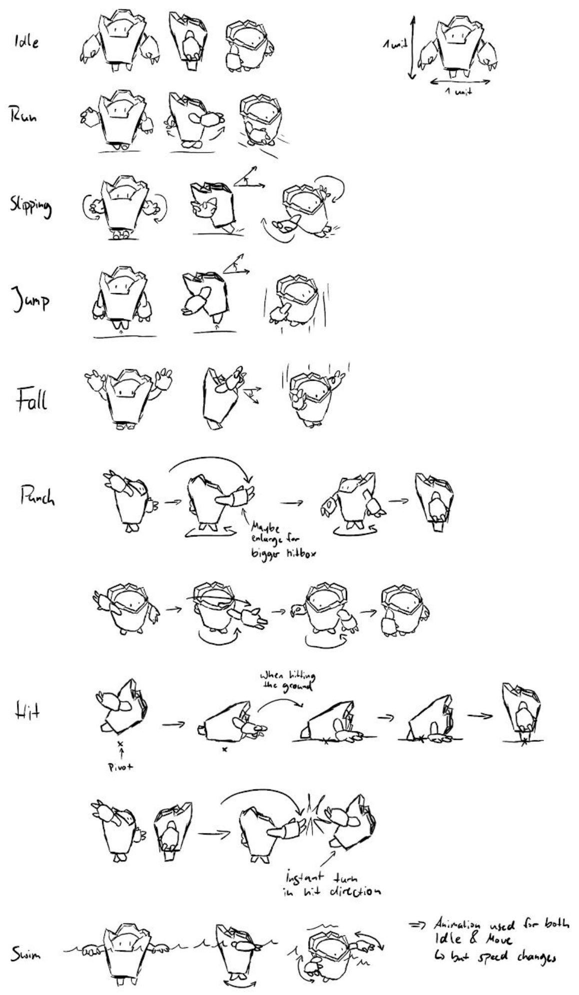
Interim Report

Overall Progress

Overall, we met most of our development goals for this milestone. The key missing factor would be polish. A lot of things are working, and working well. But the core of the game in a cleaner UI and final touches of cleanliness regarding some interactions. Some design choices also have not gotten enough playtesting yet.

Character

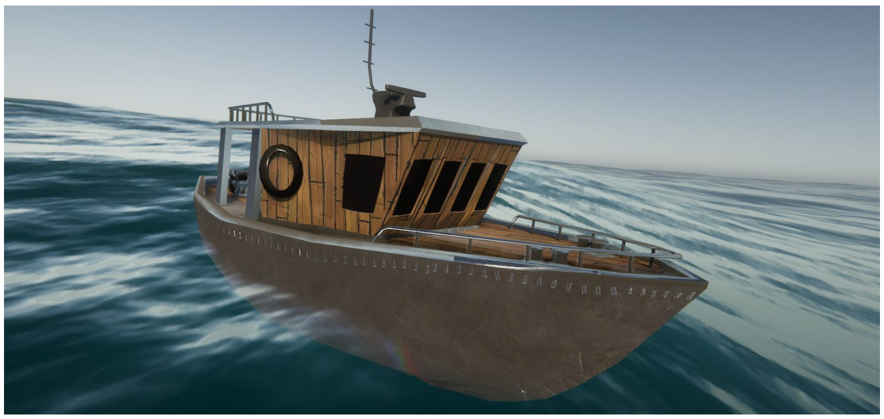
The character model was originally inspired by the Fall Guys characters with some more detail in textures and shape. The characters should look adorable and cute. The golem was modeled, uv unwrapped, textured and animated in blender. For animating the golem we used the rigify tool which is a helpful tool to easy animate humanoid characters. Having the issue that the Golem has no knee and only calfs and no thighs we wight painted the whole leg to the calves of the rig which worked quite nicely. The guidelines for the animations were similar to the modeling guidelines, it should look cute and adorable. We have the animations depicted to the right implemented. The implementation was done using the Unity animator, which is essentially a glorified state machine. The idle/run animation is determined by speed; the swimming is determined by a bool, and the other implemented animations use a trigger (i.e. punch, jump)


