Halex

Final Project Document
Instructor: Professor Tian Zhao
Author: Jeffery Jerome Meverden III
Course: COMPSCI 657 – 203 Functional Programming

Contents

Introduction

The following is the design document for the creation of Halex, a video game written in the functional programming language Haskell. This compendium evolved throughout the realization of Halex. Section I describes the intent of the project. Section II motivates the project with a solution to the problem of creating a video game in a purely functional language. Section III defines the blueprint for engineering the software. Sections IV, V, and VI are progress reports on the three phases of development. Section VII concludes the document with a retrospective meditation on the project.

I. Proposal

1. Thesis

I propose to make a video game in Haskell. The game will be a top-down exploration game in the style of the arcade classics. The genre will borrow from Soulslike Metroidvania as players explore to find power-ups that enable triumph over enemies of increasing difficulty. The game will culminate in a boss battle that the player cannot overcome without building a reserve of strength.

I will use the Simple DirectMedia Layer library to implement the game. The SDL2 library will allow for processing input and outputting graphics and audio. The SDL-TFF library can enable rendering text on a user-interface. Meanwhile, the SDL-Image library will support loading sprites. I will employ Monads to achieve the simulation of a game world. The IO and State Monads will be indispensable. I will also devise several Algebraic Data Types to represent objects in the game.

2. Novelty

My project is novel because there are few games written in Haskell, or functional languages in general. In my research, I discovered Keera Studios, a game development atelier whose achievements include several demonstrative games written in Haskell. Their titles are akin to mobile games in their simple and accessible style. A more daunting challenger arrives as Incoherent Software's *Defect Process*, a commercial game also penned in Haskell. Aside from these examples, most games written in functional languages are incomplete tech demos or rendering engines that see paltry use, such as LambaCube 3D.

In my literature review, I discovered the game-development journal of Ashley Smith, who wrote several articles on her experience blazing the trail of engineering games in Haskell. She opens her essay "Of Boxes and Threads" with the statement, "Making a game in Haskell is a pioneering process. Despite the fact that there's a page on the wiki and a full subreddit dedicated to the purpose of making a game in this beautiful language, not many people have actually succeeded making anything close to what current game developers can already achieve." Regarding the gallery of Keera Studios, she writes, "I

don't believe there's enough there to prove anything regarding the power of Haskell in games." She concludes a related article with her encouragement, "I want to see indie developers start embracing the power of high-level languages to make development easier on their already-difficult lives. If people can make cool games in JavaScript, then why not Haskell?" I consider her inspiring blog to be the launchpad for my project. Her oeuvre serves as an explorer's journal that I will follow into an obscure new world.

My investigation of Haskell games has not yielded a work quite like the project I intend to develop. My game will be unique in that it will be a top-down action game, distinguished from its peers by its genre and style. Furthermore, my project will be set apart by its completeness: the game will not be a demo, but rather a full game with a definite beginning and end.

3. Utility

My project will be useful because it will provide an example of how games can be made in Haskell, which can be studied by developers interested in stretching beyond the bounds of imperative programming. Because the menagerie of Haskell games is nearly vacant, even a simple game can serve as a brick in the wall that builds toward popularizing functional game development. Though my project may not be superior to others, its uniqueness will help diversify the population of Haskell products.

The utility of my project is bolstered by its concrete nature. Rather than pursuing a purely academic or theoretical goal, I plan to build a piece of application software that can be enjoyed by any user no matter their background. This plan is further useful to myself because it will help me improve as a game developer. Understanding my favorite subfield of Computer Science from the new and often perplexing lens of functional programming will help me make better games and better appreciate this style.

4. Difficulty

This odyssey will be wrought with challenge because functional programming is incompatible with the dominant design philosophy of video games. From a ludological perspective, a game is simply a state machine that changes given user input and its own internal logic as the player flows through the game unto either a win or loss condition. Game programs are almost entirely composed of side-effects, often emitted from a game loop. In Haskell, there is no mutable state, no looping, and no true flow of control like that of imperative languages. To build a game in Haskell, I must adopt a new architectural paradigm. This will involve recursive functions that accept the game world and return a new world that represents the updated state.

On a Stack Exchange forum post, Haskell game developer Jake McArthur describes the challenges of our shared endeavor, "There are not many great libraries for making games in Haskell, and not many Haskellers write games in it, so it's difficult to find resources and help on this matter." A drought of resources is an issue I have struggled with in this course. However, I have found this hub page on the Haskell Wiki that will guide me to many useful tutorials.

Another functional game developer, Claudia Doppioslash, writes on Quora, “There is very little overlap between people who know Haskell, and people who know how to make games at a professional level.” Although there is an intimidating divide between Haskell and game development, there is also an exciting opportunity to bridge the gap. I believe my project will be a small but positive step for game developers and Haskell.

