PLAN/PROOF-OF-CONCEPT: Game Development in Elm

Yang, Jessica jessicayzt@alumni.ubc.ca
van der Kooi, Tim vanderkooi11@gmail.com
Hong, Karen khong@alumni.ubc.ca
Rodgers, Laura laurarodgers@alumni.ubc.ca

ABSTRACT

Elm is a functional language specialized for building robust web applications. This report outlines our implementation of a prototypical platform game in Elm, and discusses the design and implementation of various features, such as the graphical user interface and the handling of user input. We will elaborate on how this prototype will be extended into a fully-featured game, providing both a low-risk and high-risk approach.

1. OVERVIEW

For our final project, we plan to extend our prototype, which will be discussed in detail in this report. Our previous report investigated the features of Elm that made it suitable for developing an interactive web application and the implications of those features. This report will outline what we have implemented for our prototype, and how we plan to reach our final project goals using our prototype as a guiding point. To illustrate our prototypical work, we will outline the process that we have followed to develop and test the application, and provide code snippets with discussions supported with screenshots of our game.

2. PROTOTYPE FEATURES

The features that we have implemented in our prototypical game, described at a high level, include:

1. An animated avatar with left/right animation variations based on events such as walking, running, idling, jumping, or dying.
2. Keyboard inputs that can make the avatar jump or move left or right.
3. Right movement past the centre results in a side-scrolling screen.
4. Positions of the suspended platforms and platform units are randomized so every instance of the game is different.
5. Platform units in the form of two hazards (spikes and nuclear waste), which do constant damage based on the type of hazard, over time, if the avatar is in collision with the unit.
6. Textual display of avatar HP, speed multiplier, and game score.
7. Score increase when side-scrolling.
8. A bottomless pit that results in immediate death.
9. A game over screen.

3. GAME DESIGN

In this section, we will discuss how we architected and designed our prototypical game, in the context of developing our fully-featured platform game.

3.1. MODULE DESIGN

This will be a discussion on the implementation and interaction of the modules that make up our prototype: `Main`, `Game`, `Avatar`, `GamePlatform`, `Key`, `View`, and `ViewUtil`.

3.1.1. MODULE RELATIONS

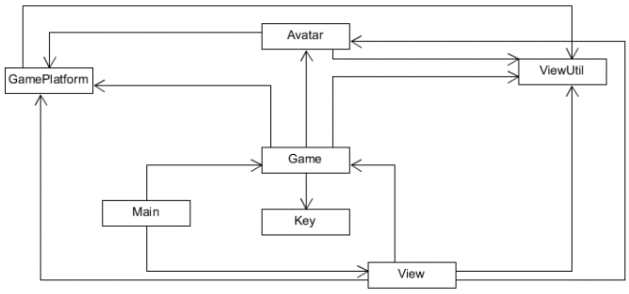

Figure 1

Figure 1 shows a visual representation of the module relations in our game, with arrows indicating dependencies. As circular

dependencies are not allowed, we had to take care to avoid them during module design.

3.1.2. ELM ARCHITECTURE

Like every Elm program, our game makes use of the Elm architecture pattern, which breaks the application down into three parts:

- the "model," or state
- a way to "update" the state
- a way to "view" the state

3.1.2.1. SUBSCRIPTIONS

Our subscriptions are outlined in `Main`:

```
subscriptions : Game -> Sub Msg
subscriptions game =
  Sub.batch
    [ AnimationFrame.diffs TimeUpdate
      , Keyboard.downs KeyDown
      , Keyboard.ups KeyUp
      , Window.resizes Resize
    ]
```

These subscriptions listen for animation frame diffs for smooth animations, keyboard events from the user (whether they pressed or released a key), and window resizes. These subscription messages are handled in our update section.

3.1.2.2. MODEL

The state of our model is initialized by `Game.initGame`

```
initGame : ( Game, Cmd Msg )
initGame =
 ( { size = Size 0 0
   , state = Playing
   , platforms = [GamePlatform.init,
                  GamePlatform.ground]
   , avatar = initialAvatar
   , score = 0
   }
   , Task.perform Resize Window.size
 )
```

which specifies the initial state of the game, the initial platforms, avatar, and game score. A message is also sent out to resize the game to the window size.

