1. Introduction

We have chosen to make a game that represents the given shoot ‘em up-option. We say ‘represents’, as we make a few adjustments.
Our game will be a top scroller; that is to say, the enemies will spawn at the top of the screen and will come down towards the player, instead of sideways. Moreover, it will feature armed people wielding guns, rather than spaceships.
Various types of enemies will feature in this game, with wildly varying behaviors, who give the player a different score upon defeating them, based on the difficulty of these types of enemies.

2. Design

2.1 Data Structures

2.1.1 The System States

```
|        data MainState = MGame GameState | MMenu MenuState | Exit
|
|        data GameState = GameState    {  world :: World
|                                       , score :: Float
|                                       , elapsedTime :: Float
|                                       , gameKeys :: KeysOfInput
|                                       }
|
|        data MenuState = MenuState    {  menu :: Menu
|                                      , game :: GameState
|                                      , menuKeys :: KeysOfInput
|                                      }
|
|        data Menu = Menu              {  options :: [MenuOption]
|                                      , selectPointer :: Int
|                                      }
|
|        data MenuOption = MenuOption {  optionText :: String
|                                      , index :: Int
|                                      , nextState :: MainState
|                                      }
```

We chose a MainState datatype in our architecture, because we want a way to integrate menu’s into our system, and switch between them and the game itself easily. The Mainstate-type has constructors of type GameState called ‘MGame’ and of type MenuState called ‘MMenu’.

MenuState has a constructor of type Menu, which contains the list of options of the menu in question, and a ‘pointer’ which will indicate the currently selected option by the player. Also, it saves the current GameState, so the player can access a menu - primarily the Pause-menu - and resume the game without losing the game’s current progress.
While in the MenuState, the game will be paused until the MainState is changed back to the saved GameState, though when a game has not yet begun, such as when the application is launched and the player is brought to a ‘main menu’ of sorts, the GameState given to MenuState is a predetermined blank GameState. We have also added a property called ‘menuKeys’ which is of type KeysOfInput. This is a list with all the possible input keys for the state the game is in.

The GameState itself consists of a ‘World’-constructor (which we will discuss below), the current score-constructor (which we want to keep track of for our highscore-records) and an elapsedTime-constructor, which will count for each second spent in-game (this is used to scale difficulty and accumulated score over time). We also added the gameKeys, which is also a KeysOfInput. It has the same functionality as the menuKeys for the menu state.

2.1.2 The World and its entities

```
|       data World = World    { player :: Entity
|                             , entities:: [Entity]
|                             , scrollspeed :: Float
|                             , spawnIncrement :: Float
|                             , overlay :: String
|                             , accumalatedScore :: Float
|                             , background :: Picture
|                             }
```

The datatype of World has several constructors, which together make up the playing field. First is the player itself, then all other ‘entities’ put together in a list, and lastly the current background of type Picture (this may vary per level or change over time).
We chose to construct a list of all ‘entities’, so we can easily check for collisions amongst these in a like-mannered way. The player-’entity’ is taken individually at the moment, but that is because we will want to interact with the player individually (say for handling input), and we can easily add this single element to the list ‘entities’ when checking all for collision.
We will further explain our ‘Entity’-datatype and other relevant datatypes below. We have also added the scrollspeed, spawnIncrement, overlay and accumalatedScore constructors to the World datatype. The scrollspeed is a speed with which the world scrolls down, this also affects the speed of some of the entities, the scrollspeed will increase overtime. Spawnincrement is a Float which in its essence is a timer. If the correct amount of time passes a new enemy can spawn, this is also going to increase overtime. The accumulated score is a Float that increases each time you kill an enemy or deflect a bullet. The background is a Picture which moves with the scroll speed for animation purposes. An overlay constructor has also been implemented, but this is mostly for debugging purposes

```
|
|        data Entity = Entity      { etype :: EntityType Float
|                                  , faction :: Faction
|                                  , hitbox :: Hitbox
|                                  , movement :: Movement
|                                  , sprite :: Picture
|                                  }
|
|        data Movement             { location :: Point
|                                  , speed :: Float
|                                  , angle :: Float
|                                  , direction :: (Float, Float)
|                                  , movementPaternID :: Int
|                                  , moveWithWorld :: Bool
|
|        data EntityType a =       Shooter a | Bullet a | Obstacle a
|
|        data Faction         =    Player | Enemy | Neutral
|
|        type Point           =    (Float, Float)
|
|        type Hitbox          =    (Float, Float)
```

The Entity-datatype is the general type for all ‘entities’ (or to put it in other words, for all shooters, bullets, obstacles, etc.).
Each entity has general information that is relevant to their movement and collision, such as ‘movement’, ‘hitbox' and to display these correctly, ‘sprite’.
The Movement datatype consists of the parameters which are needed to move an Entity. These are the location, which lets us move an entity together with its speed (which is also a parameter). The movement of an Entity is also determined by the direction, the movementPaternID and the moveWithWorld parameters. We use the direction and the speed to manipulate an entities location. if moveWithWorld is true, the scrollspeed of the world will also be taken into account for the movement speed of that entity. The movementPaternID determines how an Entity moves.
Furthermore, each entity has a ‘faction’. This is an attribute which will define which entities may collide with what other entities. For instance, we don’t want a Shooter of the faction ‘Player’ to be shot by a bullet of type ‘Player’, which would likely be its own bullet.

