Update("VIPref", VIItems:["SavedPositionSlot":Inf
                .Vals("SavedPositionSlot").VI % 25 + 1])

      // save the saved position slot index to preferences //
      Inf.SavePrefs([], VITags:["VIPref"], VSTags:[])
   }

   // make an alert that the game data was just saved //
   MakeAlert(["Okay"], Query:"Saved Board")
}

// the about window menu item action //
@MainActor @IBAction func AboutMenuItemAction(_:AnyObject)
{
   // update the window open status //
   UpdateWindowOpenStatus(true)

   // start the sheet //
   StartSheet()

   // refresh the about toolbar view //
   RefreshViews(["AboutToolbar":false])

   // open the about window //
   OpenSheet(AboutWindow)
}

// the enter game info menu item action //
@MainActor @IBAction func EnterGameInfoMenuItemAction(_:AnyObject)
{
   // update the pgn tags text fields //
   GCUI.Attributes.UpdateTOUI()

   // refresh the tags toolbar //
   RefreshViews(["TagsToolbar":false])

   // open the enter game info window as a sheet on the main window //
   OpenSheet(EnterInfoWindow)
}

// remove the timed game offset //
@MainActor func RemoveTimedGameOffset()
{
   // remove the timed game offset from the elapsed times //
   (ElapsedWhiteTime, ElapsedBlackTime) = (ElapsedWhiteTime -
   TimedGameOffset, ElapsedBlackTime - TimedGameOffset)

   // reset the timed game offset to zero //
   TimedGameOffset = 0

   // reset the eco book status to the saved value //
   Inf.RestoreSavedValue("VBPref", Key:"UseEcoBook")

   // set query timed game over and playing timed game to false //
   Inf.Update("VB", VBItems:["QueryTimedGameOver":false, "PlayingTimedGame":
              false])

   // set clocks to refresh all //
   (GCUI.GetClockView("WhiteClockView").RefreshAll, GCUI
   .GetClockView("BlackClockView").RefreshAll) = (true, true)
}

// the enter fen menu item action //
@MainActor @IBAction func EnterFENMenuItemAction(_:AnyObject)
{
   // refresh the timed game toolbar //
   RefreshViews(["EnterFENStringToolbar":true])

   // set that currently entering fen to convert //
   Inf.TurnOnVB("VB", Key:"EnteringFEN")

   // start the fen window //
   OpenSheet(FENWindow)
}

// the timed game menu item action //
@MainActor @IBAction func TimedGameMenuItemAction(_:AnyObject)
{
   // refresh the timed game toolbar //
   RefreshViews(["PlayTimedGameToolbar":true])

   // start the timed game window //
   OpenSheet(TimedGameWindow)
}

// function to run the game annotator menu item action //
@MainActor @IBAction func GameAnnotatorMenuItemAction(_:AnyObject)
{
   // set that currently annotating a game //
   Inf.TurnOnVB("VB", Key:"AnnotatingGame")

   // refresh the annotator toolbar //
   RefreshViews(["AnnotatorToolbar":true])

   // create the image frames for the annotator //
   GCUI.FramesMovie.CreateImageFrames(650)

   // set the game annotator move maximum value to 1 //
   GCUI.Objs["AnnotatorMove"]!.MaxV = 1

   // set the game annotator board images //
   GCUI.Objs["AnnotatorMoveView"]!.TheImages = [BoardImage(0, Size:650,
                                               Update:false)] + GCUI
                                               .FramesMovie.ImageFrames

   // if there are no moves in the game //
   if GCUI.Attributes.NoMoves()
   {
      // set that there are no moves to annotate //
      GCUI.Objs["AnnotatorMoveView"]!.PGNMove = "There are currently no " +
                                               "moves to annotate."

      // disable the text editor view //
      GCUI.Objs["AnnotatorMoveView"]!.DisableView = true
   }
   // there are moves in the game //
   else
   {
      // set the text editor to be visible //
      GCUI.Objs["AnnotatorMoveView"]!.DisableView = false
   }

   // update the game annotator slider //
   _ = UpdateAnnotatorSlider("AnnotatorMove", Value:GCUI
              .Objs["AnnotatorMove"]!.Val)

   // open the game annotator window //
   OpenSheet(GameAnnotatorWindow)
}

// update the entered fen string //
@MainActor func UpdateEnteredFEN(_ Key:String, Entry:String)
{
   // update the entered fen string //
   FENString = GetString().FromTextField(Entry)
}

// exit the current analysis or timed game mode //
@MainActor func ExitAnalysisOrTimedGame()
{
   // refresh the game explorer toolbar //
   RefreshViews(["GameExpToolbar":Inf.Vals("ReviewingGame").VB])

   // set stop engines to true and playing engine match move to false //
   Inf.Update("VB", VBItems:["StopEngines":true, "PlayingEngineMatch":false])

   // tell the uci engine to stop //
   Stockfish.WriteStockfishToStop()

   // make the chess engine stop //
   EngineMakeEngineStop(Engine)

   // remove all the moving piece animations //
   MovingPiece.removeAllAnimations()

   // set engine thinking to false //
   Inf.TurnOffVB("VB", Key:"EngineThinking")

   // set the go stop menu item title to engine //
   GoStopMenuItem.title = "Engine"

   // stop the progress circle animation //
   ProgressCircle.StopProgressAnimation()

   // remove the animation layer from the super layer //
   MovingPiece.HideAnimation()

   // reset the moving piece //
   MovingPiece.Reset()

   // if the explorer menu item action //
   if Inf.Vals("AnalyzingGame").VB {StartGameExplorer()}

   // if playing a timed game, remove the timed game offset //
   if Inf.Vals("PlayingTimedGame").VB {RemoveTimedGameOffset()}
}

