FINAL PROJECT REPORT: 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 final project, a 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 explore the implications of our project in full, enumerating and expanding on both advantageous areas and pain points that we have uncovered using Elm for the purpose of game development.

1. OVERVIEW

We will discuss our final project, an extension of our prototype, in detail in this report. Our previous report investigated the prototype that our final project is based on. This report will outline what we have implemented, illustrating that we have completed both the core goals and full goals that were laid out by our project plan in our previous report. To illustrate our work, we will outline the process that we followed to develop and test our application, and provide code snippets with discussions supported with diagrams and screenshots of our game.

In addition to explaining our process to achieve our end goal, we will consider what our project has uncovered on the usage of the Elm language for game development.

2. FEATURES

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

The features from our prototype:

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.

The features that composed the core goals laid out in our project plan:

1. Additional collectibles
1.1. Bones, which increase score on collision. There are two variations, one which is worth 50 points, and one which is worth 100 and is rarer (less likely to be generated).
1.2. HP, which increases the avatar's HP by 50 on collision, up to a maximum HP of 100.
1.3. Speed boost, which increases the avatar's speed multiplier by 0.5, up to a maximum of 2. The speed multiplier is reset to 1 after 4.5 seconds of not collecting another speed boost.
1.4. Shield, which grants invincibility for 4.5 seconds.
2. A main menu.
2.1. Additional keybindings for the main menu.
2.1.1. ESC quits and returns to the main menu during gameplay.
2.1.2. ENTER starts the game.

The features that composed the full goals laid out in our project plan:

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

Additional features:
1. Sorted high scores on main menu.

3. GAME DESIGN

In this section, we will discuss how we architected and designed our final game, basing off the architecture of our prototypical game.

3.1. MODULE DESIGN

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

3.1.1. MODULE RELATIONS

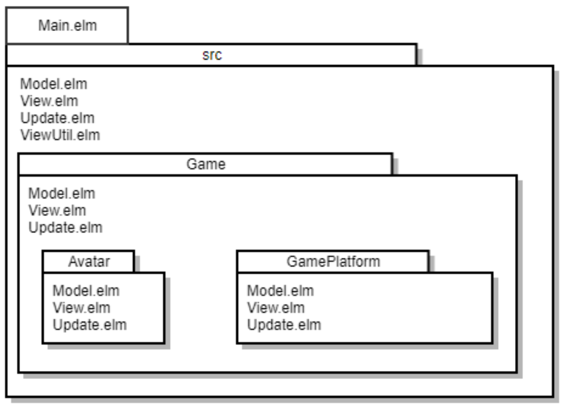

Figure 1

Figure 1 shows a visual representation of the module relations in our game, after refactoring the architecture of our prototype.

Through our research into the Elm architecture, we knew that writing code in the Model-View-Update pattern should be modular and easily extensible. However, as our code grew, these characteristics were not initially obvious. After reconsidering our module design, we came up with a directory structure that took advantage of the architectural pattern of our application.

As the complexity of our application necessitated expansion beyond one file, our code was organized into the following modules: `Main`, `Game`, `Avatar`, `GamePlatform`, `Key`, `View`, and `ViewUtil`. `Game` contained all the message handling, model, and update logic. `Avatar` and `GamePlatform` contained helpers that were called from `Game`. With this structure, we had to take extra care to avoid circular dependencies. In addition, it was not obvious where code belonged, leading to difficult to maintain code.

Refactoring the game into a more modular structure allowed a natural, conceptual separation of game elements and increased readability. While still following the Elm Model-Update-View architecture, we encapsulated all purely gameplay-related code into the `Game` directory. As in the prototype, we placed code concerned with modelling the player-character in an `Avatar` encapsulation, but have now additionally brought code concerned with updating the player avatar's model and rendering the player avatar into a more Elm-like `Avatar` directory with its own `Model`, `View`, and `Update` Elm files. We performed a similar refactoring on non-player controlled game elements (of which there are now a greater variety). These elements are now fully modelled, updated, and rendered by code within the `GamePlatform` directory. Essentially, each subdirectory functions as a standalone Elm application. This not only made it clear where code belonged, but also enforced that it gets added there.