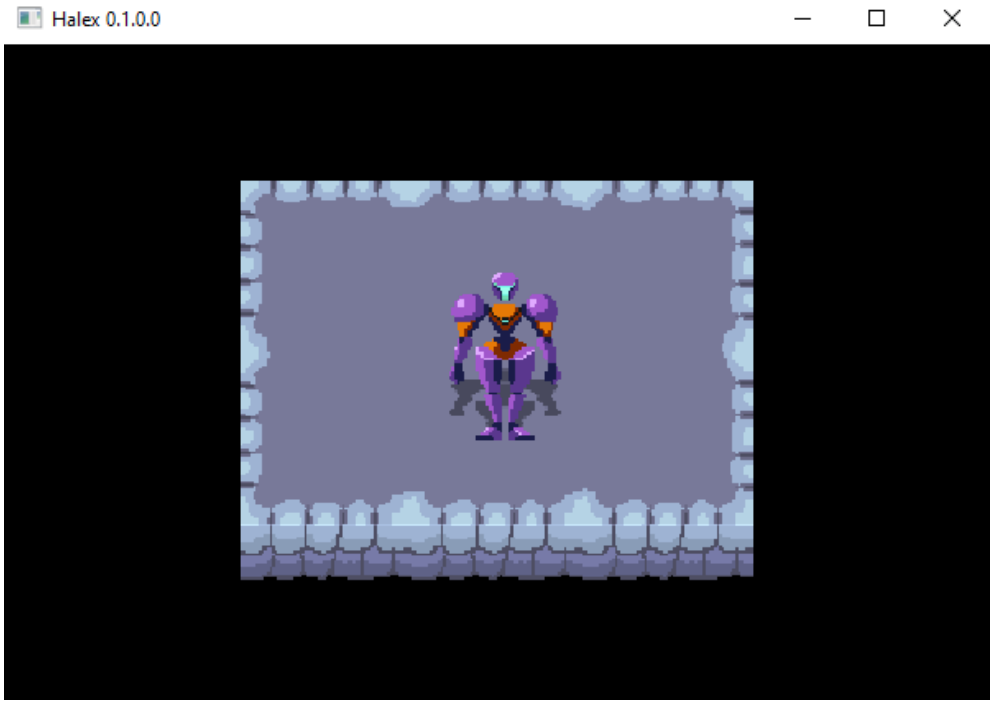
5. Achievability

To maintain a realistic goal, the scope of this project will be limited. The game will showcase simple core mechanics that may be expanded in the future, but are small enough to be implemented feature-complete given a brief timespan of development. Although there are not many Haskell games, there is a bounty of libraries and tutorials written by brilliant programmers upon whose shoulders I may clamber to realize my design.

When I began to study functional programming this semester, I wanted to see what games were made with Haskell, and was surprised to find so few. Game development has always been my most treasured aspect of Computer Science and also the subfield I am most confident in. I believe I will attain a strong respect and admiration for functional languages if I can complete this project in Haskell. I may fail, but if there was ever a programming challenge that I could overcome, I would want it to be this.

II. Motivation

1. Example Description



Our heroine stands resolute atop a set of ground sprites.

For my final project, I have proposed to develop a video game in Haskell. I have elected to craft a 2D top-down Metroidvania Soulslike that emulates classical arcade and home console games. For my example, I have taken a screenshot of my current progress. Above, we see the game window with several sprites rendered together to create a scene of the player character standing on a stone platform.

This example is more video than video game because it lacks any form of user agency. However, it helps demonstrate a powerful side-effect: the rendering of graphics. We can create sprites and draw them in a way that represents a game world and a player character to inhabit that world. My approach to solving the issue of building a functional game program is to create a function that accepts this entire world as the game state, considers user input, and then output a new world that reflects the changed state. This recursive creation of an updated world begets the illusion of life: the protagonist marching about in response to the player's orders.

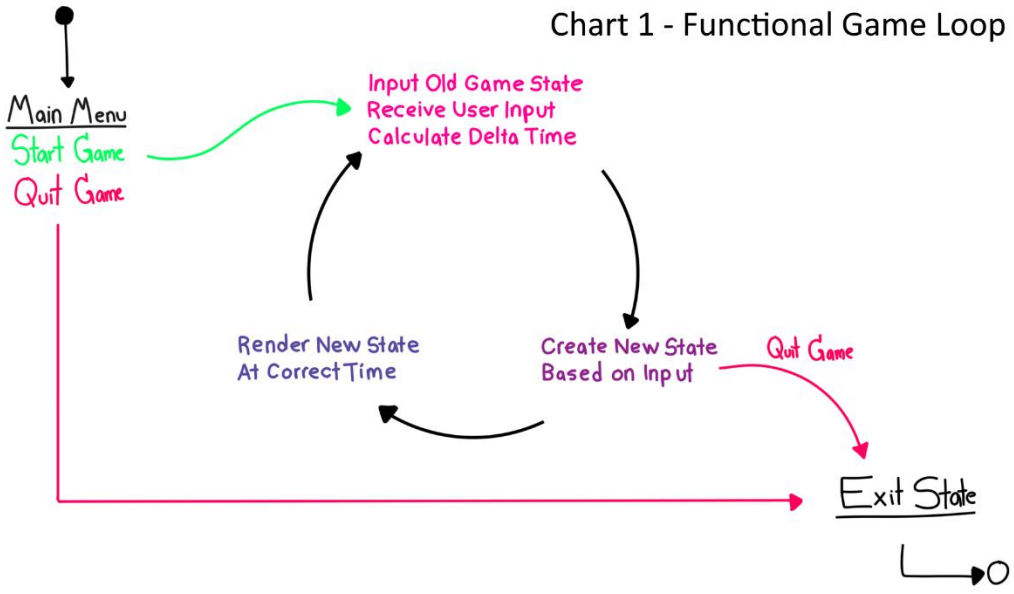
My initial design employed the Simple DirectMedia Layer library (SDL2). However, after further research and numerous failed attempts to install the library on my machine, I have opted instead to use the Gloss library with JuicyPixels to process images and user input. Gloss offers a simpler API than SDL2, though it

