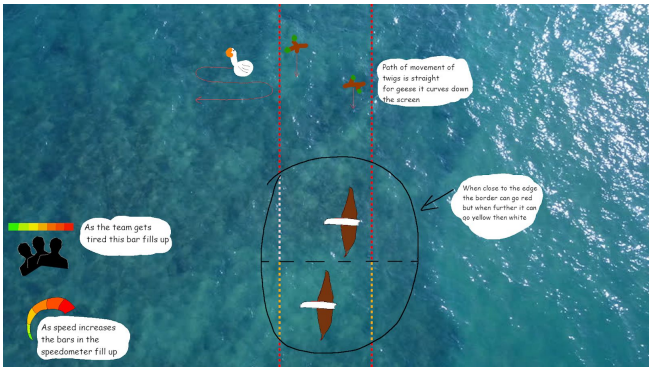
Architecture

TEAM 14 [ENG1]

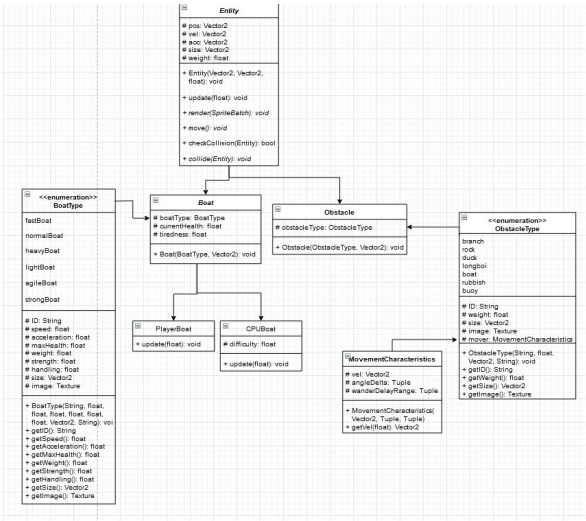
Part a.)

Draft for UI



This was our original plan for the game, we planned on having 7 lanes but only drew in the middle one because it was easier for the draft. We wanted to have a bar to show the speed of the player and a boat to show the players stamina.

The Entity Class and UML



We used uml to create the entity relationships for our java project. We have a basic entity class which the boat and obstacle classes are children of. We used UML because it allowed us to plan out our Java classes in a model that was easy to understand and refer to while developing. This also allowed us to split the work up more easily between us as we knew exactly what code we had to write and there was less need for discussion between us once we had gotten started. A drawback to this was that it took a lot of time to develop but we think that time was well spent.

We have both the boats and obstacles as children classes of Entity, this is because they share many similar properties. Using this we are able to eliminate redundant code.

We gave the entity class a position variable, a velocity and an acceleration variable. These allow us to track where the entity is on screen and where it's heading.

The size variable is the number of pixels wide and tall that the entity will be on screen.

The weight is used to determine how much damage it will do to a boat if they collide. Higher weight = more damage.

The update function determines where the entity is going and the move function changes the position of the entity. This then tells the render function to display the new position on the screen. A SpriteBatch is used to get the image.

The checkCollision function is run on each entity against every other entity to see if they collide and will return a bool, if the bool is true then it will run the collide function which will damage and slow the boat down.

Enumeration for Boats and Obstacles

We used enumeration to create the boat and obstacle classes. This was because it gave us more flexibility when tweaking each boat or obstacle. For example while testing if we decide that we want to add in a new boat, instead of having to create a new class that is a child of boat, we can write one line in the enumeration class. It also means that instead of x number of boat classes we only need 1 class and x number of lines, each defining a boat.

This provides much easier to read code as at a glance you can see the attributes of each boat type.

A disadvantage of this method is that if we want a specific boat to have a power, for example if we want one to shoot cannon balls, it will be harder to program in versus the class per boat option.

Obstacles and MovementCharacteristics()

We have all obstacles contained in one enum. This includes static obstacles that don't move, obstacles that move in one direction, and obstacles that have more complex movement, e.g. moving backwards and forwards on the screen. We have achieved this using a class called MovementCharacteristics(). This class stores the velocity, angle delta, and wander delay range of the obstacle.

