C++ Project: Turbo Hiker

“Turbo Diver”

Features

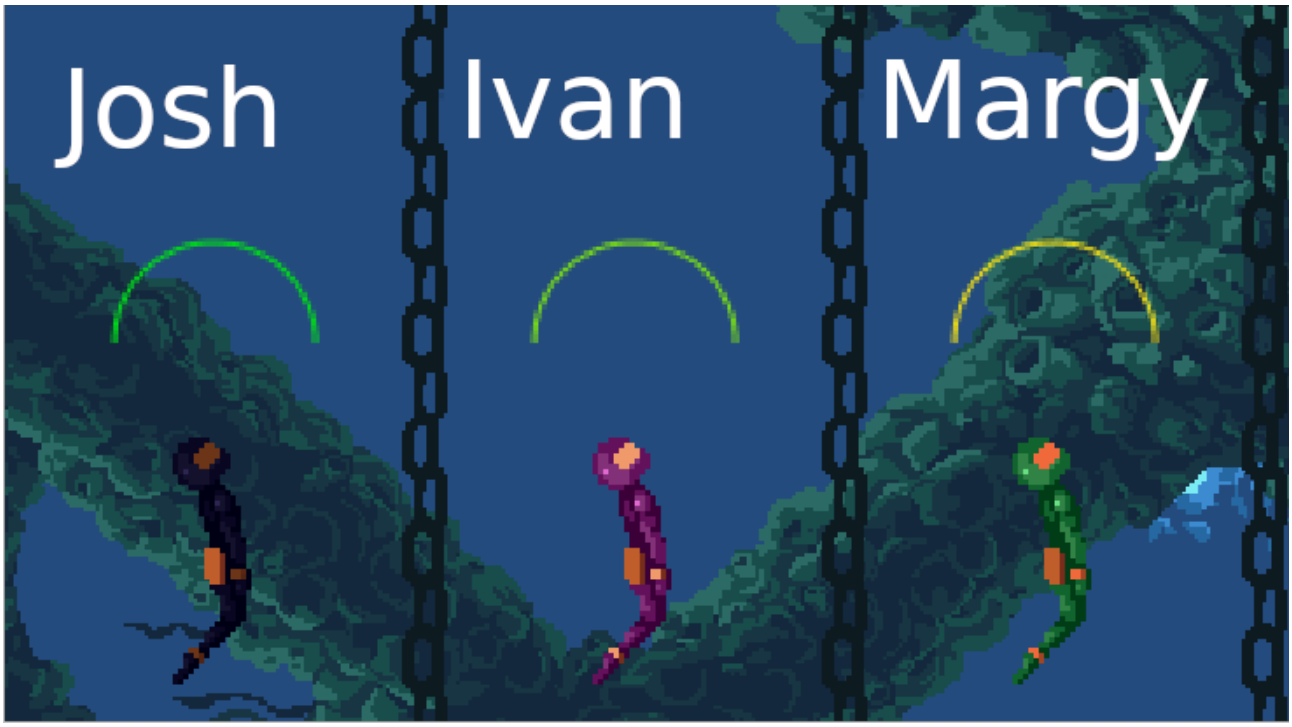
The world consists of 4 lanes on top of a parallax background for the player to freely move in either left or right while they swim forward. Initially they will start out in the most right lane.

The player may also choose to speed up or slow down. The swimming animation and sound effect speed follow the speed of your player.

Additionally the player can cast a sonar beam to communicate with the different fish that they will encounter on their dive.