lacks support for playing sound. JuicyPixels, and its extension Gloss-Juicy, will allow me to load PNG files for sprites that I create.

In my investigation of the Gloss library, I have discovered more Haskell games that I was previously unaware of. There is a pong clone, a platformer game, and even a fan-made recreation of Sonic 2, all developed in Haskell with Gloss. These titles will offer abundant examples for me to follow when developing my own game.

However, I maintain that my approach is still novel because I have not yet discovered any top-down, sprite-based Metroid clones such as the project I plan to release. I also hope that my example will be useful to other game developers interested in pursuing functional game development. I will release the finished project on GitHub so that other programmers can easily download the source code for their own analysis.

2. Architectural Diagram



This diagram is based on the schematic of a functional game by Ashley Smith (see References).

In Chart 1 above, we can see this overarching game loop function illustrated. The game begins with the main menu, and can be launched with a Start Game option. Unlike an imperative game loop that simply updates variables whenever they are altered by certain actions in the game (such as the player character's x and y-coordinates when the player moves), the functional game loop creates a new world every frame that exhibits all of the changes to the old world conducted in the previous frame.

In Chart 2 to the right, we can see the composition of the game state. This data structure will be divided into sub-structures for entities in the game, the game world, and game logic.

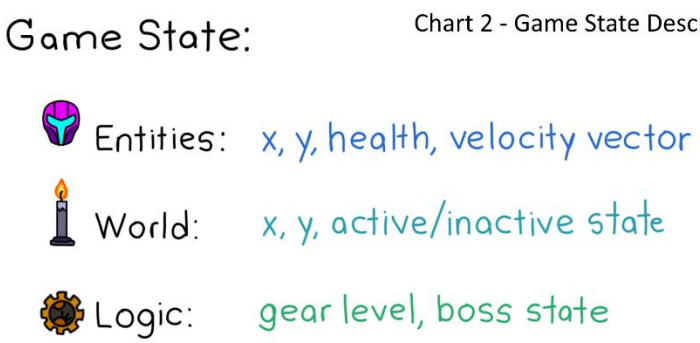


Chart 2 - Game State Description

Entities include the player character and enemies. These objects can move about the game world. The world category references ground objects and environmental scenery. I have included an active/inactive field to reference doors that can be open or closed. Logic envelops abstract states like the player's current level or the status of the final boss. Game logic is also necessary to determine if the player has achieved the win or loss condition.

3. Existing Approaches

The majority of the Haskell games that I have studied thus far have rendered images in a manner similar to the source code presented below. As part of the IO action of the main function, an image is loaded and takes on the type Maybe Picture. This data type may be a picture, or it may be nothing in the event that the file was not found. In the case that the program receives Just Picture, it can be displayed in the window. Multiple pictures can be drawn in the window and given translations to change their position onscreen. The dominant approach to creating game characters that can move around the game world is to translate the character sprite based on user input (or simple AI if the character is an enemy).

```haskell
module Main(main) where

import Graphics.Gloss
import Graphics.Gloss.Juicy

window :: Display
window = InWindow "Halex 0.1.0.0" (600, 400) (60, 40)

background :: Color
background = black

main :: IO ()
main = do

    demo <- loadJuicyPNG "Assets//Images//Demo//Demo.png"
    case demo of
        Just spr -> display window background (pictures [spr])
        Nothing -> print("Loading failed")
```

This code was used to render the example demo depicted on page 6.

The major limitation to this approach is that the game character will not have an animated sprite. Most of the Haskell game characters that I have seen are depicted as static images. In Claes-Magnus Berg's *Gloss Game*, the player character is a small sprite with no animation. Andrew Gibiansky's brilliant *Pong* tutorial loads no images and draws only primitive shapes. Morgen Thum's *Lambda Heights* humorously allows players to pilot the Haskell logo itself. The only games that I have found featuring true sprite animation are Incoherent Software's *Defect Process* and Brian McKenna's *Sonic 2* recreation, both of which are restricted to 2-directional platformer sprites.

4. Functional Solution

To achieve an animated player character, I will draw a series of sprites that can be strung together to form idle and walking animations for the character. These animations will be drawn in the four cardinal directions so the player can walk north, south, east, and west. Each sprite must be loaded into the program using the Maybe Picture type to handle any potential file error. The program will feature data structures similar to arrays, one array for each type of animation, with each element of an array containing one sprite (or one frame of that animation).

