Death Ferry Express

Gameplay Overview

Gameplay

Death Ferry Express is a 2.5D racing game that is set in classical Greek mythology. The story revolves around a player who is a ferryman of the underworld whose duty is to carry the souls of the newly deceased across the Styx river. However, there are many other ferrymen that compete against the player for Hades' favor, and it is up to the player to guide him through his perilous journey. The player who wins first place turns out to be Charon, and is awarded the sole distinction of ferrying souls from now on.

Since this is a racing game, the main goal is to arrive at the finish line before the other ferrymen. This can be accomplished by simply traversing the various parts of the Styx river and by avoiding any unnecessary collisions with obstacles.

In addition to simple racing mechanics, there power-ups that can help the player win the game, and these are based on the souls that s/he can pick up during the race. Picking up souls boosts the boat's maximum speed, thereby giving the player an incentive to pick up more souls.

<Projectile Feature Cut>
The player can also use the body of these souls as projectiles to hit other ferrymen, and these projectiles are dependent on the type of soul that was picked up.

These are the 4 types of souls:

- Greedy Soul: Stops the boat hit
- Angry Soul: Gives a quick temporary speed boost to the boat, but cannot steer
- Hungry Soul: Gives the boat double its soul collection radius
- Lazy Soul: Slows the boat hit for two seconds

Controls

Boats will be controlled by either the player, an AI, or a network player.
W, ^ will accelerate the player's ship
S, v (arrow down) will slow down
A, <- will turn left
D, -> will turn right
Space will throw a soul at an enemy boat

Graphics

Using 2.5D with pixel art.
We now use OpenSceneGraph to render the game. Boats and Souls are 3D objects, but the track is still a 2D shape. Boats will be on a fixed Z plane at all times.

Scoring

Score is mainly determined by the time taken to finish the game. In the root menu, the player will also be able to see the personal high scores for each track.
<Score is only determined by finishing place>

Architectural Overview

Physics Engine

Piloting the ships will be driven by the Box2D physics engine. Each physics frame, it will simulate each ship according to the current input state for each ship. The player will be able to change their input object in real time, and the physics engine will act on the update input on the next frame. Network players will send input frames at fixed interval to the host machine, and AI players will calculate their input state (this will be a cheap calculation) at the physics update rate.

The edges of the track will be represented as a set of box colliders by the physics engine (this became edge chains instead), and ships will collide with the walls to keep the ships inside the course.

When ships are hit with bodies, depending on the type, a “status effect” will be applied to the impacted ship, changing some of the physics parameters for that ship.
<status effects cut>
To pickup souls, ships will have a trigger collider that will pick up the soul and put it on the ship if any souls enter the collider.

Track Loader

<Feature Cut>
A parser will load the track information into the physics engine from a text file. Tracks will be comprised of an array of track segments, each having a path that specifies the middle of the track, as well as a path representing the ideal racing line in that segment. Colliders representing the edges of the track can be generated in the physics engine from the midline and a track width parameter.

Graphics Engine

We will use OpenGL to render our graphics. Most graphics will be done with sprite/pixel art.
<Ended up using OSG, and full 3D art>
The graphics will include three main parts: a background, scenery around the player and road, and the road. The background will be static throughout the entire game, the scenery around the player and road will be procedurally generated from static sprites, and the road will have a specific, pre configured look at every point.

AI

Each agent will have a separate AI. An ideal racing line is then specified for each track, and AI boats will seek towards that position plus a delta T on that racing line.

We will implement the masking collision avoidance algorithm mentioned in class to avoid colliding with other boats and players.

Some AI players will have a weak seeking preference towards souls that are on the track, but will try not to deviate from the racing line too much. This preference will be tuned by a parameter and will differ between different AI players, to give each racer a personality. AI players will also throw bodies at boats that enter a roughly conical collider in front of them.

Networking

For a multiplayer game, one player will run the game as a host while the other players on the network can connect to the game and to join the race as clients. The client players send their input data to the host for simulation on the host's machine, and have the host periodically send game states to its clients.

Both the host and the clients will be mainly communicating using UDP. The UDP packets that are being sent from the clients to the server will contain the player's state information. Since UDP does not guarantee the orderly arrival of packets, the server will also maintain a timed buffer to receive the packets. Once the timer expires, the server will then process the inputs according to the sequence number of the packets.

In the case that a packet is lost in the network, the host can just infer the client's input from its previous state. In the case that a packet arrives out of order, the host can just discard that packet.