Refactoring our code to this manner is in keeping with the Elm architecture and its intention of separating data from logic. This more modular approach made continuing development much easier than it would be if our application was using single `Update` and `View` files with no conceptual separation for gameplay-related code, avatar-related code and non-player element code.

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

Each of the modules pictured in **Figure 1** use this pattern.

3.1.2.1. SUBSCRIPTIONS

Our subscriptions are outlined in `Main`:

```
subscriptions : Model -> Sub Msg
subscriptions subscriptions =
    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. The window resize subscription is handled in our main update file, the keyboard updates are handled by the update sections in the `Game` and `Avatar` subdirectories, and the `AnimationFrame` updates are handled in the `Game` update section.

3.1.2.2. MODEL

The state of our model is initialized by `Model.initialModel` in the main `Model` file

```
initialModel : Model
initialModel =
    { size = Size 0 0
    , game = Game.model
    , screen = StartScreen
    , highScores = []
    , playerName = "Jack O'Lantern"
    }
```

which specifies the initial state of the game with `Game.model`

```
model : Model
model =
    { state = Playing
    , platforms = [ GamePlatform.model,
                    GamePlatform.ground ]
    , avatar = Avatar.model
    }
```

where the initial platforms and avatar state are specified.

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 sections act upon messages it receives. Messages are specified with tagged union types in each `Update`, for example

```
type Msg
    = TimeUpdate Time
    | KeyDown KeyCode
    | KeyUp KeyCode
    | Resize Size
    | InputName String
    | NewPlatform
GamePlatform.PlatformToGenerate
    | 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. In the case that the `Msg` is relevant for other modules, it is passed through to the relevant subdirectory `update` function.

When a `TimeUpdate` message is received, if the game state is `Playing`, `Game.update` 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 enter key will start the game if you are on the start screen. It could also be the case that the message gets passed along to an update subdirectory. 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. These messages are first handled by `Game.update` and if determined to be relevant, passed on to `Avatar.update`.

Our `Game.update` section 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.

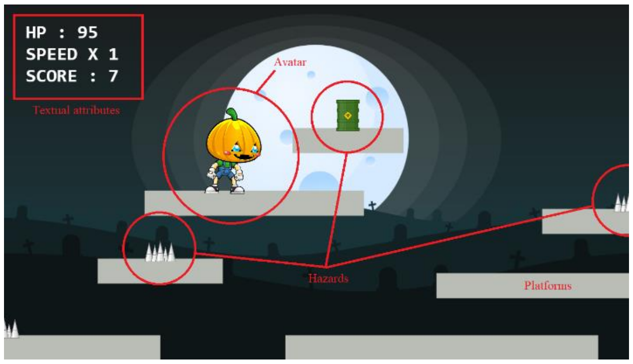

Figure 2

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.

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 and jumping from platform to platform, as per the norms of a platform game. Platforms also provide platform units (ie. hazards and collectibles) 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 the project 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.