In the game loop, we will draw the character using the current frame of the animation the character is currently performing. For instance, if the player presses a key to move right, the drawing function will pull frames from the Walk Right array. Each time the game loop function recurs, the following frame in the Move Right array will be rendered, causing an animation to occur. If the last frame in the Move Right array is drawn, then the next game update will reset the cycle to the start of the array. This animation will continue cycle until the player input changes. This can be achieved with recursive functions on lists.

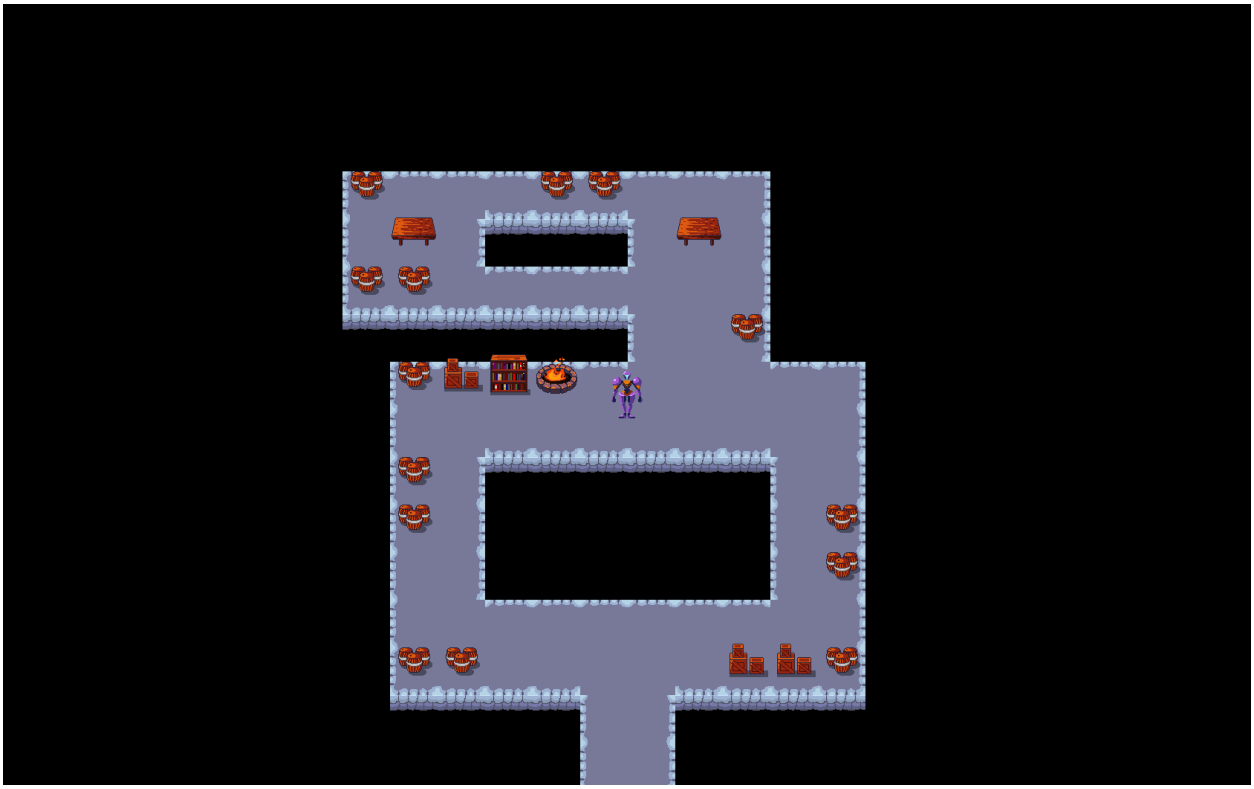
5. Solution Comparison

It seems that the market of games written in Haskell is evolving as a microcosm of the video game industry at large. Pong and Pacman have been created. I have seen a roguelike implementation. Platformers have been introduced. I hope to emulate a classic game style so that I can contribute to this evolution. One of the major milestones in the road these Haskell developers are retreading is the advent of the animated character.

Although there is more to making a game than just animating characters, I believe that my solution for animation will be a boon to the immersive nature of the game and allow for a greater range of player expression and agency than what is achievable in games that do not feature animation. A bounty of information about the state of the game world can be afforded to the player who can appreciate such animations as enemies reacting to being struck, water flowing to indicate a dangerous current, or the principal character striding about in response to the user's commands.

III. Design

1. Method



Our heroine explores her developing world and finds a lot of barrels.

For my final project, I have been developing a top-down 2D Metroidvania Soulslike video game. The great challenge that I encounter is to use the brilliant functional programming language, Haskell, which is lauded for its purity and lack of side-effects, to create the ultimate expression of user-interaction in software, which is a real-time video game. My strategy to overcome this challenge is to use recursive functions to design a game loop that receives the state of a game world as input and creates a new world each frame based on the user's input and the game program's internal logic.