// the game explorer menu item action //
@MainActor @IBAction func GameExplorerMenuItemAction(_:AnyObject)
{
   // refresh the game explorer toolbar //
   RefreshViews(["GameExpToolbar":Inf.Vals("ReviewingGame").VB])

   // start the game explorer //
   StartGameExplorer()
}

// start the game explorer //
@MainActor func StartGameExplorer()
{
   // set that the game explorer window is currently open //
   Inf.TurnOnVB("VB", Key:"GameExplorerWindowOpen")

   // set the game explorer maximum value //
   GCUI.Objs["GameExpMove"]!.MaxV = 1

   // if currently analyzing the game, reset the game data //
   if Inf.Vals("AnalyzingGame").VB
   {
      // turn off that currently analyzing the game //
      Inf.TurnOffVB("VB", Key:"AnalyzingGame")

      // reset the game data //
      ResetGameData()
   }

   // if currently diagramming, reset the game data //
   if Inf.Vals("Diagramming").VB
   {
      // turn off that currently diagramming //
      Inf.TurnOffVB("VB", Key:"Diagramming")

      // reset the game data //
      ResetGameData()
   }

   // set that currently exploring the game //
   Inf.TurnOnVB("VB", Key:"ExploringGame")

   // initialize the transposition table //
   EngineInitializeTransTable(Engine)

   // update the game explorer image //
   UpdateExplorerImage("GameExpMove")

   // open the game explorer window as a sheet on the main window //
   OpenSheet(GameExplorerWindow)
}

// process the entered annotation //
@MainActor func ProcessAnnotation(_ Key:String, Entry:String)
{
   // get the annotation string //
   AnnotationString = GetString().FromTextField(Entry)

   // if the annotation string is empty, return //
   if AnnotationString.isEmpty {return}

   // if number of left parentheses not equal number of right parentheses //
   if AnnotationString.components(separatedBy:"(").count != AnnotationString
      .components(separatedBy:")").count
   {
      // remove all the left and right parentheses from move annotation //
      AnnotationString = (AnnotationString.replacingOccurrences(of:"(",
                          with:"")).replacingOccurrences(of:")", with:"")
   }

   // remove all the left and right brackets //
   AnnotationString = (AnnotationString.replacingOccurrences(of:"{", with:
                       "")).replacingOccurrences(of:"}", with:"")

   // add brackets to encompass the entire move annotation string //
   AnnotationString = "{\(GetString().FromTextField(AnnotationString))}"

   // save the updated annotation //
   GCUI.Attributes.Inf.Update("Moves", VLSItems:[String(GCUI.Objs[Key]!
                              .Selection):AnnotationString])
}

// get the database explorer game data //
@MainActor func GetDatabaseExplorerData(_ Shift:Int)
{
   // if have not read in a game database yet, return //
   if GameListData.isEmpty {return}

   // set the database game explorer slider value if shifting //
   if (GameListData.count > 1) && (Shift != 0)
   {
      // update the game database explorer slider value //
      GCUI.Objs["DataExpGame"]!.Val = max(min(GCUI.Objs["DataExpGame"]!
                                      .Val + Double(Shift) / Double(self
                                      .GameListData.count - 1), 1), 0)
   }

   // if moved to a different part of the database game explorer slider //
   if DataGameExpVal != GCUI.Objs["DataExpGame"]!.Val
   {
      // extract and display game information from database game object //
      ExtractGameData(DatabaseGetGameData(Database, GameList, CInt(round(GCUI
                      .Objs["DataExpGame"]!.Val * Double(GameListData.count -
                      1)))), Tasks:"Replay")

      // update the saved database game explorer slider value //
      DataGameExpVal = GCUI.Objs["DataExpGame"]!.Val

      // if currently in the database explorer //
      if Inf.Vals("InDatabaseExplorer").VB
      {
         // refresh the database game move explorer slider view //
         RefreshViews(["DataExpMove":true])
      }
   }
}

// shift the game gallery slider position //
@MainActor func GameGalleryShift(_ Shift:Int)
{
   // if at the end of the game gallery slider and reviewing the gallery //
   if (GCUI.Objs["GallerySlider"]!.Val == 1) && GCUI
      .GetGalleryView("GalleryView").Inf.Vals("Reviewing").VB
   {
      // set the game gallery slider back to the starting point //
      GCUI.Objs["GallerySlider"]!.Val = 0
   }
   // else if the max gallery position is greater than zero //
   else if GCUI.FramesMovie.MaxPosition > 0
   {
      // update the game gallery slider position //
      GCUI.Objs["GallerySlider"]!.Val = max(min(GCUI.Objs["GallerySlider"]!
                                        .Val + Double(Shift) / Double(GCUI
                                        .FramesMovie.MaxPosition), 1), 0)
   }
}

// shift the game explorer slider move value by one //
@MainActor func GameExplorerMoveShift(_ Shift:Int)
{
   // if at the end of the game explorer slider and reviewing the game //
   if (GCUI.Objs["GameExpMove"]!.Val == 1) && Inf.Vals("ReviewingGame").VB
   {
      // set the game explorer slider value back to the beginning //
      GCUI.Objs["GameExpMove"]!.Val = 0
   }
   // if there are moves in the game //
   else if GCUI.Attributes.Inf.GetCount() > 0
   {
      // update the game explorer slider value //
      GCUI.Objs["GameExpMove"]!.Val = max(min(GCUI.Objs["GameExpMove"]!.Val +
                                      Double(Shift) / Double(GCUI.Attributes
                                      .Inf.GetCount()), 1), 0)
   }
}