The modules that define our model section of the architecture pattern are broken down into `Game`, `GamePlatform`, and `Avatar`. As our model was so comprehensive, it made sense to break it down into separate modules that depend on each other to construct the state together.

3.1.2.3. UPDATE

Our update section acts upon messages produced from `initGame` or `update` itself. Messages are specified with a tagged union type in `Game`:

```
type Msg
   = TimeUpdate Time
   | KeyDown KeyCode
   | KeyUp KeyCode
   | NewPlatform PlatformToGenerate
   | Resize Size
   | NoOp
```

When `update` recieves a `Msg`, it acts accordingly depending on what type of `Msg` it is. As shown above in the union type, this could be a `TimeUpdate, KeyDown, KeyUp,` and so on.

When a `TimeUpdate` message is received, if the game state is `Playing`, `updateGame` will be called to update the game state, and if platform generation is necessary, a `NewPlatform Msg` will be sent out.

The `KeyDown` and `KeyUp Msg` types invoke functions that update the state with the given user input. For example, `KeyDown` on the right arrow key will move the avatar to the right, while `KeyUp` on the same key will stop the avatar's movement.

Our `updateGame` function updates the game state, and is called on each `TimeUpdate Msg`. Updates on the game state include ending the game when the avatar's HP has been depleted, updating the platforms, updating the avatar, and updating the score, increasing it if the player is currently side scrolling.

3.1.2.4. VIEW AND GAME LAYOUT

Our view section represents the model visually.

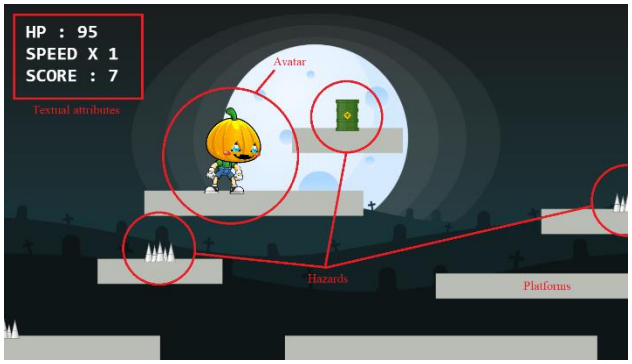

Figure 2

Figure 2 shows a view of our game. Working with the `Graphics.Collage` package for freeform graphics posed some difficulties in terms of working out the placement of the graphics on the cartesian plane. This was mitigated by introducing a `ViewUtil` module that was responsible for providing rough general locations of where various graphics should go.

Overlaying graphics was also a challenge, but after some trial and error with intuition, it made sense to lay the graphics out in the order of the background, the platforms, the platform units, the avatar, and then the textual attributes. The textual attributes, which display the avatar's HP and speed multiplier, and the game score, make sense to be overlaid last, along with the game over text if the game is over.

Rendering the avatar with its correct animated variation was surprisingly easy, as we set up a directory structure for the graphics where constructing a path to a specific animation was simple. After designing the directory structure, the path to any animation was

```
src = "../graphic/avatar/" ++ verb ++
"/" ++ dir ++ ".gif"
```

After computing `verb` (eg. "walk" or "die") and `dir` (ie. "left" or "right") of the avatar, we could access the correct animation with the computed path.

With the textual graphics, we ensured the use of web safe fonts, due to issues we had run into using a font some browsers did not support.

3.1.3. PLATFORMS

Platforms are necessary for the avatar to interact with in its environment through walking on and jumping from platform to platform, as per the norms of a platform game. Platforms also provide platform units (eg. hazards) for the avatar to interact with, enriching the game. By generating platforms and their units with some degree of randomization, we can ensure that every instance of the game is different.

Platform generation was by far the most difficult part of this prototype to implement.

Through Elm's `Random` package for random number generation, we achieved a controllable degree of randomization in our implementation of platforms and platform units.

Using `Random` for the generation of pseudo-random values consists of using `Generator`s, which are like a "recipe" for how to generate values, for example, the range (eg. 1-4, 1-20) and type (eg. `Bool`, `Float`).

Generators can then be combined to generate values together.