Thus far, this technique has worked well. Our game features a player character that can explore a world generated from a text file. Every object in the game is capable of playing an animation (such as the fires whose flames can flicker). Soon, the level will have an exit to allow for progression. The program will also use collision-detection to prevent the player from walking through objects or over the edge. These mechanics will be integrated into the existing code base, and new animations can be added as well.

2. Testing

The optimal way to test a game program is to play the game and evaluate every mechanic against every scenario. In a small game such as ours, this can be quite simple! For example, to test the aforementioned collision-detection mechanic, the tester must navigate the player character into every type of colliding object from every direction at every speed. After ensuring that the player will never pass through a structure or fall off a ledge, we can be assured this mechanic is complete.

As each new mechanic is added, it can be tested by creating special game scenarios that isolate that mechanic similar to the separation of control variables in scientific experiments. For instance, collision-detection must be tested on a level that features only the terrain, then terrain with structures on top, then a full level with terrain, structures, and decorations. By building up from the simple scenario to the complete game simulation, we will not only isolate possible bugs, but also feel certain that the game will operate well under any circumstances.

To determine that the game is complete, I will devise a set of tests that follow the format mentioned above for every mechanic in the game. I have already tested some features, such as animation, movement, level generation, and the screen following the player character. After collisions, there are only a few more mechanics to implement and test, such as level progression, abilities, combat, win/loss conditions, and a few menu screens. As each facete of the game materializes, I will run tests to maintain a stable program.

The ultimate test for this project will be a complete playthrough of the game in which the tester explores every level, uses every ability, and battles every foe. I will create a video of this test before submitting the final work. Fortunately, this video can also be useful to those who play the game but may possibly get stuck on a level.

3. Development

```haskell
-- Objects that Compose a Level.
data Environment = Environment
    {
        envAnim         :: [Picture],       -- The list of sprite frames for the object's animation.
        envFrame        :: Int,             -- The current frame of the animation.
        envX            :: Int,             -- The object's x-coordinate.
        envY            :: Int,             -- The object's y-coordinate.
        envType         :: EnvType          -- The class of environment to which this object belongs.
    }

-- The Three Types of Environment.
data EnvType = Terrain                      -- Ground that can be walked on but never walked off.
             | Structure                    -- Objects on the Ground that can not be walked through.
             | Decoration                   -- Objects on the Ground that can be walked on.
```

My obsession with proper indentation is the bane of every Java programmer.

To develop this game, I must invoke two major spheres of the Haskell language: algebraic data types and recursive functions. In the screenshot above, we can observe the data type for an Environment object and its associated Environment Type (abbreviated to EnvType). This code defines a tile that can take the form of terrain, a structure, or decoration. Terrain is the ground that characters can walk across but cannot leave, lest they fall off the edge of the world. Structures are objects like barrels and boxes that go on top of the terrain and prevent characters from passing through their solid surfaces. Decorations are items that go on top of the terrain, but they *can* be walked across, such as puddles or pebbles.

According to the fields above, we can see that all environmental objects can support an animation, which is a list of pictures representing frames of the animation, and an integer to indicate which frame will be rendered. This integer is incremented upon each frame of the simulation, which allows all objects to animate during the game.

```haskell
-- Returns a list of Environment objects based on a level file. Works on Level_Terrain, Level_Structure, and Level_Decoration files.
loadEnvironment :: String -> Int -> Int -> [[Picture]] -> EnvType -> [Environment] -> [Environment]
loadEnvironment [] x y anims envtype output = output                                -- Recursion complete. Return the output list
loadEnvironment (z:zs) x y anims envtype output = do
    case z of
        '\n'    ->
            loadEnvironment zs 0 (y - 64) anims envtype output                      -- Move down to the next row of objects.
        '0'     ->
            loadEnvironment zs (x + 64) y anims envtype output                      -- Skip this empty space by moving right.
        _       ->
            let tile = Environment (getItem (hexDigitToInt(z)) anims) 1 x y envtype     -- Create an Environment object at this x and y.
            in loadEnvironment zs (x + 64) y anims envtype (output ++ [tile])           -- Add new object to output list and recur.
```

I pretend not to like C, yet here I am passing in an empty list for a function to fill.

Among the recursive functions that drive the game is the loadEnvironment() function depicted above. It takes a String read from a text file and creates a level of the game out of Environment objects. The main function loads each sprite only once, and passes them into this function to be assigned to Environment objects. This function processes the level String character-by-character, performing pattern matching on the characters and decoding the hexadecimal digits into indexes that are used to determine which set of sprites will be assigned to the current Environment object. For example, the northwest terrain (code 8) is assigned the northwest list of terrain sprites (index 8). A list of Environment objects that compose the level is returned as output.

This development philosophy can be seen in most of the game code. Most resources are loaded in this recursive manner. Similar pattern matching occurs in the processing of user input. Any new game object such as the player character, enemies, or even abstract concepts like directions can be codified into such algebraic data types.

4. Outline

A. Algebraic Data Types

Game – This data type encapsulates the state of the game world and is recreated with each frame. It contains fields for the player character, the level, the enemies, and a set of keyboard buttons that are currently being pressed by the user.

Player – The character controlled by the user. Contains fields for animation, position, and statistics like speed and health.