Idle
1 unit
1 unit
Run
Slipping
Jump
Fall
Punch
Maybe enlarge for bigger hitbox
Hit
when hitting the ground
Pivot
instant turn in hit direction
Swim
=> Animation used for both Idle & Move
↳ but speed changes

All the animations in action
https://youtu.be/JQ4A8x02aZ0

The model textured in Unity

Art



The art style is somewhere between cartoon style and looking realistic. We want to use the hdrp pipeline in unity to have very nice looking effects such as reflection and post processing features that really enhance the overall look of the game while keep some cartoon style to not have the need of modelling to high detailed models as well as keep some level of cuteness to the characters and world. The boat is a perfect example for this art style because it is very low poly modeled with only a view details but has some more advanced textures on it with cool looking reflections. For the future we plan to have also the tiles for the main island designed with this kind of art style and add more details as well as verticality to the level.

Controls / (Local) Multiplayer

The controls were implemented using Unity's new Input System. This allowed for the development to be much more focused on the integration of the physical controls into the actual movement and handling. This extends to the multiplayer, as the new Input System is capable of seamlessly handling different user inputs and registering them to different playable characters.

Camera

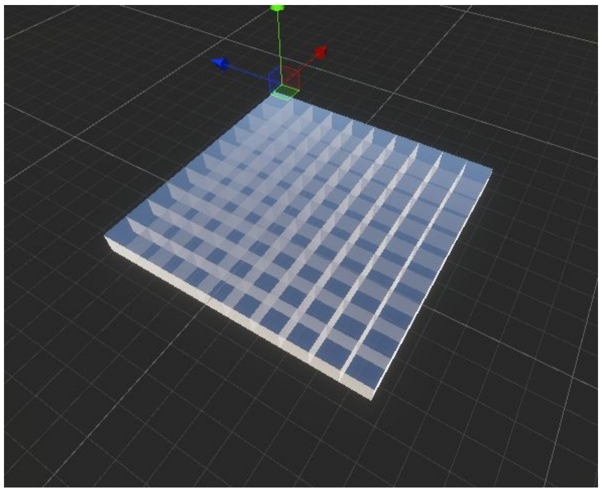
As per our design, the camera does not move, but rather shows. That means that placement was key, but with some minimal testing, it's clear that this would already be much improved if the camera had minor tracking that adjust the size and scale slightly to reframe the scene and keep all characters visible.

UI

The UI was more trouble than anticipated. We are well aware of the troubles that underestimating UI can bring, so it was already allotted its fair share of development time, but still it was quite unyielding, and often resulted in buggy behaviour. One of the biggest culprits was the "Start Game" button, which is technically a UI element, but due to how different it is (players can move around; the button for it is different than the default), we decided to handle it exceptionally to the other inputs, using a "virtual" button to trigger its OnClick method, rather than the UI's built-in submit functionality. Once the very rough bugs were worked out, the UI got to a fairly stable state and definitely should not be an issue now.

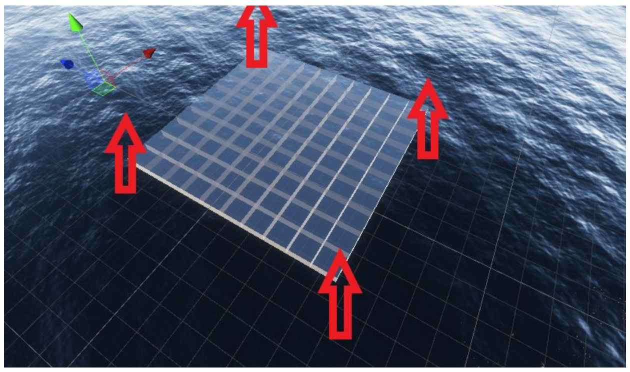
Physics

Platform



The physics system for the game was mainly built on the Unity physics engine, using simple colliders and rigidbodies (on the players) to support collisions, gravity, and physics-based movement. Other parts simulate buoyancy in slightly different ways (see buoyancy section below). In order to test the physics and mechanics of the game, we needed to create a simple platform. A flat cuboid was used to act as a platform. This cuboid is built up of 10 by 10 smaller cuboids (we will refer to them as tiles). So the tiles build up a flat surface that will be used as the main platform of the game where characters will be spawned on and where the brawl will happen. Each tile on its own is breakable and has a certain physical material. This allows us to easily integrate the different ice and snow (physics) materials (having different visuals and friction values) into individual tiles.

