TME6 - Squelette de jeu vidéo

Sorbonne Université - Master Informatique M1 - STL

MU4IN510 Programmation Avancée en Fonctionnel - 2021

Dans ce TME en forme de tutoriel, nous allons réaliser une architecture minimimale pour la programmation d'un jeu vidéo 2D en Haskell, en se basant sur la bibliothèque sdl2. Ce squelette de jeu pourra servir de base à la réalisation du projet PAF.

Contexte : la bibliothèque SDL2

La bibliothèque SDL2 est une bibliothèque portable, codée en C, et permettant d'exploiter les ressources des cartes graphiques pour la réalisation de jeux en deux dimensions. Il existe une bibliothèque Haskell surcouche de SDL (documentation sur https://hackage.haskell.org/package/sdl2-2.5.0.0) qui donne accès, avec une interface d'assez haut-niveau, aux différentes fonctionnalités de la bibliothèque. Comme il est tout de même assez difficile de partir de 0, nous avons réalisé pour vous quelques abstractions qui vous faciliteront sans doute un peu la tâche.

- dans le module `TextureMap` : abstraction permettant de charger des images pour les stocker, sur la carte graphique, sous forme de *textures*
- dans le module `Sprite` : abstraction pour les *lutins* de jeu vidéos, chaque lutin pouvant être associé à plusieurs images (elles-même faisant référence à un nom de texture).
- dans le module `SpriteMap` : une table associative permettant de stocker des *sprites* et de les récupérer ensuite
- dans le module `Keyboard` : une structure permettant de maintenir l'état du clavier

Nous avons commenté ces modules, et ils sont tout à fait compréhensibles donc n'hésitez pas à lire le code source associé, et à éventuellement ajouter des fonctionnalités.

Initialisation de SDL2

Dans le module `Main` (fichier `app/Main.hs`) :

```
main :: IO ()
main = do
  initializeAll
  window <- createWindow "Minijeu" $ defaultWindow { windowInitialSize = V2 640 480 }
  renderer <- createRenderer window (-1) defaultRenderer
```

Ici nous commençons par initialiser SDL2, puis nous créons la fenêtre principale du jeu, en utilisant les options par défaut, à l'exception de la taille de la fenêtre que nous limitons, dans notre cas, à 640x480. Dans une seconde étape, nous créons le *renderer* qui est la principale abstraction de rendu de SDL2 : tout dessin passera par ce dernier.

```
  -- chargement de l'image du fond
  (tmap, smap) <- loadBackground renderer "assets/background.bmp"
                                 TM.createTextureMap SM.createSpriteMap
  -- chargement du personnage
  (tmap', smap') <- loadPerso renderer "assets/perso.bmp" tmap smap
```

Dans ces deux lignes, nous chargeons deux *sprites* : l'image de fond `background` et le personnage du jeu `perso`. Voici le code de chargement pour ce dernier :

```
loadPerso :: Renderer-> FilePath -> TextureMap -> SpriteMap -> IO (TextureMap, SpriteMap)
loadPerso rdr path tmap smap = do
  tmap' <- TM.loadTexture rdr path (TextureId "perso") tmap
  let sprite = S.defaultScale $ S.addImage S.createEmptySprite $
```

```
            S.createImage (TextureId "perso") (S.mkArea 0 0 100 100)
  let smap' = SM.addSprite (SpriteId "perso") sprite smap
  return (tmap', smap')
```

Nous utilisons le `TextureMap` pour réaliser le chargement (`TM.loadTexture`) et nous créons ensuite un *sprite* en lui associant l'image correspondante (on utilise le `TextureId` pour référencer les textures). Pour le sprite, on précise aussi la zone concernée avec `(S.mkArea 0 0 100 100)` (qui correspond à l'image entière). Le *sprite* est ensuite ajouté à la `SpriteMap` et on retourne le couple `(TextureMap, SpriteMap)` (encapsulé dans `IO` car le chargement de texture est un effet de bord).

```
  -- initialisation de l'état du jeu
  let gameState = M.initGameState
  -- initialisation de l'état du clavier
  let kbd = K.createKeyboard
  -- lancement de la gameLoop
  gameLoop 60 renderer tmap' smap' kbd gameState
```

On initialise ensuite l'état du jeu, l'état du clavier et on entre dans la boucle principale du jeu.

Boucle principale

En terme de contrôle, la principale abstraction que nous avons réalisée est la bouche de jeu (*gameLoop*) que nous détaillons ci-dessous.