For example:

If we want to have an obstacle not move, we can set the velocity to be (0,0). this means it won't move.

If we want to have it go in a straight line we can give it a velocity in any direction and an angle delta of (0,0). This means that whenever the obstacle tries to change direction it will change by 0 degrees, i.e. it will stay in the same direction.

If we want something more complicated we are also able to do that. For example, if we want a duck to move left and right across a lane. we can set the angle delta to be (180, 180), the velocity to be (x,0) and a wander delay range (t,t) where x * t = the number of pixels between the edges of the lane. If we then spawn a duck at the left side of the lane, after t time it will have hit the other side of the lane and will turn 180 degrees to start going the other way.

If we want an obstacle to have more random movement we can give it a large angle delta and a small t. it will then make more frantic movements at different angles.

Using this method we have lots of freedom about the way the obstacles move and it is stored in one class. It takes up very little code as we only require a definition of the movement characteristics and a function to interpret them and decide where to move the obstacle.

A disadvantage of this method is that looking at the code may not easily show how to use the characteristics to create the movement required. To help fix this we will write insightful comments to show future developers who pick up this project how to use it.

More on boats

The boat class is an abstract class, and the parent of both PlayerBoat and CPUBoat. The use of class hierarchy allows us to do all the attribute loading and setting up needed for every boat whilst giving us the freedom to implement differing methods between the two child classes. One of the main benefits of using this class hierarchy is the reduction of repeated code between classes, as well as making it easy to update methods across all the required classes.

Both the CPUBoat and the PlayerBoat class have a move method which decides how the boat will move. For the player this is based on the keyboard inputs, the left arrow key moves the boat left, the forward arrow key accelerates the boat, etc. For the CPUBoat, the move method looks at the current speed of the boat to decide what to do. Both of these methods apply calculations for: slowing down after hitting an obstacle, the use of and regaining of stamina, and locking out inputs if all the boat's health is lost.

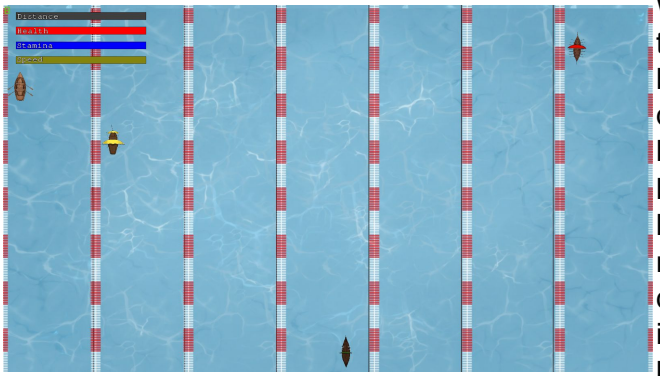
libGDX

We decided to use libGDX for its wide range of included modules which cover all scenarios we could encounter along with the in depth documentation, allowing us to quickly research the optimal methods and classes to use. A great benefit of using LibGDX is for user inputs. By having in-built input listeners, adding controls to the player's boat and navigation to the menus becomes trivial and a great amount of control can be had of what methods are called on each keypress.

LibGDX is used as the backbone of our project. Every attribute we need to know about the user's system is accessed through LibGDX, abstracting the logic away from how the game itself functions.

Part b.)

Some Changes



We moved the statistics for the player into the top left instead of having it in the bottom left. We also added a bar for distance travelled and for the player's health. We decided to standardise the bars as it gave the game a more modern look that we prefered. We also decided not to colour the lane boundaries dependent on how close you are because it made the screen too complicated. As long as the user knows that the lane barriers are bad then they would already avoid them. We also decided to make the background a lighter colour to add more contrast between it and the boats which made them easier to see.