// update the explorer image //
@MainActor func UpdateExplorerImage(_ Key:String)
{
   // process the current pgn move //
   ProcessPGNMove(Int(round(GCUI.Objs[Key]!.Val * Double(GCUI.Attributes.Inf
                  .GetCount()))) - 1)

   // run on the main thread //
   Task {await MainActor.run
   {
      // update the explorer image //
      GCUI.Objs[Key + "Image"]!.TheImage = BoardImage(Int(round(GCUI
                                           .Objs[Key]!.Val * Double(GCUI
                                           .Attributes.Inf.GetCount()))) *
                                           ((GameListData.isEmpty && Inf
                                           .Vals("InDatabaseExplorer").VB) ?
                                           0 : 1)), Size:GCUI.Objs[Key]!
                                           .Place.Width * 2, Update:false)
   }}
}

// update the clock background visibility //
@MainActor func UpdateClockBackgroundVisibility(_ Key:String, Value:Double)
                                                -> String
{
   // update the clock background visibility //
   Inf.Update("VIPref", VIItems:[Key:Int(Value)])

   // update the clock background visibility //
   UpdateClockVisibilityForBackground()

   // return the clock background visibility //
   return "Clock background" + Inf.Perc(Key)
}

// update the clock background visibility //
@MainActor func UpdateClockVisibilityForBackground()
{
   // loop over all the clocks //
   for TheClock in [GCUI.GetClockView("WhiteClockView"), GCUI
       .GetClockView("BlackClockView"), GCUI.GetClockView("ClockView")]
   {
      // update the clock background visibility //
      TheClock.UpdateClockBackgroundOpacity(Inf
              .Vals("ClockBackgroundVisibility").VI)
   }
}

// update the window texture opacity //
@MainActor func UpdateWindowTextureOpacity(_ Key:String, Value:Double) ->
                                          String
{
   // get window texture opacity from window texture opacity slider //
   Inf.Update("VIPref", VIItems:[Key:Int(Value)])

   // run on the main thread to set the window background color //
   Task {await MainActor.run {SetWindowBackgroundColor(false)}}

   // return the current window texture opacity //
   return "Window texture" + Inf.Perc(Key)
}

// update the current database explorer game //
@MainActor func UpdateDataExpGame(_ Key:String, Value:Double) -> String
{
   // get the database explorer data //
   GetDatabaseExplorerData(0)

   // return if no database or the game the description //
   return GameListData.isEmpty ? "No Database Loaded" :
          GameListData[Int(round(Value * Double(GameListData.count - 1)))]
}

// update the current database game explorer move //
@MainActor func UpdateDataExpMove(_ Key:String, Value:Double) -> String
{
   // update the game database explorer image //
   UpdateExplorerImage(Key)

   // return the move description //
   return (GameListData.isEmpty || GCUI.Attributes.NoMoves()) ? "No Moves" :
          (PGNMove.isEmpty ? "Beginning Position" : PGNMove)
}

// update the annotator slider //
@MainActor func UpdateAnnotatorSlider(_ Key:String, Value:Double) -> String
{
   // return empty string if there are no moves //
   if GCUI.Attributes.NoMoves() {return ""}

   // run on the main thread //
   Task {await MainActor.run
   {
      // update the selection //
      GCUI.Objs[Key + "View"]!.Selection = max(0, Int(round(Double(GCUI
                                           .Attributes.Inf.GetCount() - 1)
                                           * Value)))

      // process the annotation //
      ProcessAnnotation(Key + "View", Entry:GCUI.Attributes.Inf.Vals(GCUI
                        .Objs[Key + "View"]!.Selection).VLS)

      // update the entered annotation //
      GCUI.Objs[Key + "View"]!.Entered = AnnotationString

      // process the pgn move for the specified move location //
      ProcessPGNMove(GCUI.Objs[Key + "View"]!.Selection)

      // update the current pgn move //
      GCUI.Objs[Key + "View"]!.PGNMove = PGNMove
   }}

   // return an empty string //
   return ""
}

// update the game explorer move //
@MainActor func UpdateGameExpMove(_ Key:String, Value:Double) -> String
{
   // update the explorer image on the main thread //
   UpdateExplorerImage(Key)

   // return the move description //
   return GCUI.Attributes.NoMoves() ? "No Moves" : (PGNMove.isEmpty ?
          "Beginning Position" : PGNMove)
}

// update the board animation rate //
@MainActor func UpdateAnimationRate(_ Key:String, Value:Double) -> String
{
   // update current board animation rate from board animation rate slider //
   Inf.Update("VIPref", VIItems:[Key:Int(Value)])

   // return the board animation rate //
   return "Board animation speed is " + Inf.IntStr(Key)
}

// update the gallery slider //
@MainActor func UpdateGallerySlider(_ Key:String, Value:Double) -> String
{
   // run on the main thread //
   Task {await MainActor.run
   {
      // if currently in the game gallery //
      if Inf.Vals("InGameGallery").VB
      {
         // update the gallery game view //
         GCUI.GetGalleryView("GalleryView")
            .UpdateCurrentPosition(Int(round(Value * Double(GCUI
                                   .FramesMovie.MaxPosition))))
      }
   }}

   // return an empty string //
   return ""
}

// update the gallery animation rate //
@MainActor func UpdateGalleryAnimationRate(_ Key:String, Value:Double) ->
                                          String
{
   // update current gallery animation rate from the animation rate slider //
   Inf.Update("VIPref", VIItems:[Key:Int(Value)])

   // display the gallery animation rate //
   return "Gallery animation speed is " + Inf.IntStr(Key)
}

// update the illustration size //
@MainActor func UpdateIllustrationSize(_ Key:String, Value:Double) -> String
{
   // get the illustration size from the illustration size slider //
   Inf.Update("VIPref", VIItems:[Key:Int(Value)])