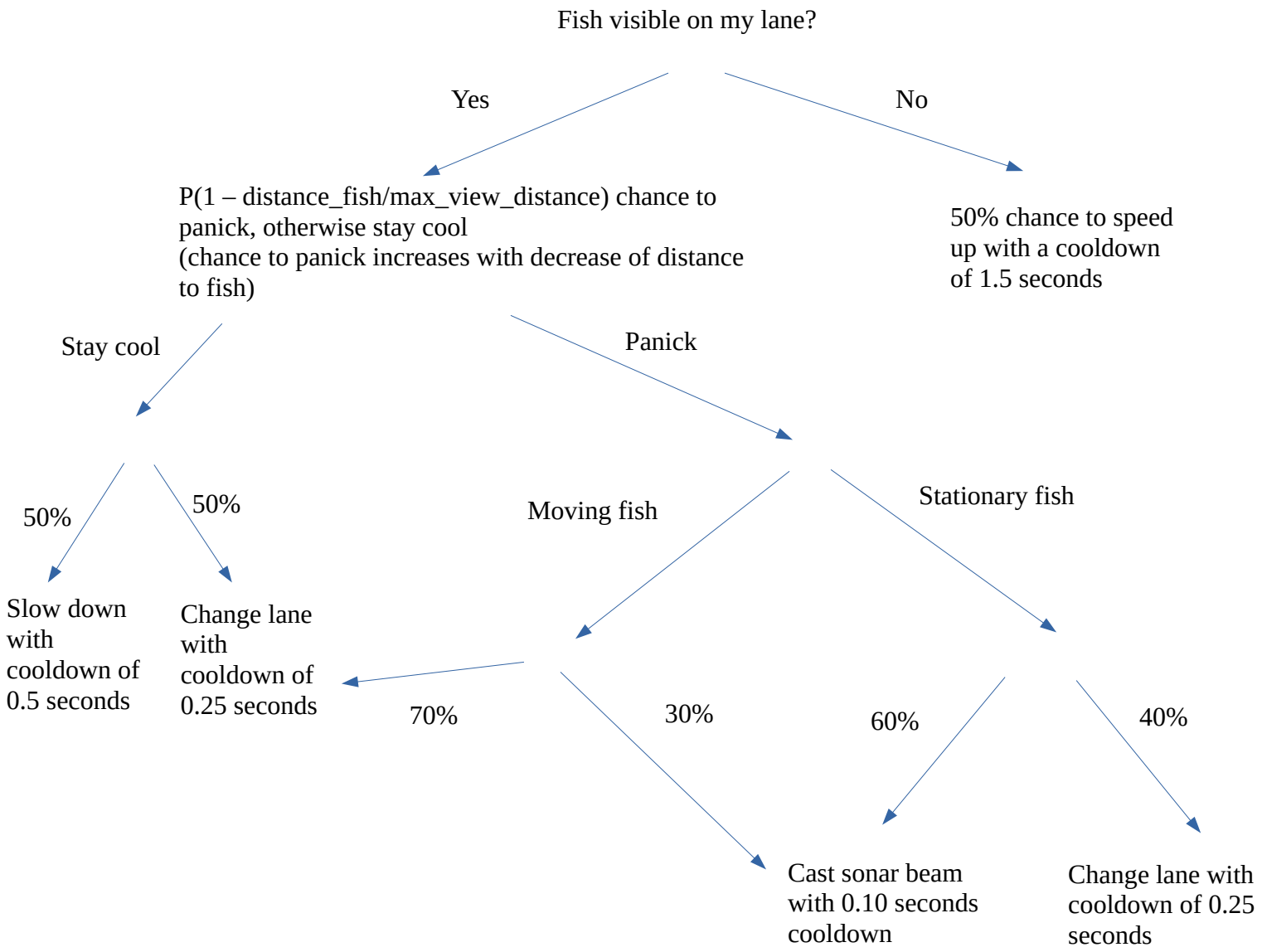
There are also 3 other computer controlled divers who are competing to reach the end. (where an ancient treasure is implied)

These divers have different looks, names and color+pitch of their sonar beams.



But besides the looks and sound, the competing divers have the same behavior. Each one of them has a “cooldown” timer which is invoked after every decision and duration depends on the decision.

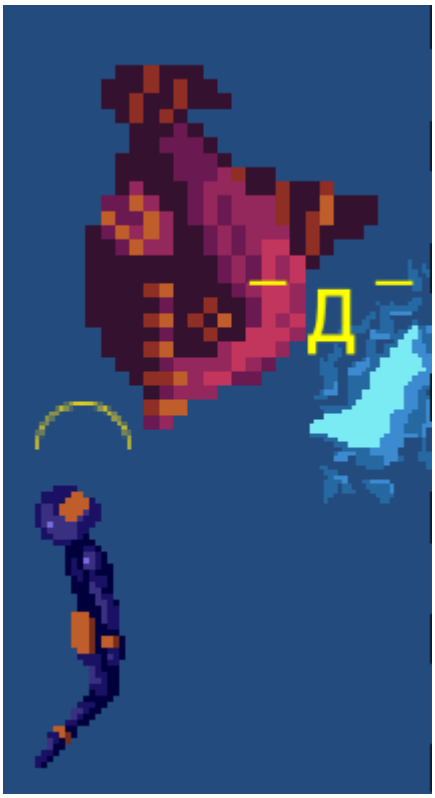
They make decisions as follows:



Additionally if the computer controlled diver is too far behind, they will be placed off screen behind the player and given a temporary speed boost to make sure that the player stays on their toes even if they manage to get far ahead.

