Lib

This module exposes the `play` function which, when called, starts up and runs the entire game.

It is in this function that SDL is initialized, and the threads for graphics, video, processing, and user input are created. By the time this function returns, the game window is closed.

```
{-# LANGUAGE OverloadedStrings #-}

module Lib (play) where
  import Control.Concurrent
  import Lib.Model
  import SDL

  play :: IO ()
  play = do
```

The first step is to initialize SDL and open up a window with a renderer. This is all pretty standard SDL behaviour.

```
    initializeAll -- TODO: probably don't need all systems
    -- TODO: figure out how graphics settings will work
    window ← createWindow "Definitely Not Fire Emblem"
      ( defaultWindow
          { windowBorder = False
          , windowMode = Fullscreen
          }
      )
    renderer ← createRenderer window (-1)
      ( defaultRenderer
          { rendererType = AcceleratedVSyncRenderer
          }
      )
```

Next is to initialize the game-related constructs. Basically just creating an instance of the `Model`. In future versions, this is where a save file may be read in to restore the user's settings to how they had them when the game was last closed.

As the model is going to be shared across threads, it is placed into a simple `MVar`. After some thought, it was determined that the requirements this game

has in terms of processing power and need for fully optimizable model updates is relatively low, so simplicity of the implementation is the way to go.

```
model ← newMVar newGame
audioChannel ← newChan
```

Finally come the four main threads of the game, in no particular order.

The first is the rendering thread. At each step it clears the screen, reads in the model, renders them, and presents the final image to the player. The model it is using, however, is a copy of the one that is in the `MVar`. This way the renderer can render an entire frame while the other threads continue to process inputs and update the model. Since this game does not require frame perfect interactions, we can sacrifice consistency for speed, to some extent.

```
forkIO $ do
  let renderLoop =
    clear renderer
    -- TODO: render the game
    present renderer
    renderLoop
  renderLoop
  return ()
```

The next thread is the audio thread. The audio thread awaits signals on the channel shared with the other threads, and plays the requested sounds at the next convenient opportunity.

```
forkIO $ do
  let playSound =
    audio ← readChan audioChannel
    -- TODO: play the sound
    playSound
  return ()
```

Finally come the interaction and game loop threads.

```haskell
-- TODO: are these two the same thing? how do I regulate the speed
    they run at?
forkIO $ do
  -- Game loop thread
  return ()

-- Interaction thread
events ← pollEvents
quit
```