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

   // return the diagram size //
   return "Diagram size is " + Inf.IntStr(Key)
}

// update the uci engine difficulty //
@MainActor func UpdateUCIEngineDifficulty(_ Key:String, Value:Double) ->
                                         String
{
   // update uci engine difficulty from uci engine difficulty slider //
   Inf.Update("VIPref", VIItems:[Key:Int(Value)])

   // return the uci engine difficulty //
   return "Stockfish Engine Difficulty is " + Inf.IntStr(Key)
}

// update the uci engine hash //
@MainActor func UpdateUCIEngineHash(_ Key:String, Value:Double) -> String
{
   // update the current uci engine hash from the uci engine hash slider //
   Inf.Update("VIPref", VIItems:[Key:Int(Value)])

   // return the uci engine hash //
   return "Stockfish Engine Hash is " + Inf.IntStr(Key) + " MB"
}

// update the game tree search time //
@MainActor func UpdateSearchTime(_ Key:String, Value:Double) -> String
{
   // update the current game tree search time from the search time slider //
   Inf.Update("VIPref", VIItems:[Key:Int(Value)])

   // return the current search time //
   return "Engine search time is " + Inf.IntStr(Key) + " seconds"
}

// make the computer move or stop button //
@MainActor @IBAction func GoStopMenuItemAction(_:AnyObject)
{
   // run on the main thread //
   Task {await MainActor.run
   {
      // set that want to start the clock //
      Inf.TurnOnVB("VB", Key:"StartClock")

      // if playing engine match //
      if Inf.Vals("PlayingEngineMatch").VB
      {
         // set that no longer playing an engine match //
         Inf.TurnOffVB("VB", Key:"PlayingEngineMatch")
      }

      // if the engine is currently thinking //
      if Inf.Vals("EngineThinking").VB
      {
         // if playing against uci engine //
         if Inf.Vals("UCIEngineOpponent").VB
         {
            // tell the stockfish engine to stop //
            Stockfish.WriteStockfishToStop()
         }

         // if not playing against uci engine, make built-in engine stop //
         if !Inf.Vals("UCIEngineOpponent").VB {EngineMakeEngineStop(Engine)}
      }
      // if the engine is currently not thinking //
      else
      {
         // start an engine move //
         StartEngineMove()
      }
   }}
}

// perform a move of a piece on the board //
@MainActor func PerformEngineMove()
{
   // stop the progress circle animation //
   ProgressCircle.StopProgressAnimation()

   // set that there is no moving piece to currently animate //
   MovingPiece.Reset()

   // convert the uci move to start and end squares //
   BPData.UpdateStartAndEndSquareForMove(MovingPiece.UCIMove, Dims:Dims,
          Rotated:GCUI.BPComp.Inf.Vals("RotateBoard").VB)

   // animate the piece moving from the start square to the end square //
   RunCoreAnimation()
}

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

// set whether opponent is a human, greenchess engine, or uci chess engine //
@MainActor func UpdateOpponent(_ Opponent:String)
{
   // set whether playing against human, greenchess, or uci engine //
   Inf.Update("VB", VBItems:["HumanOpponent":Opponent == "Human",
              "EngineOpponent":Opponent == "GreenChess", "UCIEngineOpponent":
              Opponent == "Stockfish"])

   // update the opponent states //
   (HumanOpponentMenuItem.state, EngineOpponentMenuItem.state,
   UCIEngineOpponentMenuItem.state) = (UpdatedState(Inf.Vals("HumanOpponent")
   .VB), UpdatedState(Inf.Vals("EngineOpponent").VB), UpdatedState(Inf
   .Vals("UCIEngineOpponent").VB))

   // update the opponent value //
   Inf.Update("VSPref", VSItems:["Opponent":Opponent])

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

// update the pgn label pieces to symbol representations //
@MainActor func ProcessPGNMove(_ Index:Int)
{
   // get the save pgn notation //
   PGNLabel = GCUI.Attributes.Inf.Vals(Index).VNS

   // return if the pgn label is empty //
   if PGNLabel.isEmpty {PGNMove = ""; return}

   // if currently a white move //
   if ((Index + GCUI.Attributes.MoveOffset()) % 2) == 0
   {
      // for all the elements in the white piece symbols dictionary //
      for (Code, Symbol) in WhiteDict
      {
         // replace each code occurrence with the corresponding symbol //
         PGNLabel = PGNLabel.replacingOccurrences(of:Code, with:Symbol)
      }

      // if 1st letter of label is file and pawn move, add white pawn symb //
      if SquareLabel.IsFile(PGNLabel.first) {PGNLabel = "&™¥\(PGNLabel)"}
   }
   // currently a black move //
   else
   {
      // for all the elements in the black piece symbols dictionary //
      for (Code, Symbol) in BlackDict
      {
         // replace each code occurrence with the corresponding symbol //
         PGNLabel = PGNLabel.replacingOccurrences(of:Code, with:Symbol)
      }

      // if 1st letter of label is file and pawn move, add black pawn symb //
      if SquareLabel.IsFile(PGNLabel.first) {PGNLabel = "&™¥\(PGNLabel)"}
   }

   // set the pgn move number //
   PGNMove = String(format:"%d.", (Index + GCUI.Attributes.MoveOffset()) / 2
                    + GCUI.Attributes.FENStart())

   // if black is moving, add .. to pgn move for black //
   if ((Index + GCUI.Attributes.MoveOffset()) % 2) == 1 {PGNMove += ".."}