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

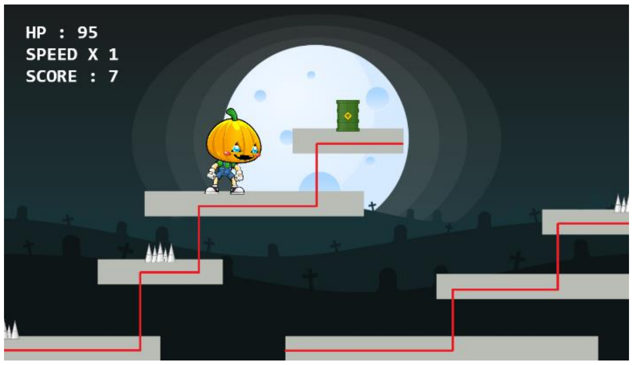

Figure 3

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
    | Zombie ( Float, Direction )
    | HP
    | TwoBones
    | ThreeBones
    | Boost
    | Shield
    | 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 == 1 then
        Zombie ( 0.0, Right )
    else if generated <= 4 then
        Spikes
...
    else
        None
```

we can see that there is a 1/25 (4%) chance of generating a `Zombie`. That means that for every 25 platforms, we should expect about one zombie.

There is one unit per platform, situated at platform `x`, except in the case of the `Zombie` moving enemy unit.

3.1.4.1. COLLECTIBLES

To model the collectibles, we added each collectible to the `Unit` tagged union type.

As collectibles need to be removed from the platform upon first collision, changes were made to `updateGame`. Hazards are not be able to be collected, so they must be distinguishable from collectibles. This is easy as `Unit` is a tagged union type

```
hasCollectible : Maybe Model -> Bool
hasCollectible platform =
    case platform of
        Just platform ->
            case platform.unit of
                Boost ->
                    True

                HP ->
                    True

                ...

                _ ->
                    False

        Nothing ->
            False
```

The logic for the collectibles was not difficult at all with the foundation of what we had already implemented for the prototype, as we had done it in an extensible way. The bones and HP collectibles were simpler, as there is no effect over time, just a single effect when it is collected. The shield and speed boost required that we keep track of the amount of time they have been in effect for the purpose of expiring them.

3.1.4.2. MOVING ENEMY UNITS

Moving enemy units were easy to implement given our existing work on the prototype. In the `Unit` tagged union type, the `Zombie` moving enemy unit has the declaration

```
Zombie ( Float, Direction )
```

This allows us to construct a `Zombie` with a `Float` and a `Direction`. The first of the tuple (the `Float`) is designated as the offset from the platform `x`, which is where all units are situated at if they are not moving units. The second of the tuple is designated the direction the `Zombie` is currently facing.

These two new elements are the only additional elements necessary to model a moving enemy unit, and the logic was simple to implement: on each tick, we increment or decrement its offset depending on its `Direction`, and change the `Direction` when it is at the end of the platform.

The collision logic for the `Zombie` unit is simply to calculate the platform `x` + the first of the `Zombie` tuple, which is the offset. This will give us the unit `x`, where we can check against the avatar `x`.

The animation techniques for the `Zombie` moving enemy unit are the same as the ones used for the avatar, as we have extrapolated what we have done to animate our avatar to animate this new `Unit`.

3.1.5. AVATAR

Figure 4, our dear hero, Jack O'Lantern

Our avatar implementation has the `type alias`