Direction – The eight cardinal directions are captured in this simple data type.

Environment – A tile that composes the game world. Contains fields for its animation, position, and type.

EnvType – Classifies the environment into three categories, each with unique collision mechanics – Terrain, Structure, and Decoration.

Exit – An object that, when reached by the player, transitions the game to the next level by linking to an adjoining location. Contain a field for what level is loaded next when the exit is reached.

B. Functions

Loaders – Several functions are needed to load the game objects into the game data type, such as the level and the entities within.

Animators – These functions increment the frames of animated objects like enemies and decorations.

Player Control – These functions allow the player character to move and use abilities in response to user input.

Enemy Control – These functions drive enemy characters to navigate the level. I will likely not indulge in any graph-searching algorithms for enemy pathfinding, but will instead make enemy movement as simple as possible. An enemy can move forward until colliding with an object, at which point it deflects in a random direction, attacking the player character whenever it draws near.

Game Logic – These functions will allow the player to use abilities once they are unlocked and progress in the game when certain levels are reached. These functions also govern the win and loss conditions, leading to a victory state or a game over screen.

5. Timeline

Acknowledging the deadline for this project at midnight on May 16th, I look forward to my project's completion with optimism! Most of my time in early April was spent trying to learn about the proper libraries to use for game development and researching how other games were made. However, after I gained a strong understanding of how to build this game, my progress has accelerated greatly! In two weeks, I have developed at least two-fifths of this game. I believe that, given these last four weeks, I will be able to realize this design to completion.

I will be graduating from university very soon and I consider this project to be the most important thing I will create in the course of my education, not necessarily for its academic value, but because game development is an art after my own heart, and I want one of my final acts at this university to be constructing the best game that I can. In light of this, I will give my best effort to finishing it in time!

IV. Phase 1

Date: 4/27/2022

Version: Halex 0.25.0.0

1. Objectives

Per the previous documents, our objectives for the first phase include:

- Create a graphical representation of a game world from a top-down perspective using the Gloss library.
- Implement the functional paradigm of game simulation to let the player explore the game world through the main character.
- Create algebraic datatypes to represent objects in the game world.
- Create a framework to support the animation of any object in the game.

2. Progress

In the game's current state, all of the objectives have been completed. The developer can create a level out of a few text files that describe the objects that compose the level. The game will read the text files and construct the level, allowing the player to explore the environment with a fully-animated player character. The codebase is flexible enough to allow the animation of every object that may be deemed dynamic. The current product is essentially a maze-solving puzzle game with no enemies, abilities, menus, or game-over conditions.

3. Accomplishments

After drawing many sprites in the style of classic arcade games, I can assert that the game is nearly asset-complete in that almost all graphical resources are available to be loaded into the game and rendered in the game world. The only sprites that have not been drawn are those for enemy characters.

A crippling lag issue has been remedied by the optimization and refactoring of the entire codebase. The lag issue was caused by all objects being considered animated even if they only possessed a singular static sprite. The reason the early game was implemented this way was because I was enamored by the idea of being able to bring any object (even the ground terrain) to an animated state simply by including new image files in their respective resource folders. The original build allowed for this, but the excessive use of lists-of-lists and expensive animation functions caused the game to run slowly in larger levels.

The solution to this issue was to separate static objects from dynamic objects. Now that I am aware of specifically which objects ought to be animated, there is no reason to include static objects in the functions and data structures pertaining to animated objects. This separation has alleviated the lag issue significantly, allowing for very large levels featuring many objects to be rendered smoothly. This is exciting because exploration is so key to the entertainment value of the game, so it is good that players will not be confined to miniscule levels to avoid performance issues.

4. Goals

The objectives for the next phase include:

- Implement collision detection to inhibit players from moving through walls or over edges.
- Create exits that transport players to different levels.
- Allow levels to be connected nonlinearly like nodes in a graph to allow for more exciting exploration.

For the game to be considered a true Metroidvania, I planned to implement abilities that the player could discover in the game world that allow for greater exploration (such as a shield that allows one to walk over a field of fire). This positive feedback loop of expedition rewarding further opportunities for expedition is core to the Metroid experience. Furthermore, for the game to be considered a true Soulslike, there must be enemy characters that halt the player's progress with significant challenge until they can be overcome and defeated.

However, these goals will require more time to implement appropriately, and so they will not be considered for the next phase unless the bulleted objectives can be completed and tested first. If time does not allow for the implementation of these final features, I will still endeavor to give the game a satisfying conclusion and construct a codebase flexible and modular enough to allow for future extension.

5. Experience

One of the greatest experiences I have had in the project (and this course in general) is the ample opportunity to master recursive algorithms. With none of the iterative control structures that I am used to in imperative programming, I have been challenged to rely on recursion to an extent unrivalled in my other ventures in Computer Science. I am especially proud of the level generation function I have written that recursively considers each character in a text file, pattern matches the hexadecimal characters contained within, and builds level objects in a grid that correspond to each character. It is very fun to build levels in a simple text editor, and I feel this part of the program benefitted greatly from the techniques of functional programming.