A platform is generated with a `NewPlatform Msg`, which uses `GamePlatform.platformGenerator`

```
if List.length game.platforms < 15 then
  ( updateGame game
   , Random.generate NewPlatform
platformGenerator
   )
```

`platformGenerator` uses `Random` to generate a random width, height difference, and `Unit` for the new platform. While specifying an ideal interval for the width was vital to generating the platforms correctly, it was more important to generate the height difference correctly, so that consecutive platforms would be able to be reached by the player through jumping.

The new platform `y` is calculated as the previous platform `y` plus the added randomly generated height difference. If this new `y` value is too high, we generate another extension of the ground instead. The new platform `x` is calculated as the previous platform `x` plus a constant platform gap.

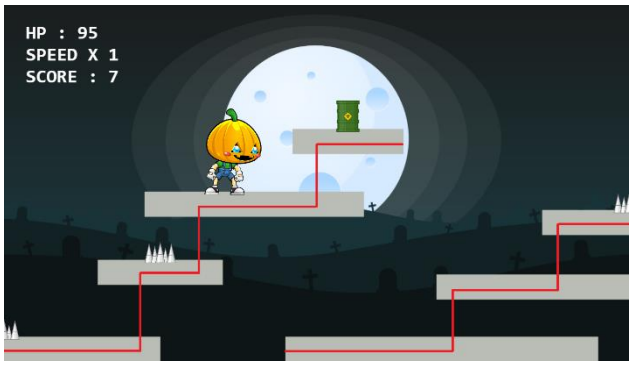

Figure 3

This way of generating platforms creates a "staircase" structure, shown in **Figure 3**, with reasonably randomized widths and height differences between the platforms.

For efficiency, we decided to only keep 15 platforms around at any given time, so that only 15 platforms would ever need to be rendered and updated. As platforms go offscreen to the left, they are removed and replaced with new platforms offscreen to the right. This way, platforms are generated "on demand" during side-scrolling motion.

3.1.4. PLATFORM UNITS

It made sense to model the randomly generated platform units as an easily extensible tagged union type

```
type alias PlatformToGenerate =
   { w : Float
   , heightDiff : Float
   , unit : Unit
   }

type Unit
   = Spikes
   | Waste
   | None
```

With `Random`, it was easy to control the exact probability of various units being generated. In `unitGenerator`

```
unitGenerator : Generator Unit
unitGenerator =
    Random.map assignUnit (Random.int 1
                                        25)
```

we call `assignUnit` with a random number from 1 to 25, inclusive. In `assignUnit`

```
assignUnit : Int -> Unit
assignUnit generated =
    if generated <= 5 then
        Spikes
...
    else
        None
```

we can see that there is a 5/25 (20%) chance of generating `Spikes`.

There is one unit per platform, situated at platform `x`.

3.1.5. AVATAR

Figure 4, our dear hero, Jack O'Lantern

Our avatar implementation has the `type alias`

```
type alias Avatar =
   { x : Float
   , y : Float
   , vx : Float
   , vy : Float
   , dir : Direction
   , hp : Int
   , speedMultiplier : Float
   , invincible : Bool
   }
```

in which `x`, `y`, `vx`, `vy`, and `dir` are necessary for the implementation of physics, allowing our avatar to walk and jump. Our avatar has two directions, `Left` and `Right`, modelled as a tagged union type. `Game.updateAvatar` is where time dependent updates are done, such as constraints, gravity, physics, and collisions (ie. changes of `hp`, `speedMultiplier`, and `invincible` due to collisions with units).

To urge the player to go forward, there is a constraint that the avatar cannot continue going left past the left edge.

Gravity was simple to implement: if the avatar is on a platform, there is no need for gravity. Otherwise, we decrease `vy` (thereby increasing the downward velocity).

Physics was slightly trickier, as we had to take into account our side-scrolling environment. In the end, it made sense to lock the avatar `x` at 0 during side-scrolling, which is checked with avatar `x` being at 0 or greater, and `dir` being `Right`. The rest of the physics implementation was straightforward with `x`, `y`, `vx`, and `vy`.