Toujours dans le module `Main` (fichier `app/Main.hs`) :

```
gameLoop :: (RealFrac a, Show a) => a -> Renderer -> TextureMap -> SpriteMap
                                    -> Keyboard -> GameState -> IO ()
gameLoop frameRate renderer tmap smap kbd gameState = do
  startTime <- time
  events <- pollEvents
  let kbd' = K.handleEvents events kbd
```

On commence par récupérer l'horloge courante, qui nous permettra de contrôler notre vitesse d'affichage (ici, on demande un `frameRate` d'environ 60 images par seconde, ce qui est assez classique). On récupère également la liste des événements en attente avec `pollEvents`. On passe immédiatement cette liste à la fonction `handleEvents` pour mettre à jour le clavier.

```
  clear renderer
  --- display background
  S.displaySprite renderer tmap (SM.fetchSprite (SpriteId "background") smap)
  --- display perso
  S.displaySprite renderer tmap (S.moveTo (SM.fetchSprite (SpriteId "perso") smap)
                                  (fromIntegral (M.persoX gameState))
                                  (fromIntegral (M.persoY gameState)))
  ---
  present renderer
```

Nous arrivons maintenant dans la partie affichage, on efface tout d'abord le *renderer* puis nous affichons l'image de fond, ainsi que le personnage en récupérant ses coordonnées depuis le modèle du jeu (cf. `Model`). Les affichages sont ensuite réalisés avec `present renderer`.

```
  endTime <- time
  let refreshTime = endTime - startTime
  let delayTime = floor (((1.0 / frameRate) - refreshTime) * 1000)
  threadDelay $ delayTime * 1000 -- microseconds
```

Ici on calcule un délai nécessaire pour arriver au `frameRate` désiré.

```
  endTime <- time
  let deltaTime = endTime - startTime
  -- putStrLn $ "Delta time: " <> (show (deltaTime * 1000)) <> " (ms)"
```

```
  -- putStrLn $ "Frame rate: " <> (show (1 / deltaTime)) <> " (frame/s)"
  --- update du game state
  let gameState' = M.gameStep gameState kbd' deltaTime
```

On calcule ensuite le temps passé depuis la dernière mise à jour du modèle du jeu, et on invoque le `gameStep` qui prend en compte ce raffraîchissement. Comme notre "jeu" (ébauche de jeu) est très simple, on n'a pas besoin du *deltaTime* mais dans la plupart des jeux vidéos cette information sera très utile.

Remarque : vous pouvez décommenter les affichages du *delta time* et du *frame rate* qui sont bien sûr liés.

```
  ---
  unless (K.keypressed KeycodeEscape kbd')
         (gameLoop frameRate renderer tmap smap kbd' gameState')
```

Finalement, la boucle principale est relancée, sauf si le joueur appuie sur la touche *escape* qui permet de sortir du jeu.

Modèle du jeu

Le *challenge* de la programmation d'un jeu en Haskell concerne la séparation la plus nette possible entre :

- d'un côté les effets de bord : affichage, sons, entrées clavier/souris, etc. qui sont réalisées dans le `main` et encapsulé dans la monade `IO`
- de l'autre côté les calculs purs, qui peuvent être réalisés de façon purement fonctionnelle.

Dans le module `Model` on a placé le "coeur de calcul" du jeu, et nous vous demandons de le modifier pour prendre en compte les déplacements, à la vitesse désirée, du personnage principale dans les limites du terrain.

Exercice 1 : Déplacements clavier

Pour l'instant le personnage ne se déplace pas, à vous de modifier le modèle pour que les déplacements se réalisent.

Exercice 2 : Déplacement souris

En lisant la documentation de SDL2 (cf. https://www.stackage.org/haddock/lts-17.2/sdl2-2.5.3.0/SDL.html), modifier le programme pour prendre en compte les clicks de souris. Sur la sortie standard, la phrase "Touché !" apparaît lorsque l'on clique sur le personnage avec la souris.

Exercice 3 (Défi) : chasse au virus !

Dans le répertoire `assets/virus.bmp` nous avons mis l'image d'un virus. L'idée du défi est d'étendre le modèle de jeu et la boucle principale pour pouvoir réaliser le jeu suivant :

- un virus apparaît à un endroit aléatoire de l'écran (cf. bibliothèque `random`)
- le personnage doit le rejoindre pour le détruire (car il est immunisé)
- etc.

Rappel : il s'agit d'un défi, à ne faire que si cela vous motive et si vous trouvez le temps.