   // add the pgn label to the pgn move //
   PGNMove += " \(PGNLabel)"
}

// print the game information to the gui //
@MainActor func RefreshTableViews()
{
   // create the frames diagram list //
   GCUI.FramesMovie.CreateImageFrames(Inf.VF("IllustrationSize") * 2)

   // update the move list font //
   MoveListFont.UpdateFont("", Selection:0)

   // update the game info font //
   GameInfoFont.UpdateFont("", Selection:0)

   // clear the pgn moves list //
   PGNMoveList.removeAll()

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

   // set the current game state //
```

```
   GameState = (BoardIsCheckmate(Board) && BoardIsWhite(Board)) ? 1 :
               ((BoardIsCheckmate(Board) && BoardIsBlack(Board)) ? 2 :
               (BoardIsStalemate(Board) ? 3 :
               (BoardIsThreefoldRepetition(Board) ? 4 :
               (BoardInsufficientMaterialToMate(Board) ? 7 : 0))))

   // if the current game state is zero //
   if GameState == 0
   {
      // append the current move number to the game info //
      GameInfoList.append(String(format:"Move %d", (GCUI.Attributes.Inf
                          .GetCount() + GCUI.Attributes.MoveOffset()) / 2 +
                          GCUI.Attributes.FENStart()))

      // if white is currently to move //
      GameInfoList[0] += BoardIsWhite(Board) ? " and White to move" :
                          " and Black to move"

      // if in check and to move next, append out of check //
      if BoardIsInCheck(Board) {GameInfoList[0] += " out of check"}
   }

   // if the current game state exists //
   if GameOver[GameState] != nil {GameInfoList.append(GameOver[GameState]!)}

   // add the game information list from the game attributes //
   GameInfoList += GCUI.Attributes.GetInfoList()

   // if want to show the stockfish analysis //
   if Inf.Vals("ShowStockfishAnalysis").VB && !StockfishPV.isEmpty
   {
      // process the stockfish analysis string //
      GCUI.Attributes.ProcessAnalysis(StockfishPV);

      // reset the stockfish analysis string to empty //
      StockfishPV = ""

      // remove the invalid analysis moves and update the pgn notation //
      RemoveInvalidMovesAndUpdateNotation("Analysis")
   }

   // loop over move list including the analysis moves //
   for Index in 0..<GCUI.Attributes.Inf.AnalysisCount()
   {
      // process the pgn move //
      ProcessPGNMove(Index)

      // if not in the stockfish analysis move list //
      if Index < GCUI.Attributes.Inf.GetCount()
      {
         // get the current move annotation string //
         AnnotationString = GCUI.Attributes.Inf.Vals(Index).VLS

         // convert all the numeric annotation glyphs //
         for GlyphIndex in 0..<Glyphs.count
         {
            // replace the code occurrences with the corresponding symbols //
            AnnotationString = AnnotationString.replacingOccurrences(of:
                               Glyphs[GlyphIndex], with:
                               GlyphSyms[GlyphIndex])
         }

         // add the corresponding annotation to the pgn move string //
         PGNMove += " " + AnnotationString

         // add the pgn move string to the pgn move list array //
         PGNMoveList.append(PGNMove)
      }
      // if not at beginning of a new white/black move sequence or at end //
      else
      {
         // add the pgn move to the stockfish principal variation string //
         StockfishPV += "(" + PGNMove + ") "
      }
   }

   // if have stockfish analysis to print //
   if !StockfishPV.isEmpty && Inf.Vals("ShowStockfishAnalysis").VB &&
      !PGNMoveList.isEmpty
   {
      // append the stockfish analysis string to the pgn moves list //
      PGNMoveList[PGNMoveList.count - 1].append("..... Stockfish Analysis: "
                                                + StockfishPV)
   }

   // reset the stockish analysis string to empty //
   StockfishPV = ""

   // clear the move rows list //
   MoveRowsList.removeAll()

   // clear the info rows list //
   InfoRowsList.removeAll()

   // loop over all the frame images to display //
   for Index in 0..<GCUI.FramesMovie.ImageFrames.count
   {
      // add the image and text to the move rows list //
      MoveRowsList.append(AListRow(RowImage:GCUI.FramesMovie
                          .ImageFrames[Index], RowText:PGNMoveList[Index]))
   }

   // loop over the game info list //
   for Index in 0..<GameInfoList.count
   {
      // add the text to the game info rows list //
      InfoRowsList.append(AListRow(RowImage:Image(""), RowText:
                          GameInfoList[Index]))
   }

   // initialize the move list and game info table view values //
   (GCUI.Objs["MoveListTableView"]!.ListRows, GCUI.Objs["MoveListTableView"]!
   .Inf.DisplayImage, GCUI.Objs["GameInfoTableView"]!.ListRows) =
   (MoveRowsList, !Inf.Vals("HideIllustrations").VB, InfoRowsList)

   // refresh the game info list font //
   RefreshGameInfoFont()

   // refresh the move list font //
   RefreshMoveListFont()

   // refresh the table views //
   RefreshViews(["MoveListTableView":false, "GameInfoTableView":false])
}

// update the game info font //
@MainActor func RefreshGameInfoFont()
{
   // update the game info list font type //
   (GCUI.Objs["GameInfoTableView"]!.FontType, GCUI.Objs["GameInfoTableView"]!
   .TextColor) = (GameInfoFont.FontType, Color(GCUI
   .GetColorView("GameInfoColorView").GetCurrentHue()))
}
```

```
// update the move list font //
@MainActor func RefreshMoveListFont()
{
   // update the move list font type //
   (GCUI.Objs["MoveListTableView"]!.FontType, GCUI.Objs["MoveListTableView"]!
   .TextColor) = (self.MoveListFont.FontType, Color(GCUI
   .GetColorView("MoveListColorView").GetCurrentHue()))
}