Collisions with platform units was simple as each platform had a platform unit at the platform's `x`. The logic for collisions in the prototype was to determine if the avatar was currently on a platform, and if it were, to see if it was close to (in our case, within 20) of the platform's `x`. We could then access the platform unit and use `case` to deduce the unit's type, and update the avatar's HP accordingly. If the avatar is not on a platform, we then check whether it is below the pit, in which case we update the HP to 0.

3.2. KEY COMMANDS

For our prototype game, we implemented 3 key commands, to move left, right, and jump. This proved to be very easy, through Elm's subscription model to handle user input. With our `Key` module, we declared a tagged union type

```
type Key
   = LeftArrowKey
   | RightArrowKey
   | SpaceBar
   | Unknown
```

that was easily extensible if we wished to add more key commands to the game. In our `Key` module, we also have a function to convert keycodes to our `Key` type, so we could use `case` instead of messy conditionals over the integer keycode.

On a keydown, the avatar either moves left, moves right, or jumps, depending on the key. On a keyup on the keys that the avatar uses to move left or move right, the avatar idles.

3.3. TESTING AND DEBUGGING

Testing our game manually with `elm-reactor` was straightforward and simple due to our project's fast build time and Elm's friendly compiler errors. The documentation on various Elm packages have been convenient to access and use, and we have not needed to investigate any Elm debugging tools.

4. ADDITIONAL FEATURES FOR FINAL GAME

In this section, we will discuss the minimal core goals we aim to achieve for the final project, and how we will go about achieving these goals.

We will outline the additional features that would be appropriate for our final game, and group them into two categories, low-risk and high-risk. Our low-risk approach consists of implementing all the low-risk features, while our high-risk approach entails implementing all the low-risk features *and* the high-risk features. Our minimal core goals for the final project are the low-risk features.

4.1. LOW-RISK APPROACH

Our low-risk features include:

1. Collectibles
1.1. Bones, which will increase score on collision. There will be two variations, one which will be more valuable than the other.
1.2. HP, which will increase the avatar's HP (which will be capped at 100) by a constant amount on collision.
1.3. Speed boost, which will increase the speed multiplier (which will be capped to a maximum value). The speed multiplier will decrease over time, to a minimum of 1, which is the default.
1.4. Shield, which will give invincibility for a constant time.
2. Bind game restart to a key so players can restart the game whenever they wish.

4.1.1. LOW-RISK PLAN

Our plan for modelling the collectibles (ie. adding them to our model section) is to add each collectible to the `Unit` tagged union type.

As collectibles need to be removed from the platform upon first collision, changes will need to be made to `updateGame`. Hazards will not be able to be collected so they must be distinguishable from collectibles.

The logic for the collectibles should not be too difficult with the foundation of what we have already implemented, as we have done it in an extensible way. The bones and HP collectibles are simpler, as there is no effect over time, just a single effect when it is collected. The shield and speed boost will require that we keep track of the amount of time they have been in effect for the purpose of expiring them.

As we have already implemented 3 key commands, adding another key command to restart the game should be quite straightforward. To restart the game, we could just set the model to the one specified in `initGame`, thereby resetting the model.

4.2. HIGH-RISK APPROACH

Our higher-risk features include:

1. Moving enemy units on platforms as an additional hazard.
2. Smooth animations for the avatar.

4.2.1. PLAN

Our plan to add moving enemy units requires adding an `x` position to platform units. This is necessary as our current prototype's platform units are at their platform's `x`. As these enemy units will need to move back and forth on their platforms, it is necessary to know their current `x` position to calculate when our avatar collides with these units.

Another challenge that adding moving enemy units poses is animating them. As we have animated our avatar in our prototype, we could extrapolate what we have done to animate our avatar to animate our moving enemy `Zombie` unit. This means that it would also need to have a `Dir`, so we could animate it accordingly. It will need to change directions when it reaches the end of the platform, and endlessly walk back and forth.

Our plan to get smooth animations for our avatar is to try this list of fixes:

1. Using smaller file sizes for animations through compression of frames, size, and colour.
2. Preloading/caching animation assets.

If these fail, we plan to reach out online to the Elm community to formulate a solution.

SUMMARY

In this report, we have provided proof-of-concept in the form of a prototype, and a plan to fulfill the final project requirements.