As for the game state itself, the host will periodically send the game state to its subscribed clients, and the clients can render the graphics on the client side.

In the case that a client drops its connection, it will stop sending its player input. If the host detects that it has been missing at least 20 consecutive packets from a client, the game will pause and wait another 5 seconds for the disconnected client to connect. If the client does not connect, it will be dropped and the game will resume as normal.

In the case that the host drops its connection, it will stop sending its game state packets. Similar to a client dropping its connection, if a client detects that it has been missing at least 40 consecutive packets from its host, it will completely terminate the race.

Input Handling

We will use the SDL library to handle user inputs. As defined in the Controls section, the player can only accelerate, decelerate, steer left, steer right, and launch a corpse. Once these inputs are captured, the boat object will update one or multiple of its states from the following list of states:

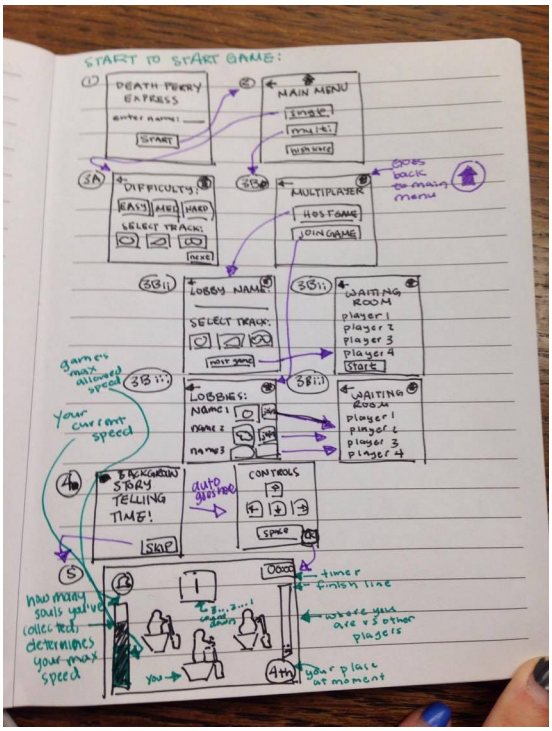
- Player is accelerating / decelerating
- Player is turning left / turning right
- Player shoots a corpse

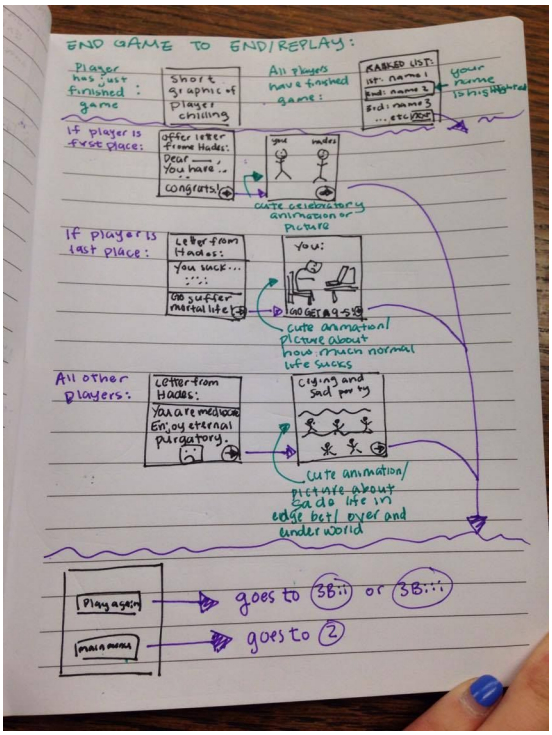
The corpse shooting state is going to be different from the boat steering states. The client player will enter a “firing” state, and will stay in its “firing” state until the firing is recorded in the latest game state. We are implementing this state in this manner because firing a corpse is an important action, and in the case that the user input UDP packet is lost, the client will pester the host to ensure that the corpse will be shot.

This state interface should be mirrored for AI and Network players, and the Physics engine will consume an object that implements this interface.

User Interface

We will start by implementing a simple interface using GLUI based on the diagrams below. If we have time, we can use other libraries to make the user interface more appealing.





Debug UI and Build Scripts

For debugging and testing, we will use Box2D’s built in debug UI, which will draw outlines of colliders in 2d space. We will add vectors for velocity. The AI will also output desired vectors to this debug UI.