```
type alias Model =
    { x : Float
    , y : Float
    , vx : Float
    , vy : Float
    , dir : Direction
    , hp : Int
    , speed : Speed
    , invincible : Invincible
    , score : Int
    }
```

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. `Avatar`'s update section 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. The logic for collisions is to determine if the avatar was currently on a platform, and if it were, to use `case` to deduce the unit's type and see if it was colliding with it. We could then 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.

3.1.5.1. ANIMATION

Unfortunately, our attempts to get smooth animations with an easy fix were futile. Using smaller file sizes for the animations through the compression of frames, size, and colour did not improve the jittery animations enough for it to be considered smooth, and there was no clean way to preload or cache the animation assets.

When observing browser performance of an early game build, we noticed an unavoidable lag at times when a new avatar image was requested (when the character went from standing to walking as the result of an arrow key press, for example) that had not been cached: loading the image, even when hosting the application locally, took longer than our frame refresh rate.

As Elm does not have mutable state or persistent objects, pre-loading or caching avatar frames for future use in the usual sense was not possible. We attempted to cache the needed frames using eagerly evaluated but unused identifiers, but this approach was unsuccessful. We ultimately arrived at a somewhat inelegant, but very successful workaround: all character frames would be continuously cached directly by the avatar view module. We used the `Graphics.Collage.Group` function to store a list containing all avatar animation frames for all avatar rendering, but ensured all frames aside from the appropriate current frame were rendered with an opacity of zero.

The other examples of high-complexity "game-like" Elm applications we consulted in our research generally did not make use of large images or other costly-to-load assets, and we suspect this is because Elm does not have straightforward support for using such assets in responsive game animation.

Doing so would be beyond the scope of our project, but it would be interesting to attempt a similar game that primarily made use of simpler natively-rendered graphic assets and compare both the concordance of Elm to these different animation approaches, as well as benchmarking comparative performance.

3.1.5.2. PHYSICS AND THE GAME LOOP

On top of having smooth animations, we also wanted to ensure consistent and stable gameplay. Our criteria for a consistent gameplay experience is that the avatar moves at the same speed and is governed by the same game physics regardless of the browser. Many of the challenges that we faced attempting to reach this goal were caused by our reliance on the browser event loop.

When implementing the game loop, it made sense to use Elm's `AnimationFrame` library to subscribe to the browser's natural render rate. This guarantees that we only produce one view per browser refresh. As a result, our animation runs as smoothly as possible without computing any extra frames. Our initial implementation updated the positions of every moving component by a constant rate each time the stream emitted an event. However, we quickly noticed that there was a problem: the average refresh rate between browsers is different. On Chrome, Jack O'Lantern moved significantly faster than he did on Safari. Even more troubling is that within the same browser, the refresh rate is inconsistent. `AnimationFrame.diff` (a subscription to the time diffs between animation frames) returned values from 10ms to 100ms while running on the same browser. This meant that the avatar could appear to jerk forward or suddenly slow down whenever the elapsed time was on either extremity. Therefore, we needed to implement extra logic to stabilize and smooth out the animations.

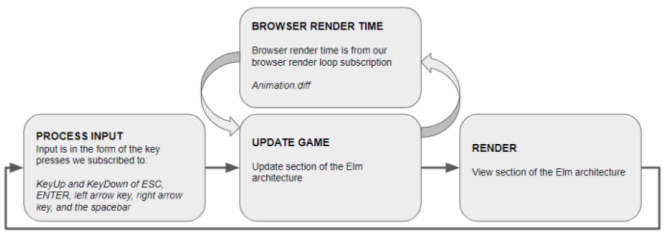
BROWSER RENDER TIME
Browser render time is from our browser render loop subscription
Animation.diff
PROCESS INPUT
Input is in the form of the key presses we subscribed to:
KeyUp and KeyDown of ESC, ENTER, left arrow key, right arrow key, and the spacebar
UPDATE GAME
Update section of the Elm architecture
RENDER
View section of the Elm architecture

Figure 5, our game loop

In order to deal with the inconsistencies of browser event loops, we used common game loop strategies that deal with variable time steps. The update function continued to move the scene forward by a constant rate. However, we now treat each application of the update function as moving the scene 15ms into the future. Instead of only running the function once per `Animation.diff` event, it is run as many times as necessary to catch up to the elapsed time. To give you an example of how the code works, let's say for example that the elapsed time from `Animation.diff` is given as 30ms. The update function increases the scene by 15ms each time it is run. So, in this case, we run the update twice before passing the model on to render. Everything looks great because our scene is now exactly 30ms in the future. However, there is still a slight problem. Let's say that the elapsed time is actually 40ms. We run update twice and the scene is now 30ms ahead but we now find ourselves in the "middle of a frame" because the real time is 10ms ahead of us. In order to fix this, we run the update function one last time, passing in a “multiplier.” The “multiplier” is given by dividing the leftover lag (10ms in this example) by the milliseconds per update (15ms). This increases the entire scene by our leftover milliseconds and puts us 40ms ahead, exactly where we want to be.