Finally, there is ‘EntityType’, which will describe what kind of entity we are exactly dealing with, and defines the final part of its behaviour.
An Entity can be a Shooter, a Bullet or an Obstacle, but this could easily be expanded in the future. A Shooter will have an argument of type Float (given by the Entity-datatype), which will represent its health. For an entity of EntityType Bullet, this argument of type Float will determine how much damage the bullet will cause on collision. Lastly, the Obstacle-type, like Shooter, has an argument of type Float which represents how much health it has.

Datatypes Point and Hitbox are pretty self-explanatory: One determines the location of the entity on the field, whereas the other determines its hitbox (which is useful when checking for collision), respectively.

2.1.3 Player and Movement

As discussed in 2.1.2, Player is of type ‘Entity’. Its faction-argument will be ‘Player’, and its EntityType will be ‘Shooter’. It behaves largely in the same way as the ‘enemy shooters’, except, it is controlled by the player, rather than being moved based on an algorithm over the list of ‘entities’ (more on this in 2.2).

The player will be able to move side-to-side and slightly up-and-down, within a certain invisible box on the bottom half of the screen. The movement will be handled by the player pressing the keyboard buttons of ‘WASD’, which move them up, left, down, or right respectively. Finally, the spacebar will have the player shoot a bullet.

2.1.4 Purity of the game

Entities, such as enemies, we want to be impure, because we want these to ‘spawn’ from random locations from the top of the screen (as our game will be a topscroller), and be randomly selected from a list of enemy-types.

Entity interactions (such as collision and enemy-AI), speed of entities, and damage calculation we want to be pure and always result in the same outcome, depending on factors such as ‘bullet damage’, which we will change between enemies and scale over time.

Player controls will always respond the same, and thus be pure. That is to say, pressing the ‘left’-arrowkey will always cause the player-entity to move to the left, for instance.

2.2 Computer Adversary

We would like to implement different types of enemies if possible. Differences between these enemies would be in their behavior. One enemy might shoot at the player in a different way (say, they 'aim' at the player, adjusting the angle of their shot based on where the player is located in relation to the enemy's location), or they may adjust their vertical position to align themselves with the player on the x-axis, whereas other enemies will fly in a unique pattern or disassemble upon death and split into more enemies. A list of the chosen enemies may be found below:

- Static enemy:
  An enemy stands 'still', is static to the background, and will aim at the player by shooting a bullet with an angle based on a 'drawn line' between this enemy's location and the player's location.
- Generic enemy:
  This enemy flies straight across the map much faster, and also aims at the player's current location to shoot.
- Aiming enemy:
  This one will 'chase' the player by actively trying to align its position with that of the player on the x-axis, shooting when 'in sights'. This enemy will remains at top of screen for a time, before its behavior converts into that of the enemy we listed above.

They will be represented as an 'Entity' of 'EntityType' Shooter, and Faction set to Enemy. These entities will be pre-written somewhere in the code, and called upon when randomly selected to spawn.

For example the initiation of our 'static enemy', excluding arguments which will require more depth (denoted by <>'s):

```
|       staticEnemy :: Entity
|       staticEnemy =  Entity (Shooter 30) Enemy  (2,2) 0 <Movement>
```

The enemytypes we will then gather into a list of tuples, of type [(Entity, Int)], where 'Entity' represents the enemy and the 'Int' its respective chance of spawning. These numbers will then somehow change as time goes on in-game, or we determine different odds between levels.

2.3 Interface

As our game will be a topscroller-game in which enemies spawn at the top of the screen and will come at the player, the interface for this game will be a scrolling background in which the player can move around and shoot at enemies (see 2.1.3).
Actual things shown on the screen will be the player, the enemies, and ofcourse the bullets that will be shot by both parties, as well as a background that will scroll down. When the

player destroys an enemy their score will increase based on what type of enemy they killed and on how long the game is running for. The score that the player accumulates and the amount of time the game has been running for will be displayed in the top right corner of the screen.

The player only interacts with the game using the keyboard as explained in 2.1.3. This will be done during the gameplay, but the player will also be able to pause the game. In this case the interface will look a little different. A small menu will pop up. The player can now use his arrow keys to select an option on the menu. These options will be to resume the game, end the game or close the entire application. When the player chooses the last two options (or the player dies) the player is asked to type his/her name. The resulting name will then be used to update an external file. The player will see what he/she has typed before it will be confirmed and saved to the external file.

3.1 Implementation of the Minimal Requirements

We implement the minimum requirements as follows:

Player

The player will use the keyboard to control a player entity across the screen from left to right. The player will also be able to use the spacebar to shoot at enemies.

Enemies

Enemies will be of different types as aforementioned. A couple of these enemies will have a form of intelligence. Which will be following the players position, this way the enemies will always shoot at the player.

Randomness

The spawn points of all the enemies will be randomized at the top of the screen. We will use the System.Random library for this. Once the random value has been assigned the enemy will spawn at the top of screen and tries to kill the player.

Animations

Animations are implemented through updating our static enemy’s rotation in accordance to the player’s location.

Pause

When the player presses the pause button the game will freeze and a pause menu will pop up. In this pause menu we will have functions/buttons that correspond with: continuing the game, going back to the main menu, or quitting the game in its entirety.
The

Interaction with the file system

As for interaction with the file system we will keep score during the game. When it is game over then the score will be written to a file, along with the survived time.

3.2 Implementation of the optional Requirements

Different enemies

We plan on using different enemy types. These enemies will have different behavior, and will spawn with regards to the game’s current runtime (as described in 2.2).