V. Phase 2

Date: 5/4/2022

Version: Halex 0.5.0.0

1. Objectives

Per the previous report, our objectives for the second phase include:

- Implement collision detection to inhibit players from moving through walls or over edges.
- Create exits that transport players to different levels.
- Allow levels to be connected nonlinearly like nodes in a graph to allow for more exciting exploration.

2. Progress

In the game's current state, the second and third objectives have been accomplished. While there is still no collision detection, the game now offers more than a single level. The game features six levels connected together to create a large game world with multiple avenues of exploration. There are currently three major setpieces that can be discovered in the world. Previously, these setpieces were intended to be locations where the player would discover new abilities and encounter stronger enemies. In the more limited sense of the game, however; these serve as landmarks on the player's journey to the final level.

3. Accomplishments

In order to enable the player to travel between levels, special exit ADTs were created and placed throughout a level to connect to other realms. The exit ADT contains an integer to describe which level it leads to, with the first level being Level 1 and the final level being Level 6. This is not only useful for transporting the player to the correct level, but also for spawning the player character at the correct position in that new level. For instance, if the player warps through an exit in Level 1 that leads to Level 2, the player will appear standing near the exit in Level 2 that would lead the player back to Level 1, bringing better cohesion to the player's travels.

To support multiple levels, the game loads all resources upon launch, and cycles between sets of environments whenever the level changes. New functions were created to handle the changing of levels, which can be appreciated in the source code that is now available on GitHub.

4. Goals

The objectives for the next phase include:

- Implement collision detection to inhibit players from moving through walls or over edges.
- Create a main menu and victory screen for when the player reaches the final room in Level 6.

5. Experience

My most recent learning experience resulted in my increased admiration for the functional programming paradigm of functions being considered first-class values. When developing the ability to transfer between levels, I noticed it was necessary to reposition the player character to the exit they just travelled through, as mentioned above. To accomplish this, I employed the functions I had written to move the environment. But instead of moving the game world by a single pixel to represent the player taking a single step, I moved the game world to the exact coordinates that would position the player at the correct exit. When creating the updated game state, I needed to pass in the environment as an argument to construct the new game. Knowing that my movement function returns an environment, I simply passed the movement function itself as an argument for the game environment. The lazy evaluation of Haskell mixed with the first-class value of functions made this strange move possible, and was a boon to the development of player teleportation.

VI. Phase 3

Date: 5/9/2022

Version: Halex 1.0.0.0

1. Objectives

Per the previous report, our objectives for the third phase include:

- Implement collision detection to inhibit players from moving through walls or over edges.
- Create a main menu and victory screen for when the player reaches the final room in Level 6VII

2. Progress

In the game's current state, both objectives have been accomplished. The game is now a feature-complete maze-exploration game!

It seems I was too ambitious in my initial description of this project. As mentioned previously, the game cannot be considered a Soulslike because there are no challenging enemies to battle. Furthermore, the game cannot be considered a Metroidvania because there are no special abilities for the player to wield. Like most game developers, I have not delivered on all of my promises. However, I believe this project is still unique because there are no top-down maze-exploration games like this implemented in Haskell.

I hope that this project will bridge the gap between game developers and functional programmers. Because the source code is openly available on GitHub, and the creation of the game was chronicled in this report, this project can be studied by both kinds of Computer Scientists so they may be inspired to pursue their own projects. For this reason, I consider this project to still be useful.

Perhaps the most important achievement of this project was simply the functional implementation of a video game. This game is essentially an interactive state machine which displays graphics and accepts user and file input, and it was written in a language that forbids changing state, prides itself on avoiding side-effects, and guards against the impurity of input processing. I believe the functional approach to game development is demonstrated well in this project.

3. Accomplishments

The major accomplishment of this final phase was a clever implementation of collision checking. Collision checking occurs in the function that updates the player character. If the player presses the button to move North, but this movement would collide the character with a structure or an edge, the update function forbids this movement.

Boundaries can be outlined in a text file similar to how levels are designed, and collision tiles can be placed anywhere. This would allow for invisible barriers and secret passages through seemingly solid surfaces, which would create very crafty mazes to be explored. However, I did not include any secrets in the actual game world.

To ensure the player character remains fixed at the center of the screen, when the player presses a movement key, the game world itself is translated around a stationary character whom remains at the central origin point. This gives the illusion of the viewport following the character as she walks. However, when implementing the collision tiles, I did not want to waste processing power updating the positions of a huge array of invisible tiles, so I opted for a different strategy.

When the player moves, although the character is always drawn at the origin, the Player ADT has its x and y coordinate fields altered. These coordinates are used to check if the character is intersecting with a collision tile. As a result, we do not have to update or move any of the collision tiles because the player character will come to them rather than the collision tiles moving to the player. This saves much processing power and does not disturb the visuals of the game, because the collision tiles are invisible.

4. Future Work

With this final report, Halex is now a feature-complete functional video game, and has passed into version 1.0.0.0. Further development may be conducted by myself or another programmer outside the scope of this project. My implementation may serve as the springboard for a different video game. The game assets could even be used in a similar game written in a different language. There are many paths that Halex may take from here, and I feel the artwork presented at the conclusion of the game reflects this sentiment well.