After making these improvements to the game physics, we were able to receive a consistent experience across browsers. Faster browsers would continue to render smoother animations, but the speed at which the character moves no longer varies.

3.2. KEY COMMANDS

From our prototype, we had 3 key commands implemented for gameplay, to move left, right, and jump. This proved to be very easy, through Elm’s subscription model to handle user input. In our avatar’s model, we declared a tagged union type

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

in which we have now added 2 more key commands to the game, `Escape` and `Enter`. In our main update 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.

Using the union type made the minor extension of `Escape` and `Enter` extremely straightforward. If we were to embark on a more ambitious extension or refactoring to game input, for example modifying for a touch-based interface, having the conceptual input types separate from their precise IO mechanics would prove invaluable.

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 compiler errors proved to be extremely helpful during the refactoring process. The documentation on various Elm packages have also been convenient to access and use, and we have not needed to investigate any Elm debugging tools.

We tested the full game build and animation performance using the Chrome developer console for recording over time, which helped us pinpoint sources of lag and proved generalizable enough to help us solve cross-browser animation problems.

4. IMPLICATIONS OF PROJECT

In this section, we will discuss what our project has uncovered about the usage of Elm for the purpose of game development, highlighting several advantages and disadvantages of this particular application of the language.

4.1. ADVANTAGES OF ELM FOR GAME DEVELOPMENT

Working on this project, the advantages for Elm for this purpose was apparent. The usage of declarative programming soon became quite natural, and readability was never an issue. We greatly enjoyed programming in Elm with its clean, readable, and intuitive syntax, which allowed us to produce uncluttered code that was easy to refactor. Refactoring was made incredibly easy with Elm’s compiler. Elm was also easy to pick up, and the clean and straightforward list manipulation and pipe operators made potentially knotty state-to-state transformations glass-clear.

The extensibility from the prototype was also a huge advantage, as no major refactoring that could potentially break other areas of the application had to be done to add a single feature. This was also aided by Elm's tagged union types, which yielded a lot of reusability.

The enforced modularity due to the Elm architecture is an advantage we would like to highlight, as multiple teammates could work on different features, and knew exactly where they would have to make changes. Changes did not need to propagate to unnecessary areas of the application, which heavily reduced the risk of breaking the rest of the application.

A key advantage that this project demonstrates about Elm and functional programming for reactive applications is how it can allow for clean event handling, avoiding the messy use of callbacks. It is also important to note that throughout our project, we never had any problems with performance, which may or may not be attributed to the usage of the Elm language and its virtual DOM.

Overall, we can conclude that the combined benefits of Elm being used for this particular application makes it suitable.

4.2. DRAWBACKS OF ELM FOR GAME DEVELOPMENT

Though Elm's error types are meant to enforce added safety and robustness, they were occasionally frustrating to deal with, as it required us to check invariants that we knew could never be violated.

Such strong type enforcement at times resulted in us writing what felt like odd, unnecessarily complicated, and unnatural code. For example, it was more complicated than expected to batch-convert the list of unused avatar frames to a collage group because doing so would have required a map function comfortable with functions that do not take and return identical types.

Additionally, Elm's subscriptions were straightforward and easy-to-use, but didn't allow for overall control. For instance, when our avatar collected a time limit collectible like invincibility or a speed boost, the model would not update until there had been a key up event called. Therefore, if a player continued to hold down a key while collecting one of these collectibles, the intended effect wouldn't take place until the player released the key. This was difficult to debug or fix in Elm since it isn't easy to engage at a low level with a high level abstraction like subscriptions. This seems to be a trade-off with Elm that could be a weakness in advanced game development. Although the high level abstractions allow for ease of development out of the box, there's a lack of control over the finer details that other languages like C++ offer to the developer.

SUMMARY

In this report, we have detailed our final project in terms of what we have achieved, the process we undertook to reach the final goal, and the implications of our project.

While Elm has many features that make it easy to learn, and is particularly easy to use for collaborative work and to develop an application iteratively, the higher-level abstractions present in much of Elm's core language and architecture make fine tweaks difficult and required some odd ad-hoc solutions. Altogether, game development in Elm was a rewarding exercise, but Elm may be better suited to other types of web applications than game development.