// convert the UCI move to origin and destination squares //
@MainActor func ProcessUCIMove()
{
   // if have pawn promotion, extract piece type the pawn is promoting to //
   MovingPiece.ExtractPromotionTypeIfNeeded()

   // update the start an end squares for the uci move //
   BPData.UpdateStartAndEndSquareForMove(MovingPiece.UCIMove, Dims:Dims,
                                         Rotated:GCUI.BPComp.Inf
                                         .Vals("RotateBoard").VB)

   // set the piece that is going to be animated //
   MovingPiece.Set(BPData.StartSquareValue())

   // remove the piece from the original square //
   BPData.RemoveStartSquarePiece()
}

// set the position editor cursor //
@MainActor func SetCursor(_ Type:Int)
{
   // set position editor moving piece and position editor cursor image //
   (PosMovingPiece, PosCursor) = (Type, NSCursor.init(image:(Type == 0) ?
   Graphics.CGImageToNSImage(ImageRenderer(content:IconImage("xmark.square",
   .red)).cgImage!, Size:Dims.Square.Width) : Graphics.CGImageToNSImage(GCUI
   .BPComp.GetPieceCGImage(Type), Size:Dims.Square.Width), hotSpot:CGPoint(x:
   Dims.Square.Width * 0.5, y: Dims.Square.Width * 0.5)))
}

// update the multithread stockfish engine option //
@MainActor func UpdateMultithreadStockfish(_ Key:String)
{
   // toggle the multithread stockfish engine option //
   Inf.ToggleVB("VBPref", Key:Key)
}

// update the autosave game toggle button //
@MainActor func UpdateAutosaveGame(_ Key:String)
{
   // toggle the autosave game toggle button //
   Inf.ToggleVB("VBPref", Key:Key); StoreAutoData("")
}

// update if using the opening book //
@MainActor func UpdateUseEcoBook(_ Key:String)
{
   // toggle if using the opening book //
   Inf.ToggleVB("VBPref", Key:Key)
}

// update the hide illustrations in game moves option //
@MainActor func UpdateHideIllustrations(_ Key:String)
{
   // toggle the hide illustrations in game moves option //
   Inf.ToggleVB("VBPref", Key:Key)

   // refresh all //
   Refresh("All")
}

// go right one move in the game explorer //
@MainActor @objc func GameExplorerGoRight() {GameExplorerMoveShift(1)}

// go right in the game gallery for the game review //
@MainActor @objc func GalleryGoRight() {GameGalleryShift(1)}

// review the gallery //
@MainActor func ReviewGallery()
{
   // if the game gallery review maximum position is zero, return //
   if GCUI.FramesMovie.MaxPosition == 0 {return}

   // if currently reviewing the game gallery //
   if GCUI.GetGalleryView("GalleryView").Inf.Vals("Reviewing").VB
   {
      // stop the game gallery review timer //
      GalleryReviewTimer.invalidate()

      // set that no longer reviewing the game gallery //
      GCUI.GetGalleryView("GalleryView").Inf.TurnOffVB("VB", Key:"Reviewing")
   }
   // not currently reviewing the game gallery //
   else
   {
      // create the gallery review timer //
      GalleryReviewTimer = Timer.scheduledTimer(timeInterval:0.5 + Double(100
                           - Inf.Vals("GalleryAnimationRate").VI) / 35,
                           target:self, selector:#selector(GalleryGoRight),
                           userInfo:nil, repeats:true)

      // add the gallery review timer to the main run loop //
      RunLoop.current.add(GalleryReviewTimer, forMode:RunLoop.Mode.common)

      // set that currently reviewing the game gallery //
      GCUI.GetGalleryView("GalleryView").Inf.TurnOnVB("VB", Key:"Reviewing")
   }

   // refresh the gallery toolbar //
   RefreshViews(["GalleryToolbar":GCUI.GetGalleryView("GalleryView").Inf
               .Vals("Reviewing").VB])
}

// review the current game being played //
@MainActor func ReviewGame()
{
   // return if there are no moves to review //
   if GCUI.Attributes.Inf.GetCount() == 0 {return}

   // if already reviewing the game //
   if Inf.Vals("ReviewingGame").VB
   {
      // stock the game explorer review timer //
      GameExplorerTimer.invalidate()

      // set that no longer reviewing game //
      Inf.TurnOffVB("VB", Key:"ReviewingGame")
   }
   // if wasn't already reviewing the game //
   else
   {
      // set that currently reviewing the game //
      Inf.TurnOnVB("VB", Key:"ReviewingGame")
```

```
      // set the game explorer review timer //
      GameExplorerTimer = Timer.scheduledTimer(timeInterval:0.5 + Double(100
                          - Inf.Vals("AnimationRate").VI) / 20, target:self,
                          selector:#selector(GameExplorerGoRight), userInfo:
                          nil, repeats:true)

      // add the gallery review timer to the main run loop //
      RunLoop.current.add(GameExplorerTimer, forMode:RunLoop.Mode.common)
   }

   // refresh the game explorer toolbar //
   RefreshViews(["GameExpToolbar":Inf.Vals("ReviewingGame").VB])
}

// start a new game //
@MainActor @IBAction func NewGameMenuItemAction(_:AnyObject)
{
   // set that currently querying if want to start a new game or not //
   Inf.TurnOnVB("VB", Key:"QueryStartNewGame")