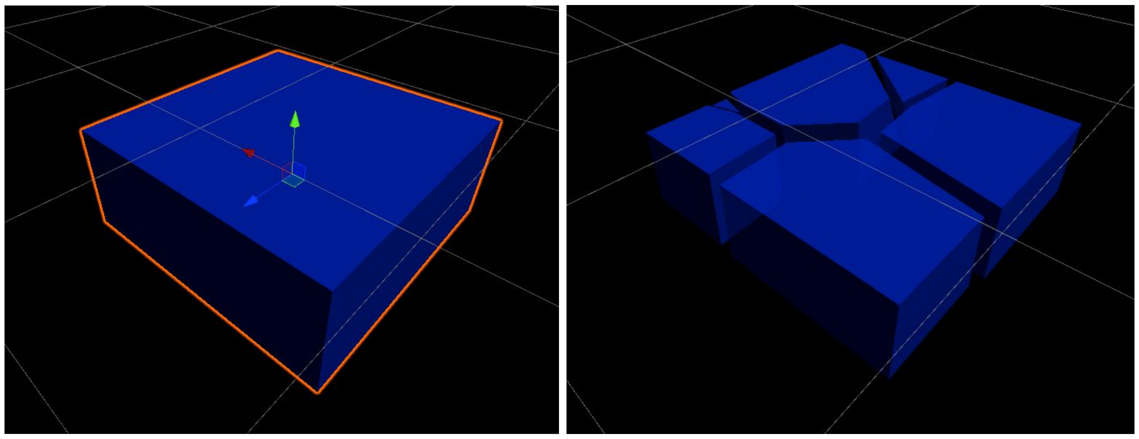
Buoyancy



Since the platform will be placed on water, we needed to simulate buoyancy to it. At first, to achieve this, generic buoyancy scripts were used. But the result of these scripts were not good enough, as they caused the flat platform to be easily flippable and out-of-control. This behaviour, although can be realistic in some way, could be very frustrating for players as they can fall off easily from the platform and lose the game. So another mechanic was used in order to simulate the buoyancy of the platform. This mechanic aims to make the platform not flippable but at the same time would have realistic floating movement that would give the desired difficulty of control. The idea of this mechanic is that instead of applying buoyancy to the platform as a whole with respect to its center of mass, we just add several forces to the corners of the platform to simulate the buoyancy. These forces act as thrusters that keep the platform a float with respect to the ocean level (which is dynamically changing), and these thrusters will only push the platform at their positions when they become below ocean level. The positions and directions of the thrusting forces are represented by the red arrows in the next image.

Destruction

As mentioned in the Platform subsection, the platform is built up of 10 by 10 tiles. These tiles are breakable, and they should be broken by certain events in the game. In order to implement the tile destruction mechanics, first a procedural mesh destroyer script was used. This script would randomly split the tile to a specific number of pieces that will be created as new meshes in the game. This mechanic was dynamic and gave random results which looked acceptable, but it proved to be very performance intensive which could result in frame drops when breaking several tiles at the same time. Instead of doing so, we used predefined broken mesh which was created using a 3D modelling tool.



As seen above, there are two states to the tile, the original one and a broken version which is split to multiple meshes. Upon activating the destruction of the tile, it is replaced with the broken version. After that a rigidbody component is added to each piece and an explosion force is applied on them to make them scatter away. In further iterations, we intend to have more intermediate meshes to indicate the damage level to players. Currently, it internally registers a set number of damage levels (without showing any indications of damage to the player) before suddenly exploding.