We decided to keep the inheritance structure as it made development much easier. If we wanted to have a change to both players and cpus we could change it in boats, if we had a problem with all entities then we could just change entity. It proved very valuable to have everything organised as a tree with entity at the root. The final concrete architecture is a refined version of the abstract, it is easily expandable and has been built very generally to make the transition and future development for the second team easy. We kept the use of enumeration for the boat types and obstacle types and we fully implemented the abstract UML document. The only significant change was to use libGDX screens which I discuss more below.

Requirements

We originally planned on having the game (i.e. title screen, menus, and the boat race) run in one class , after realising that managing the variables for this would be very difficult and that it would be difficult to expand we split the game into libGDX screens. We have screens for the main menu, the boat selection, the boat race, the transition between races, the boat death and the finale. If we want to add any more it's incredibly easy. When the game is launched the main menu screen is first presented. This fits with User Requirement UR_Main_Menu. It says "Press space to choose a boat" and "Press ESC to exit". If the user presses space then they are taken to the boat selection screen completing UR_Option. We show the user each enumeration of BoatType.java and the different stats that each has. This fits FR_Selection. Because the game is made in Java we can save it as a .jar file and as long as the system we are playing it on has the Java Runtime Environment it can run the game, this means that we fit NFR_Availability.

After choosing the boat the user is taken into the first leg of the race. We have a collision method that is run on every object constantly throughout the game, if an obstacle collides with a users boat then the users boat takes damage. If the collision causes the user to run out of health then their boat is destroyed and the game is over, it will take them to a boat broken screen and tell them to press space to restart. This fits UR_Damage, FR_Boat_Broken, and FR_Damage. To fit FR_Boat_Health we have a bar in our UI that shows the users health, it visibly goes down when they take damage, showing a shadow of what it has decreased from, making it obvious that the user has lost health. We have other

bars in our UI to show the user their stats. It stores these using variables on the Boat.java class, as the CPUBoat and PlayerBoat classes are children of the Boat class they will both have these statistics and all boats will have their stats decline and increase throughout the game while having that tracked. This fits FR_Stats, we track the health, stamina, and speed of every boat. In order to have UR_Penalty we gave Boat.java a variable called laneBoundaries. This would track the furthest left and right that any boat could go, if it goes out of those boundaries it receives a penalty which is added onto their time at the end of the round.

If the user completes all 3 legs and is in the top 3 fastest boats, then they are taken to the final fourth leg. If they complete this in the top 3 then when they are taken to the finale screen it will display them winning either a gold, silver, or bronze medal. If they have not finished in the top 3 then it will still take them to the finale screen but will show that they have lost. This fits UR_Complete, FR_Complete and UR_Medal_Screen.

In between every round and at the end it will show the user their current placing and their placing in the last round, it will also show them their times. This fits UR_Records. It will store this data on each boat and each boat will be passed between screens. The way we calculate the time for every round is by: measuring the system time at the start of the round and saving it as start time, repeatedly checking each boat to see if it has passed the finish line, if any boat passes the finish line we update its attribute final time to be the current system time minus the start time, if the player passes the finish line we finish the leg and estimate the times for the other boats, we do this by performing the function : (player finish time) x [(leg length)/(cpu distance travelled)], as the legs are quite long this gives us a good estimate on how long the cpu would have taken to complete the leg without having the player wait for them to finish. At the end of each leg we compare the users score to that of the CPUs and that gives us the users placing. This fits FR_Compare and FR_Time. The boat time had a few changes from the abstract to the concrete. We added variables to measure laneBounds, finishTime, distanceTravelled and timePenalties. We didn't anticipate needing these but they ended up being crucial for core functionality.

The user's input is checked constantly throughout the game and if they input any of the arrow keys the velocity of the boat will be influenced in that direction. This fits Ur_Race_Track and NFR_Control. The keys to move left and right are the left and right keys only. This fits UR_Boat_Move. Because this is checked so frequently we are able to have quick response times which fits NFR_Response_Time. While these keys are held the stamina of the boat is decreased, this fits Ur_Endurance. If the player is to collide with an obstacle or another boat their velocity is reduced, this fits FR_Boat_Slowed_Down.