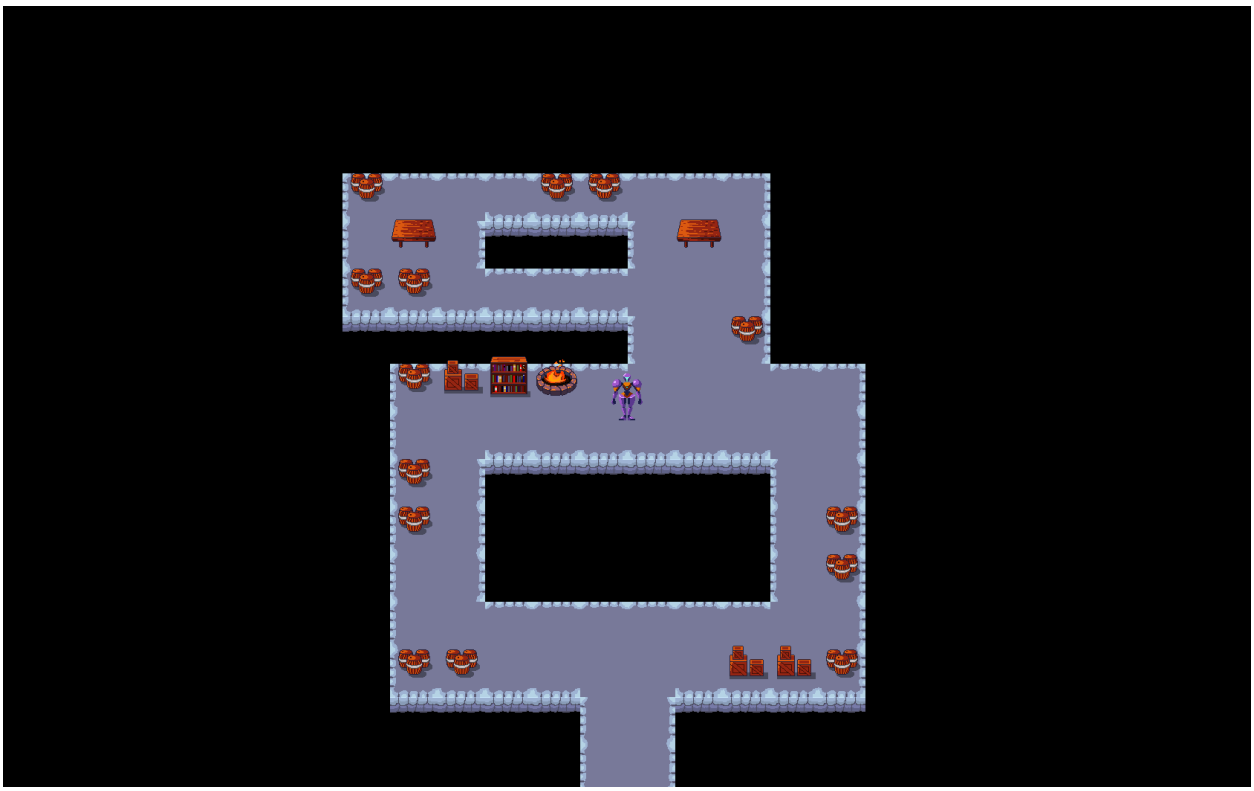
Jeffery Meverden
COMPSCI 657 – 203
Functional Programming
Final Project Design
4/19/2022

Final Project Design

1. Design



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.

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 function 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 and include it with my project files 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.

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.

Outline

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.

Enemy – Enemy characters that threaten the player's progress. Contains similar fields as those in the Player type.

EnemyType – Classifies the enemies into three categories, each with unique abilities – the Heathen, the Hexon, and the Heretic.

Projectile – Dangerous particles launched by characters that deal damage upon contact. Contains fields for speed, position, and a timer that determines the projectile's lifespan.

ProjectileType – Classifies the projectile into three categories, each with unique sizes – small bullets thrown by the player, medium fireballs cast by the Hexon, and large meteors hurled by the Heretic.

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.

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.

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!