Character Physics

The character supports physics through a collider as well as a rigidbody. This allows access to all of the needed physics functionality. For the collider, a simple capsule collider was adjusted to the player's avatar; this is in contrast to a more complex collider as this allows us to save computations where they're unneeded (the physics we should be using resources to compute is actually outside the player model!). The rotations in the X and Z directions were frozen for the rigid body to allow movement through physical forces, without the 'capsule' toppling over.

Punching

An additional object that stores the player's punch location is stored, and, when the player punches, the C# code dynamically checks a radius in front of the 'punch', and applies a scaled impulse along the vector of the punch to the hit player.

Jumping

The jumping implementation is fairly straight forward. The player receives an impulse in their upward vector direction, and can not jump again until a collision is detected. At the moment, the ground does not have a proper tag set (look at known bugs section below), but it's unclear whether it would be better with just the ground refresh the jump, or any collision (needs playtesting maybe)

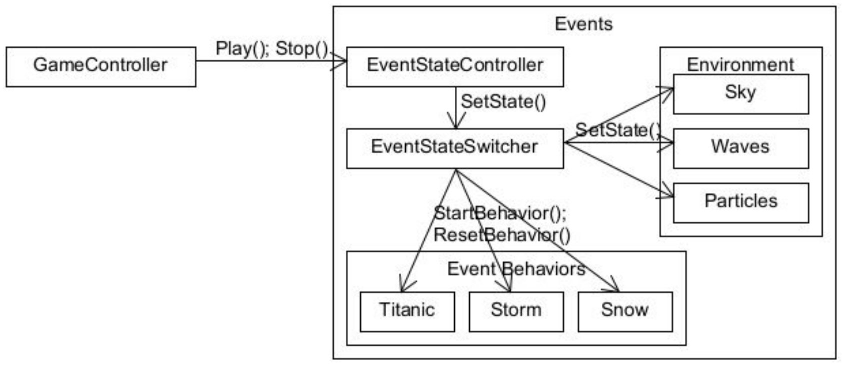
Swimming

Swimming for the player works very similarly to the platform; using the same buoyancy script (with different values), the player is kept afloat, wherein an upward force constantly acts upon the player to keep them at water level. One of the issues we faced was that the water has no real friction so where moving could simply apply a forward force to the player that is stopped by friction with the ground, this was no longer feasible for swimming, so we had to devise a new system wherein a velocity is 'forcibly' maintained as long as the user is 'moving' (through the controller). This could probably be improved.

Events and their Effects

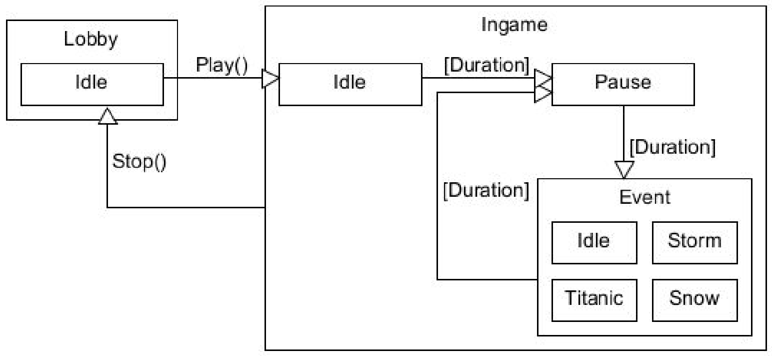
Base structure

The Events system has multiple subsystems and has one interface to the main Event Controller provided to the Game Controller. The interface provides starting and stopping the Events transition logic. The interface has access to the Events Switcher which handles the subsystems Sky, Wave, and Particles. Each manages their own states independently which are set by the Switcher dependent on the current Event. Also, the Switcher enables the current Event Behavior subsystem. Each Behavior subsystem is playable and stoppable by the Switcher.