Opportunities for further work include:

- Create enemies that challenge the player's journey.
- Create abilities that the player can discover and use to aid in exploration.
- Create a larger game world featuring more maps.
- Design more types of structures and decorations (the engine supports 16 of each).
- Upgrade the level editor from text-based to image-based to levels can be drawn in bitmaps.
- Upgrade the codebase to rely more heavily on monadic solutions.
- Present a more detailed story for Halex to create a compelling player experience.

VII. Conclusion

The question awaiting at the end of this endeavor asks about the advantage of writing a video game in a functional language. After completing this journey, I believe there is no advantage. Because a video game by nature is opposed to the pillars of functional programming, it seems the functional approach offers more inconvenience than advantage. While this project was rewarding, it was also immensely challenging in ways that felt unnecessary. An imperative language is better suited to this task.

One may argue that there are always easier ways to make games. With the advent of powerful and popular game engines like Unity, Unreal, and CryEngine, people with budding programming experience can realize their vision through beautiful games that can be implemented with ease. Yoyogames, the company that maintains the GameMaker engine, has strived for people to be able to make games without writing a single line of code. This was a promise vaguely realized when I used GameMaker as a child, but in recent times their vow has become greatly potent. Making games cannot be easy enough.

But programmers are like artists in that they do not pursue ease, but rather the passionate and often painful realization of a vision. There is a concept in game design dubbed “restricted creativity” which teaches that artists who choose challenging or limiting techniques will create better work than what would be possible after embracing the latest luxuries. This is why many people suggest that games were made better in the past compared to modern times.

In the early days of game development, memory limitations forced background music to have only a few notes in polyphony, which resulted in the iconic chiptune genre. Sprites could only have a few colors, resulting in the timeless designs of characters like Mario and Donkey Kong. Even the creators of Metroid faced tremendous difficulties as they worked in the 1980s, and begat a work of genius that changed popular culture forever. The greatest gifts may arrive in the smallest packages.

That is why I am proud to have furthered the unity between game development and functional programming. Although my contribution is miniscule given this grand perspective, Halex remains a magnificent learning experience that has forged me into a better game developer by imposing limitations and compelling my creativity to outwit my restrictions. As I developed the game, so too did the game develop me. And I am all the more grateful to have been given this opportunity. Thank you.

References

1. Andrew Gibiansky. “Making Your First Haskell Application (with Gloss)”
https://andrew.gibiansky.com/blog/haskell/haskell-gloss/

2. Ashley Smith. “An Introduction to game development in Haskell using Apecs”
https://aas.sh/blog/making-a-game-with-haskell-and-apecs/

3. Ashley Smith. “Of Boxes and Threads: Game Development in Haskell”
https://aas.sh/blog/of-boxes-and-threads/

4. Brian McKenna. “Sonic 2 in Haskell: Sprite Animation!”
https://www.youtube.com/watch?v=ZpgnAWsuhV4

5. Claes-Magnus Berg. “Making a Small Game with Gloss”
https://blog.jayway.com/2020/11/01/making-a-small-game-with-gloss/

6. Haskell.org. Gloss Library
https://hackage.haskell.org/package/gloss

7. Haskell.org. JuicyPixels Library
https://hackage.haskell.org/package/JuicyPixels

8. Haskell.org. Gloss-Juicy Library
https://hackage.haskell.org/package/gloss-juicy-0.1

9. Haskell.org. SDL2 Library
https://hackage.haskell.org/package/sdl2

10. Haskell.org. SDL TFF Library
https://hackage.haskell.org/package/SDL-ttf

11. Haskell.org. SDL Image Library
https://hackage.haskell.org/package/SDL-image

12. Haskell Wiki. “Game Development”
https://wiki.haskell.org/Game_Development

13. Incoherent Software, LLC. *Defect Process*
https://store.steampowered.com/app/1136730/Defect_Process/

14. John Slap. “Make You A Roguelike in Haskell for Greater Good”
https://www.youtube.com/watch?v=5URsWAlkT_0

15. Keera Studios. “Haskell Game-Programming – Examples”
https://github.com/keera-studios/haskell-game-programming/wiki/Examples

16. LibSDL.org. SDL2 Library
https://www.libsdl.org/

17. Morgen Thum. *Lambda Heights*
https://github.com/morgenthum/lambda-heights/blob/master/screenshot.png

18. Quora via Claudia Doppioslash. “Is Haskell used in video game programming”
https://www.quora.com/Is-Haskell-used-in-video-game-programming

19. Stack Exchange via Jake McArthur. “What are the challenges and benefits of writing games with a functional language?”
https://gamedev.stackexchange.com/questions/374/what-are-the-challenges-and-benefits-of-writing-games-with-a-functional-language

20. YouTube via Kaviolalainen. “LambdaCube 3D (Haskell rendering engine) – Quake 3 example – 2012-09-08”
https://www.youtube.com/watch?v=JleoASegUlk

21. GitHub via Jeffery Jerome Meverden III. “Halex”
https://github.com/JefferyMeverden/Halex