But also if the player gets to far behind, the other divers will slowdown until they are visible on screen once more, to give the player a chance to catch up.

There are two types of fish to encounter, each of which will be seen at least 3 times during the play-through.



The red fish will has a bigger mass than the drivers but lower than the blue fish.

It will swim towards you and if a diver casts a sonar beam it has a 50% chance of slowing down to half it's speed and will react with a higher pitched sound and a random frustrated emoticon:

ᵕ̈_ᵕ̈ ¯Д¯ ¬_¬ ¯^¯ ˘_˘ ➘_➘ >_< o~o ಠ_ಠ

The blue fish will not move unless a diver or another fish pushes against it.
If a diver casts a sonar beam, it has 50% chance of going to another lane, reacting with a lower pitched sound and a random startled emoticon:

˚Д˚ 〇o〇 o_o, ⊙_⊙ ノω ヽ °□° ヽ(°〇°)ノ ┌(>_<)┘ (つ>＜)つ

The collision system is such that each diver and each fish can push against each other and the force of said push is determined by the mass of the entity and the velocity of it.

So for example, if the player speed up they will be able to push a blue fish harder which may be necessary if it ignores sonar beams. But considering that the blue fish has the biggest mass, it can push and block many entities, especially when it is changing lanes, so it poses an extra threat and can be used strategically if a diver is lucky with their sonar beams.

Due to a quirk in the collision code, the player may choose to take advantage of an easter-egg. If you push a against a competitor while they are below your waist, you will also push them down, but beware this works also the other way around to push them further in front of you!

Fish are spawned in waves, the game level is split up vertical sections of a fixed size and each time the player enters a new section a certain number of fish will be spawned.

20% chance to spawn just one fish
50% chance to spawn two
20% chance to spawn three
10% chance to spawn four

Each fish is put on a separate lane and randomly vertically spaced from each other (within a certain range).

Once the player has reached the next section, all the fish that have already passed and are not on screen, will be culled.

Each diver starts off with a score of 0 and may get penalized for the following actions:

-100 for colliding with a red fish (per second)
-200 for colliding with a blue fish (per second)
-100 for colliding with another diver (per second, for both divers)
-100 for casting a sonar at red fish
-50 for casting a sonar at blue fish

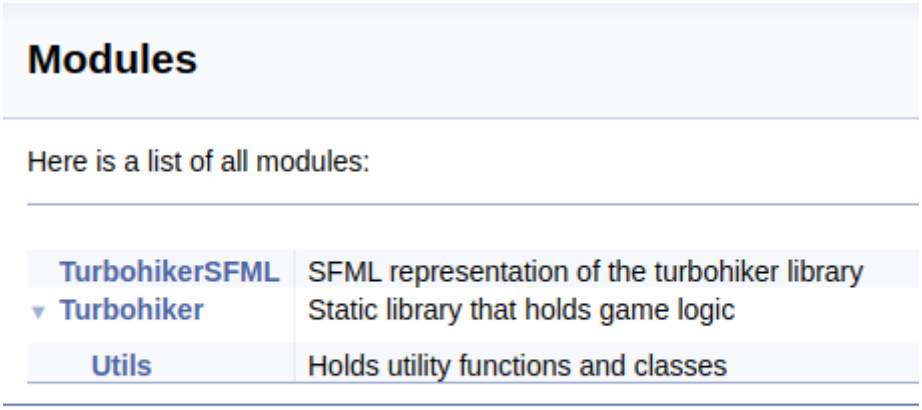
The only way to earn points is by finishing

3000 for first place
2000 for second place
1000 for third place
0 for last place

Design

All game objects and game logic is encapsulated in a static library called “Turbohiker”.

“TurbohikerSFML” provides the visual representation, sounds and captures and passes through user inputs for the turbohiker library.