Logic

The main logic of the Events is as follows:



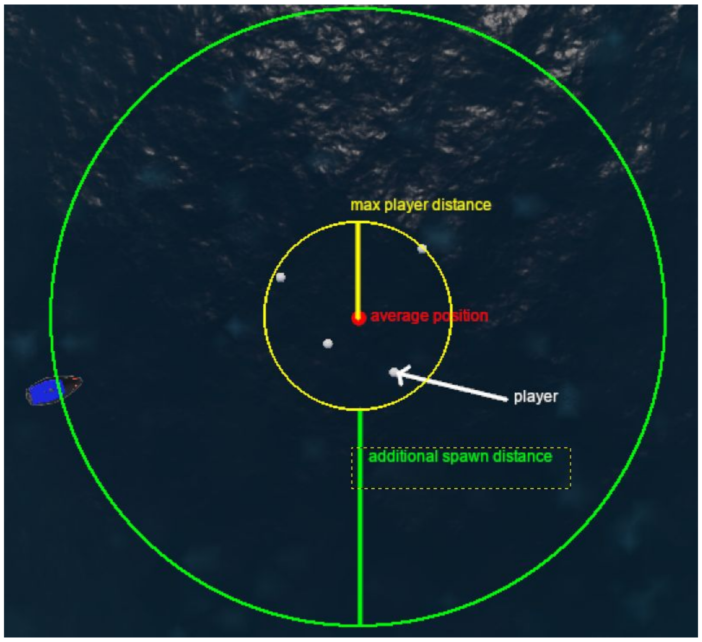
We most probably need to balance the event and transition duration and their probabilities, so they need to be easily adjustable. Also, we plan to change them over time which is why we also need to define the time after which the logic changes ends. For that, we use AnimationCurves as parameters so we can easily adjust the probabilities and durations over time.

Event Behaviors

The Behaviors are enabled by the Switcher if their respective Event is called and the transition to that Event is over. At the end of the current Event, the Switcher also reset the behavior again.

Titanic

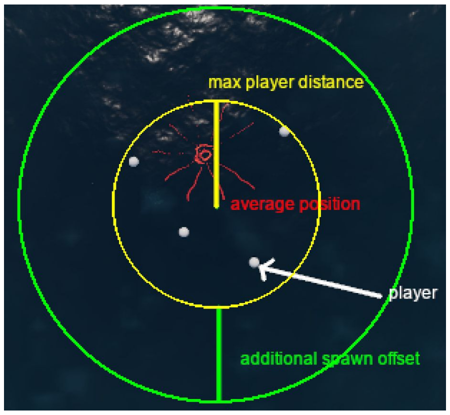
The behavior of the ship depends on the player positions. The spawn point is set on a circle around the center of all players. The ship movement direction is set to the center and moves continuously in that direction. The speed is dependent on the current Event duration given by the Switcher. The destruction behavior is not yet merged with the Tile behavior. For now, the ship sinks when Switcher calls the reset.



Storm

Spawner

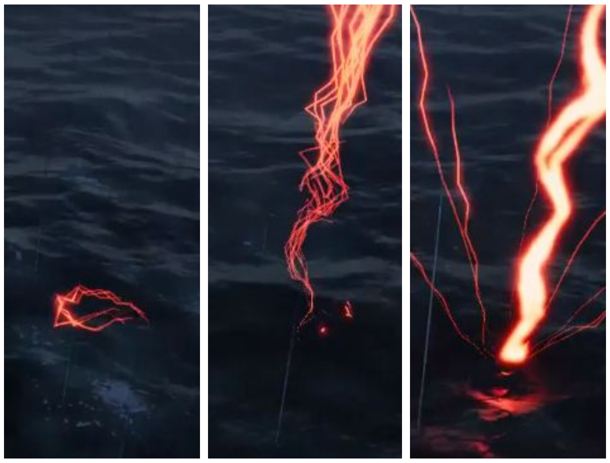
The behavior of the storm also depends on the player positions. It spawns Lightning objects out of the object pool within the circle around the center of the players. The amount of Lightning Strikes over the Event is currently constant, but a dynamic spawn amount will be added to be more flexible.



