Lib.RC

Short for rendering context, the `RC` is used to construct the `StateRC`, allowing textures, surfaces, and fonts to be created or loaded and stored for later use by a `Renderer`.

```
module Lib.RC
  ( Key
  , keyFor
  , RC
  , newRC
  , StateRC
  , getTexture
  , freeTexture
  , getSurface
  , freeSurface
  , getFont
  , freeFont
  , getRenderer
  ) where
  import Control.Monad.State
  import qualified Data.Map as Map
  import Data.Map (Map, (!?))
  import Data.Symbol
  import SDL (Surface, Texture, Renderer)
  import qualified SDL
  import SDL.Font (Font)
  import qualified SDL.Font as Font
```

The main feature of this module is the `StateRC` monad, which is really just the `State` monad with an `RC` as its environment. The `StateRC` monad is capable of performing IO, as it is an instance of `MonadIO`.

The `RC` itself holds three types of resources (`Texture`, `Surface`, and `Font`) as well as the actual `Renderer` they are providing the context for.

```
  type StateRC a = StateT RC IO a

  newtype RC = RC RC_
  data RC_ = RC_
    { textures :: Map Symbol Texture
    , surfaces :: Map Symbol Surface
    , fonts    :: Map Symbol Font
    , renderer :: Renderer
    }
```

```
newRC :: Renderer → RC
newRC = RC ∘ RC_ Map.empty Map.empty Map.empty
```

As the `RC` is actually implemented by an internal type, a few helper methods exist to make working with the `StateRC` monad a little easier.

```
getRC :: StateRC RC_
getRC = do
  RC rc ← get
  return rc

putRC :: RC_ → StateRC ()
putRC = put ∘ RC

getsRC :: (RC_ → a) → StateRC a
getsRC f = f <$> getRC
```

Using `Keys`, the resources of an `RC` can be accessed when within the `StateRC` context. A key encapsulates both the identifier for a resource as well as the function used to generate said resource, so a given key will always refer to the same resource, which can (and only ever will) be loaded when it is required with no extra thought from the programmer.

```
newtype Key a = Key (Key_ a)
data Key_ a = Key_ Symbol (IO a)

keyFor :: String → IO a → Key a
keyFor str accessor = Key $ Key_ (intern str) accessor
```

Passing a `Key` to the `get*` functions will look up the corresponding key in the state. If the key's resource already exists in memory, it is returned. If not, the key is evaluated to create the resource, which is then cached by the state before being returned.

The `getRenderer` function simply produces the associated `Renderer`.

```
addTexture :: Symbol → Texture → StateRC Texture
addTexture symbol tex = do
```

```
  rc ← getRC
  putRC rc { textures = Map.insert symbol tex (textures rc) }
  return tex

getTexture :: Key Texture → StateRC Texture
getTexture (Key (Key_ symbol accessor)) = do
  tex ← getsRC (flip (!?) symbol ∘ textures)
  case tex of
    Nothing   → liftIO accessor >>= addTexture symbol
    Just tex  → return tex

addSurface :: Symbol → Surface → StateRC Surface
addSurface symbol surf = do
  rc ← getRC
  putRC rc { surfaces = Map.insert symbol surf (surfaces rc) }
  return surf

getSurface :: Key Surface → StateRC Surface
getSurface (Key (Key_ symbol accessor)) = do
  surf ← getsRC (flip (!?) symbol ∘ surfaces)
  case surf of
    Nothing   → liftIO accessor >>= addSurface symbol
    Just surf → return surf

addFont :: Symbol → Font → StateRC Font
addFont symbol font = do
  rc ← getRC
  putRC rc { fonts = Map.insert symbol font (fonts rc) }
  return font

getFont :: Key Font → StateRC Font
getFont (Key (Key_ symbol accessor)) = do
  font ← getsRC (flip (!?) symbol ∘ fonts)
  case font of
    Nothing   → liftIO accessor >>= addFont symbol
    Just font → return font

getRenderer :: StateRC Renderer
getRenderer = getsRC renderer
```

Finally the `free*` functions provide the opposite effect as the `get*` functions: the resource associated with an identifier is removed from the `RC`, freeing the memory for later use.

```
removeTexture :: Symbol → StateRC ()
removeTexture symbol = do
```

```
  rc ← getRC
  putRC rc { textures = Map.delete symbol (textures rc) }

freeTexture :: Key Texture → StateRC ()
freeTexture (Key (Key_ symbol _)) = do
  tex ← getsRC (flip (!?) symbol ∘ textures)
  case tex of
    Nothing  → return ()
    Just tex → do
      SDL.destroyTexture tex
      removeTexture symbol

removeSurface :: Symbol → StateRC ()
removeSurface symbol = do
  rc ← getRC
  putRC rc { surfaces = Map.delete symbol (surfaces rc) }

freeSurface :: Key Surface → StateRC ()
freeSurface (Key (Key_ symbol _)) = do
  surf ← getsRC (flip (!?) symbol ∘ surfaces)
  case surf of
    Nothing  → return ()
    Just surf → do
      SDL.freeSurface surf
      removeSurface symbol

removeFont :: Symbol → StateRC ()
removeFont symbol = do
  rc ← getRC
  putRC rc { fonts = Map.delete symbol (fonts rc) }

freeFont :: Key Font → StateRC ()
freeFont (Key (Key_ symbol _)) = do
  font ← getsRC (flip (!?) symbol ∘ fonts)
  case font of
    Nothing  → return ()
    Just font → do
      Font.free font
      removeFont symbol
```