Modules

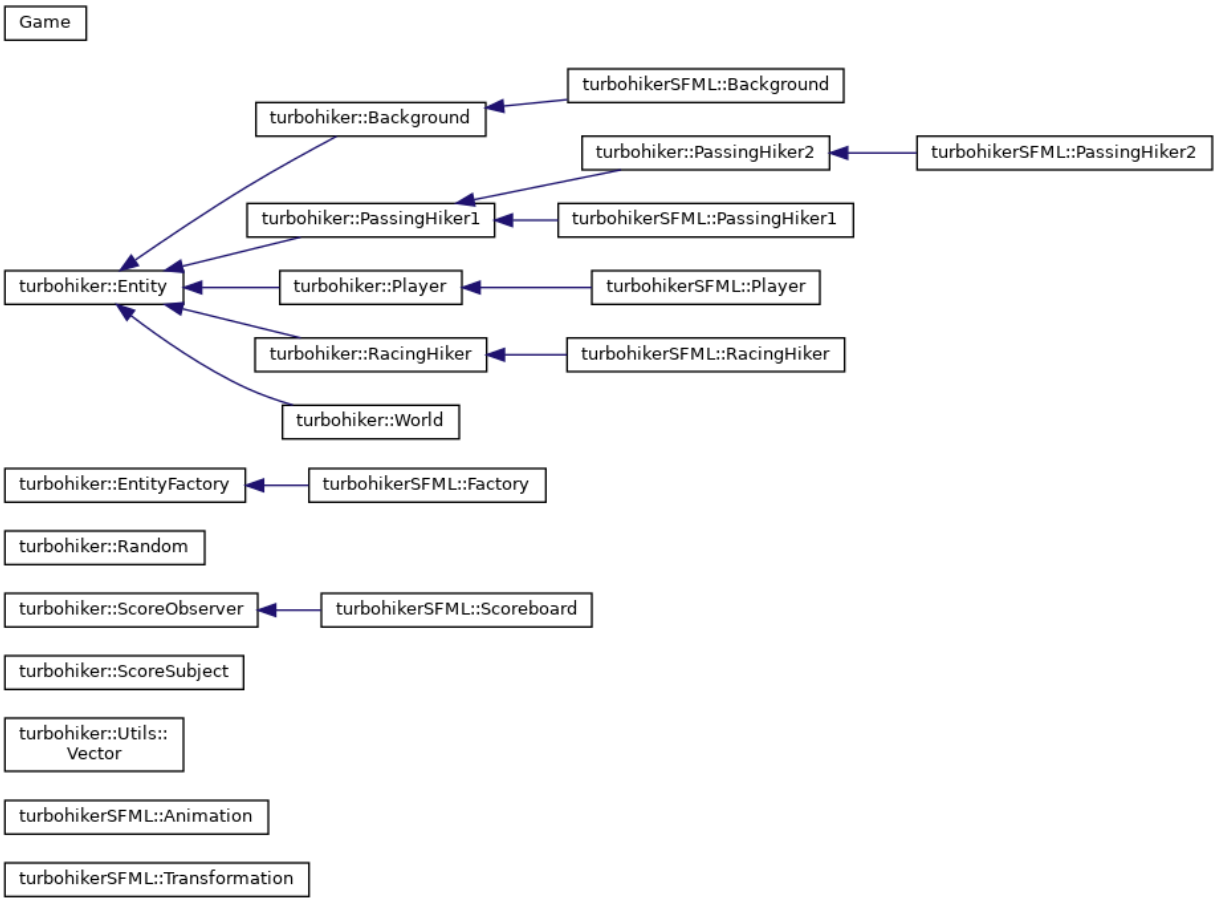
Here is a list of all modules:

TurbohikerSFML	SFML representation of the turbohiker library
▾ Turbohiker	Static library that holds game logic
Utils	Holds utility functions and classes

Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

▾ N **turbohiker**	
▾ N **Utils**	
C **Vector**	2d vector
C **Background**	Abstract class to draw background elements
C **Entity**	Most abstract game object
C **EntityFactory**	Abstract factory for creating entities
C **PassingHiker1**	Abstract class to handle moving hiker logic
C **PassingHiker2**	Abstract class to handle static hiker logic
C **Player**	Abstract class to handle player logic
C **RacingHiker**	Abstract class to handle competing hiker logic
C **Random**	Singleton to generate pseudo random numbers
C **ScoreObserver**	Observes the score object, holds score counts
C **ScoreSubject**	Gathers score information, registers and notifies observers
C **World**	**Game** world, handles physics, hold entities and delegates draw requests
▾ N **turbohikerSFML**	
C **Animation**	Abstracts animations for the SFML sprites
C **Background**	SFML background presentation
C **Factory**	Concrete factory for creating SFML entities
C **PassingHiker1**	SFML passinghiker1 presentation
C **PassingHiker2**	SFML passinghiker2 presentation
C **Player**	SFML player presentation
C **RacingHiker**	SFML racing hiker presentation
C **Scoreboard**	SFML scoreboard presentation
C **Transformation**	Converts pixel values to the visible 2D game world space of [-4,4] x [-3, 3]
C **Game**	Interacts with world by calling the world's methods to enable the game logic holds the gameloop and calculates the delta between frames



The *Game* class holds the game world and the window. It is the only object to always hold a shared pointer to the window as all the other objects that need it only hold a weak pointer to borrow it. It interacts with world by calling the world's methods to enable the game logic and holds the game-loop. In the loop it captures and forwards the user inputs to the world and also calculates the delta time between frames, which is used for game logic and visuals. It is also responsible for resetting the world and the other required objects(such as the transformation singleton), once the game has finished and the user wishes to play again.

Entity represents the most abstract game object. Each entity has a position, a possible next position (for collision detection), a velocity, a rectangular size, a speed and a mass. An entity may also be "frozen" which will teleport it off screen and not interact with any game logic. Each entity also has an update command to update it's logic and a draw command.

Background is an entity that doesn't interact with other entities and is only used to position and update the background visuals.

Passinghikers represent the two types of fish, they hold logic for reacting upon shouts and updating their velocities.

Player class receives and handles user inputs, such as lowering the speed, changing velocity or shouting. Although in the case of shouting, it is purely visual, as the world handles the logic of

```
/**
 * entities
 *  0 : npc0
 *  1 : npc1
 *  2 : npc2
 *  3 : player
 *
 * actions
 *  0 : hit passinghiker 1
 *  1 : hit passinghiker 2
 *  2 : hit another competitor
 *  3 : shouted at passinghiker 1
 *  4 : shouted at passinghiker 2
 *  5 : arrived at finish
 *
```

finding the target and invoking a reaction on said target.

RacingHiker represents the computer controlled hiker, each one holds an integer in [0,2] upon construction to distinguish them. This class hold the logic for the different moves it can do but it doesn't make the decisions for said moves. The world handles the decision making as it has exclusive access to all the other entities within it.

World represents the game world and handles most logic related to it. As mentioned before it holds exclusive pointers to all other game entities, the *EntityFactory* and the *ScoreSubject.* It is responsible to forwarding update and draw commands to those entities(Composition pattern). It will receive inputs from the game object and pass it through to the player or in case of a shout it will find the target of the shout and invoke the correct responses on the sender and receiver of the shout. It will also handle non playable character decision making as stated previously and spawn passing hikers. Additionally after each entity has updated their velocity and speed information, it will calculate their new positions and resolve any possible collisions, of which it will notify all score

observers through the score subject. Finally it will detect if the game has finished and will let the game object know.

ScoreSubject and ScoreObserver follow the observer pattern, the subject will hold and notify all observers of an event. The observer will receive the events, hold scores for each diver and update the scores according to the event. *ScoreBoard* is the derived class of the observer and is responsible for visually displaying the scores. The possible events are:

EntityFactory is an abstract factory, used by the world to create all other entities. The *TurbohikerSFML::Factory* is a concrete factory for the SFML implementation of the game. It will create the SFML derived versions of the game entities and will give each one of them a weak pointer to the window object.

Transformation is a singleton used by the SFML derived classes to translate the logical game coordinates and sizes into pixels. It will also receive position information from the SFML Player derived class to move the “camera” to follow our player and used by the SFML Background derived class to create the parallax effect based on that information.

Random is another singleton, used by the game logic and representation to get pseudo random values.

Vector a help class to represent 2D vectors for use in the game logic.

Animation a help class to represent tile map animations for use in the SFML derived entities.

Additional documentation can be found in the doxygen generated /Docs directory or in the source files themselves.

Assets

Nautilus – background music – CC0 – ***poinl –*** link

Underwater Diving Pack – sprites – CC0 – ***Luis Zuno*** - link