   // make the new game alert //
   MakeAlert(["Yes", "No"], Query:"New Game?")
}

// process the alert //
@MainActor func ProcessAlert()
{
   // if doing query for invalid pgn file //
   if Inf.Vals("QueryInvalidPGNFile").VB
   {
      // set query save position to false //
      Inf.TurnOffVB("VB", Key:"QueryInvalidPGNFile")
   }
   // if querying to quit the application //
   else if Inf.Vals("QueryQuitApplication").VB
   {
      // if user chose the first alert button //
      if ReturnCode == NSApplication.ModalResponse.alertFirstButtonReturn
         .rawValue
      {
         // tell the uci engine to stop //
         Stockfish.WriteStockfishToStop()

         // if have an attached sheet //
         if CurrentWindow() != MainWindow
         {
            // end the attached sheet //
            MainWindow.endSheet(CurrentWindow())
         }

         // close the main window //
         MainWindow.close()

         // terminate the application //
         NSApplication.shared.terminate(self)
      }
      // if user doesn't want to quit the application //
      else
      {
         // set that no longer querying whether want to quit application //
         Inf.TurnOffVB("VB", Key:"QueryQuitApplication")
      }
   }
   // if save position query //
   else if Inf.Vals("QuerySavePosition").VB
   {
      // set query save position to false //
      Inf.TurnOffVB("VB", Key:"QuerySavePosition")
   }
   // if loading a saved position and there are no saved positions //
   else if Inf.Vals("QueryLoadPosition").VB
   {
      // set loading saved position to false //
      Inf.TurnOffVB("VB", Key:"QueryLoadPosition")
   }
   // if querying that timed game is over //
   else if Inf.Vals("QueryTimedGameOver").VB
   {
      // remove the timed game offset //
      RemoveTimedGameOffset()
   }
   // if querying whether wants to save the edited board position //
   else if Inf.Vals("QuerySaveEditedPosition").VB
   {
      // if the fen layout is legal //
      if FENLayoutIsLegal()
      {
         // if user chose to save the edited position //
         if ReturnCode == NSApplication.ModalResponse.alertFirstButtonReturn
            .rawValue
         {
            // save the edited position //
            Inf.TurnOnVB("VB", Key:"WillSaveEditedPosition")
         }

         // if didn't choose to edit the position again //
         if ReturnCode != NSApplication.ModalResponse.alertSecondButtonReturn
            .rawValue
         {
            // exit the position editor //
            ExitPositionEditor()
         }
      }
      // the fen layout was not legal and chose to exit //
      else if ReturnCode == NSApplication.ModalResponse
               .alertSecondButtonReturn.rawValue
      {
         // exit the position editor //
         ExitPositionEditor()
      }

      // set that no longer querying to save edited position //
      Inf.TurnOffVB("VB", Key:"QuerySaveEditedPosition")
   }
   // if querying user if wants to start a new game or not //
   else if Inf.Vals("QueryStartNewGame").VB
   {
      // if user chose the first alert button //
      if ReturnCode == NSApplication.ModalResponse.alertFirstButtonReturn
         .rawValue
      {
         // start a new game of chess //
         StartNewGame("All")

         // initialize the game clock //
         InitializeGameClock()

         // store the autosave game data //
         StoreAutoData("")
      }
```

```
      // set that done querying user if wants to start a new game or not //
      Inf.TurnOffVB("VB", Key:"QueryStartNewGame")
   }
   // if querying what types of piece user wants to promote pawn to //
   else if Inf.Vals("QueryPawnPromotion").VB
   {
      // get the return code from the button clicked on //
      ReturnCode -= NSApplication.ModalResponse.alertFirstButtonReturn
                    .rawValue

      // promote the pawn to the specified piece //
      MovingPiece.PromoteTo(Board, ReturnCode:ReturnCode)

      // finish up the move //
      FinishUpMove()
   }
}

// hide the application window //
@MainActor @IBAction func HideAppMenuItemAction(_:AnyObject)
{
   // if the main window is not hidden, hide the app //
   if !Inf.Vals("MainWindowHidden").VB {NSApp.hide(nil)}

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

   // update the window open status //
   UpdateWindowOpenStatus(Inf.Vals("MainWindowHidden").VB)
}

// if have an attached sheet //
@MainActor func HaveAttachedSheet() -> Bool
{
   // return if have an attached sheet or not //
   return MainWindow.attachedSheet != nil
}

// get the current window //
@MainActor func CurrentWindow() -> AWindow
{
   // return the current window //
   return (MainWindow.attachedSheet as? AWindow == nil) ? MainWindow :
          (MainWindow.attachedSheet as? AWindow)!
}

// start the sheet //
@MainActor func StartSheet()
{
   // set that going into a panel //
   Inf.TurnOnVB("VB", Key:"InPanel")

   // set the arrow cursor //
   NSCursor.arrow.set()

   // hide the current window if it is not the main window //
   if CurrentWindow() != MainWindow {CurrentWindow().alphaValue = 0}
}

// end the sheet //
@MainActor func EndSheet()
{
   // unhide the current window //
   CurrentWindow().alphaValue = 1

   // set that no longer in a panel //
   Inf.TurnOffVB("VB", Key:"InPanel")
}

// determine if user wants to quit the application //
@MainActor @IBAction func QuitAppMenuItemAction(_:AnyObject)
{
   // set that currently querying user if wants to quit the application //
   Inf.TurnOnVB("VB", Key:"QueryQuitApplication")

   // update the window open status //
   UpdateWindowOpenStatus(true)

   // alert the user if they want to quit the application or not //
   MakeAlert(["Quit", "Cancel"], Query:"Quit GreenChess?")
}

// undo the last chess move played in the game //
@MainActor @IBAction func UndoMoveMenuItemAction(_:AnyObject)
{
   // return if not undoing to the previous move //
   if !GCUI.Attributes.Inf.UndoToPreviousVS() {return}

   // refresh all the views //
   Refresh("All")

