Turbohiker design report

In this short document I will be explaining my design of the game and the decisions that led up to it.

Game class

It's simpler to begin from the very beginning, I will be explaining things deeper as we go. Let's start with the game class, since this is the class that ties everything together. This class has a whole lot of responsibilities, maybe too many even. Firstly it is responsible for the game loop and everything that ties into it, such as calculations of timers, the fixed delta for entity updates etc. Secondly it handles the terrain generation, admittedly terrain generation could've been contained in its own class, but at the time this seemed the most intuitive place to put it in.

In this class I instantiate pretty much everything, from the score display to colliders to other hikers. However I don't believe that it's a bad thing per se, the class only does simple things.

World class

The game class does the most heavy lifting, which is why this world class is more intuitive and logical in what it does. Namely it provides collision control and resolution based on the AABB collision detection algorithm. It also does some cleanup of the game world, for instance the entities and tiles which have already been passed by the game view get deleted there. It also handles yelling and everything that has to do with it.

Another big part of its responsibility is keeping track of everyone's position (also whether an entity has reached the finish) and notifying the respective score observers.

Entity class

This class acts as an interface from which every other entity and the world is derived, it provides basic functionality such as move, setters and getters, etc.

It is a derived class from another super class called Observable, I guess the name Observable should make it clear as to what it means.

GameAI class

This class is used to handle movement of the moving hiker and the racing hiker. It takes into account a square of 2x2 units around the entity which can be used as some sort of sight. Based on the location of the entity and the entities inside this 2x2 space, a direction in which the entity should move can be inferred. If there are no obstacles or other entities within this space, then the entity can decide to either speed up or slow down.

All AI-controlled entities have a rubber banding effect to their movement such that they don't stray away too far from the player. Otherwise the player would have almost no chance of winning.

These entities can also yell although no sound plays for that, I found it rather annoying.

Player and competitors

The competitors get their input from the GameAI class whereas the player is controlled through keyboard input

The competing entities (player included) can move freely inside the game view. And only the player can decide how fast the world “moves”.

All competing entities are allowed to yell to try to avoid an obstacle (and as such big point deductions). The AI-controlled competitors have a random chance of attempting to yell instead of dodging an obstacle. This is done because it is not really favorable to yell instead of dodging, especially for an AI which can dodge obstacles with ease.

The player however can always attempt to yell with a press of a button. Different obstacles have different behavior when they get yelled at. Static obstacles (small ponds in my game), have 50% chance of getting a bridge across them if they’re yelled at. Moving obstacles (skeletons) also have a 50% chance of successfully getting yelled it, they effect however is different. Namely they simply slow down by a factor of two.

Score observers and the leaderboard class

The score observers listen to events (an event is just a simple enumeration) which get sent out mostly by the world and game class when a change is detected there. Based on each event the score of the observed entity gets altered.

The leaderboard class basically ties all the score observers together, paired with each entity is a name which can be edited in the `.ini` file for the player, or inside the code for the other competitors. Based on that we can display at the end of the game. This class is also responsible for saving the high scores to disk and reading them from disk too.

Closing remarks

In the prototype phase of this project I noticed that I had been switch assets very often, it was quite annoying to change it in the code itself or even when I tried a slightly better solution which was putting the paths inside a header file. That however required a recompilation every time I wanted to test a new asset. So after that I made it a priority to make my game as customizable as possible. Which is why I have a lot of entries in the config file.

Early on in development I faced a couple of problems, one of which was the [-4, 4] x [-3, 3] game world size requirement/suggestion. I couldn’t find an intuitive solution for it no matter how hard I tried. I wanted everything to be nice and round, such as a player having a size of (0.5,0.5) or a tile being (1,1). Working with a fixed game world size just wasn’t as intuitive in my opinion, so I extended the y value domain to infinity. This was very easy to work with. However, the [-4, 4] x [-3, 3] requirement wasn’t just ignored, because at any given time the player can only see that much of the game world, as far as the player is concerned it is how big the world is (I could be loading and unloading entities all the time). This meant that the game would work perfectly on a 4:3 aspect ratio resolution.