Gamestate

The game state information will consist of the following:

- All Player States
  - Current Position (vec2)
  - Current Velocity(vec2)
  - Angular Rotation (float)
  - Angular Velocity (float)

    - Current Soul(int)
    - Number of Souls(int)
- All Pickup Positions (vec2[])
- Race State <This became implicit>
    - List of player positions (int[])
    - Part of Track of each player (int[])

Engineering Overview

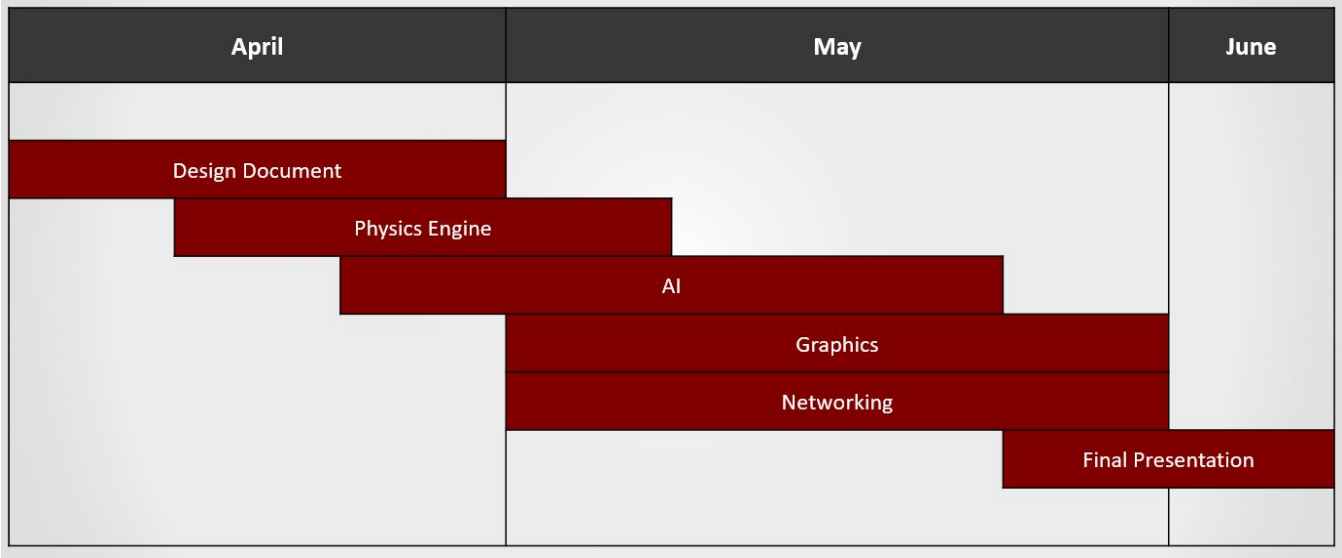
Cross Dependencies

Physics Engine	**None**
Agents	**Physics Engine**
Graphics Engine	**Physics Engine, needs to have something to render**
Debug UI	**Physics Engine**
Networking	**Some input handling and basic Physics**
AI	**Physics Engine, Agents**
Input Handling	**None**
User Interface	**None**

Major Milestones and Presentation Dates

[x] Design Document	**Tue, April 11th**
[x] Game Concept Presentation	**Thu, April 13th**
[X] Basic Physics Engine	**Thu, April 20th**
[X] Status Report Presentation	**Tue, April 25th**
[X] Basic AI	**Tue, May 2nd**
[X] Status Report Presentation	**Thu, May 4th**
[X] Status Report Presentation	**Thu, May 18th**
[X] Final Presentation	**Thu, June 1st**

Gantt Chart


April
May
June
Design Document
Physics Engine
AI
Graphics
Networking
Final Presentation

Group Member and Roles

Jordan Kozmary	**AI, Input Handling, and Physics**
Siddhant Kothari	**AI, Networking, and Physics**
Gianni Chen	**Networking, Graphics, and Fill as needed**
Keely Zhang	**Graphics and Input Handling and User Interface**

Coding Style Options

Preferred Coding Language	**C++**
Naming Convention	**Camel Case**
Curly Braces in own line	**Yes**
No Brackets for single code	**Yes**
Tabs or Spaces	**Tabs**
Common IDE	**No, will take care with makefile/build scripts**
Version Control	**GitLab** (https://gitlab.com/kozmary/game-construction)
Communication Channel	**Slack** (https://game-construction.slack.com)

Sample Art Style