   // compact the transposition table //
   EngineCompactTransTable(Engine, false)

   // store the autosave game data //
   StoreAutoData("")
}

// remove the invalid moves and update the pgn notation //
@MainActor func RemoveInvalidMovesAndUpdateNotation(_ Type:String)
{
   // initialize the board //
   InitializeBoardAttributes(Board)

   // loop over all the moves in the move list including analysis moves //
   for Index in 0..<GCUI.Attributes.Inf.AnalysisCount()
   {
      // mark the move as initially being valid //
      GCUI.Attributes.Inf.MarkValid("Moves", Key:Index)

      // make the move on the board //
      MakeMoveOnBoard(GCUI.Attributes.Inf.Vals(Index).VS)

      // if the move exists //
      if Inf.Vals("MoveExists").VB
      {
         // save the pgn notation //
         GCUI.Attributes.Inf.Update("Moves", VNSItems:[Index:
                                 String(cString:BoardPGNLabel(Board))])
      }
      // the move doesn't exist //
      else
      {
         // mark the move an invalid //
         GCUI.Attributes.Inf.MarkInvalid("Moves", Key:Index)
      }
   }

   // compress the moves based on the type //
   GCUI.Attributes.Compress(Type)

   // initialize the board //
   InitializeBoardAttributes(Board)
```

```
      // loop over all the moves and make them on the board //
      for Index in 0..<GCUI.Attributes.Inf.GetCount()
      {
         // make the move on the board //
         MakeMoveOnBoard(GCUI.Attributes.Inf.Vals(Index).VS)
      }
   }

   // replay the moves from start to end //
   @MainActor func ReplayMoves()
   {
      // initialize the board //
      InitializeBoard()

      // loop over all the moves in the move list //
      for Index in 0..<GCUI.Attributes.Inf.GetCount()
      {
         // get the move from the move list to animate //
         MovingPiece.SaveUCIMove(GCUI.Attributes.Inf.Vals(Index).VS)

         // make the move on the board //
         MakeMoveOnBoard(MovingPiece.UCIMove)

         // process the current uci move //
         ProcessUCIMove()

         // update pawn to the specified piece if needed //
         MovingPiece.UpdateToPromotionType()

         // put the piece that moved onto the destination square //
         UpdateDestinationSquare()

         // update the chess board data //
         UpdateBoardData(Index + 1)
      }
   }

   // update the board preview images //
   @MainActor func UpdatePreviews(_ RefreshBoard:Bool, Board:Bool, Diagram:Bool,
                                  Texture:Bool)
   {
      // run on the main thread //
      Task {await MainActor.run
      {
         // refresh the board if need to //
         if RefreshBoard {Refresh("Board")}

         // update the board preview image if need to //
         if Board
         {
            // update the board image //
            GCUI.Objs["BoardPreview"]!.TheImage = BoardImage(-1, Size:800,
                                                             Update:false)
         }

         // update the diagram preview image if need to //
         if Diagram
         {
            // update the diagram preview image //
            (GCUI.Objs["DiagramPreview"]!.TheImage, GCUI.Objs["DiagramPreview"]!
            .Pads) = (BoardImage(-1, Size:Inf.VF("IllustrationSize") * 2,
            Update:false), 0.5 * (265 - Inf.VF("IllustrationSize")))
         }

         // update textured window background preview if need to //
         if Texture
         {
            // update the textured window background preview //
            GCUI.Objs["WindowPreview"]!.TheImage = Image(WindowTexture.TheImage,
                                                         scale:2, orientation:
                                                         Image.Orientation.up,
                                                         label:Text(""))
         }
      }}
   }

   // update the capture and threaten highlights //
   @MainActor func UpdateCaptureThreatenHighlightLists()
   {
      // set the board so can check for squares where pieces are threatened //
      BoardCopy(ThreatenedBoard, Board)

      // update the capture and threaten highlights //
      BPData.UpdateCaptureThreatLists(String(cString:
            BoardGetListOfCaptureSquares(Board)).components(separatedBy:"*"),
            ThreatSquares:String(cString:
            BoardGetListOfThreatenedSquares(ThreatenedBoard))
            .components(separatedBy:"*"))
   }

   // update the chess board data //
   @MainActor func UpdateBoardData(_ Index:Int)
   {
      // update the chess board data //
      GCUI.PGNMovesData.Update(Index, BPData:BPData)
   }

   // refresh the graphics //
   @MainActor func Refresh(_ Tasks:String)
   {
      // if need to replay the moves //
      if Tasks.contains("Replay") || Tasks.contains("All") {ReplayMoves()}

      // if need to recomposite the board and refresh the view //
      if Tasks.contains("Board") {RefreshView(true)}

      // if just need to refresh the view //
      if Tasks.contains("View") || Tasks.contains("All") {RefreshView(false)}

      // if need to refresh all the table views //
      if Tasks.contains("Tables") || Tasks.contains("All") {RefreshTableViews()}
   }

   // refresh the main view //
   @MainActor func RefreshView(_ UpdateBoard:Bool)
   {
      // set the main view image //
      MainViewImage = BoardImage(-1, Size:1320, Update:UpdateBoard)

      // update the sketch if diagramming //
      if Inf.Vals("Diagramming").VB
      {
         // set the sketch background to the main view image //
         Sketch.SetBackground(MainViewImage)

         // refresh the sketch //
         Sketch.Refresh(false)
      }

      // update the main view if not diagramming //
      if !Inf.Vals("Diagramming").VB {display(bounds)}
   }
}

// get the application value //
let ReturnValue:Int32 = NSApplicationMain(CommandLine.argc, CommandLine
                                          .unsafeArgv)
```