Lightning

The Lightning has three phases during its lifetime. Their duration can be adjusted:

1. Ribbon: Indicates the Lightning Strike position on the water surface. Is made with the Ribbon feature of the Shuriken particle system.
2. Fall: Using Shuriken, multiple Lightning Trails fall from the sky onto the strike position. A script was needed in order to move all the particles to the point. The way they move until the strike can be adjusted.
3. Strike: A single Lightning Line Renderer pops up and the collider that hits the player is enabled for a short period of time. After that, the Lightning deactivates and is available again in the object pool.

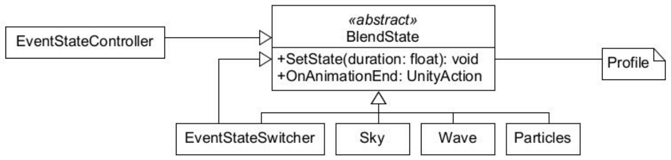


Snow

Currently, only a collider is enabled during the Event. What is to be done, is also fixing the tiles and regenerating.

Blending Subsystems

All the subsystems Switcher, Sky, Wave, and Particles have states that transition smoothly between them. That is why all of their scripts inherit the abstract class BlendState.cs. This class defines the smooth transition between states. The inherited classes only need to define what is to be modified and the State Profiles for each state. The State Profile is a scriptable object that describes the properties, so we can adjust them easily on the asset.



Sky

Intensity of the sun, exposure, and indirect lighting are modified in this subsystem. There is no need of an actual sky box since our view is rotated downwards.

Wave

This subsystem handles the Ocean behavior. We use the asset *Ocean Crest* by *Wave Harmonics* that we are provided by the NGO *Cyan Planet*. This asset creates an ocean surface with height displacement whose waves property are modified by our Wave subsystem on runtime. The asset also provides a sampler to determine the wave height on a specific position, which we have adjusted for our purpose.

Particles

This subsystem changes the emission rate of two particle systems Rain and Snow.

Rain Decals

This feature is actually an Extra feature but has been implemented anyway out of technical curiosity. The challenge here was to spawn decals where a particle hits the ocean with good performance. As a result, we track all particles every [x] frames and check if their position is within a specified range around the current ocean height. If that's the case, a Decal Projector is spawned out of a limited object pool. The decal scales up over lifetime and despawns after that.

Integration

The integration was quite simple. As we proceeded with development, every feature (or set thereof) had its own scene, and for most features, when it was done, it could simply be saved as a prefab. We had a rolling main scene that we could then just load the prefabs into and attach the components as needed. For a lot of the singleton scripts (GameController, LobbyBehaviour, etc..), no references needed to be held as they used static functions that could simply be called as-is without reference.

Future Iterations

Possible Improvements

- Camera could move/scale slightly to make sure all players are kept in-frame as much as possible. Currently players can go
- UI as a whole still needs actual images and text; currently using Unity defaults
    - Lobby UI should be more explicit ("show" an actual lobby)
    - Pause Menu UI should be more explicit
- UI behaviour can still be more stable
- Determine clear jump refresh behaviour
    - Currently refreshes off any collision
- End-game behaviour needs a fair amount of work
    - Consider feedback regarding finality of death
        - Currently players do not can respawn instantly when they die
            - A feature not a bug?
    - No way of restarting game or going back to lobby

Known bugs at the time of the Interim Report

- Jumping does not properly damage ground tiles
    - This is due to the structure of the collider/triggers in the children/parents of the platform which does not correctly recognize the OnCollision Unity functions
- Swimming is fairly buggy
    - Currently no friction to provide real physics-based movement like on the ground
- Players can respawn right away after dying and no state is held to indicate this
    - This is a problem if there are more than 2 players in the game. Then the end-game condition of 0-1 players alive is never reached and the game continues indefinitely
- No damage indication on the floor tiles
    - This can make the game almost unplayable as seemingly perfect tiles will suddenly collapse under the players without any visual queues
- Light effects have some shading and illumination bugs