
# Tub Wars

P3D 3rd Assignment - June 2021

***Daniel Gonçalves [98845]***

***Diogo Silva [87652]***

***Henrique Gaspar [98879]***

# Table of Contents



# 1 The Game

## 1.1 Description & Gameplay

**Tub Wars** is a *Third Person Shooter* in which the player controls a yellow rubber duck, stuck in a depleting bathtub, who must attempt to survive for as long as possible, and take down as many enemy red rubber ducks as it can. The game is complete with music, sound effects, and the entire scene was custom made with textures from **CG0Textures** and models from **Google Poly**. Each match of **Tub Wars** lasts for 5 minutes, or until the player runs out of lives. As time goes on, the **water** on the bathtub **lowers**, with the match reaching its conclusion when the tub empties. On the sides of the tub are markers that help the player identify how much time has passed: if the water is at the **-- 1 --** mark that means that the water is full, if the water is at the **-- ½ --** mark, that means the water is half full, and so on. Scattered around the tub are also **immovable toy submarines** (possess a **box collider** and a **rigidbody** with very high mass and drag) and **movable toy octopuses** (possesses a **mesh collider** and a **rigidbody** with low mass and some drag) which serve as obstacles for either the player or even the enemies. The HUD also contains a **minimap** which helps the player know where enemy ducks are coming from (although their quacking can also help).

It should be noted that, making the water level lower would cause a whole slew of problems due to it being the player's ground, hence collisions might have behaved oddly, and there would've been some noticeable stuttering due to **Unity's Physics engine** adapting to an ever changing floor. As such, rather than lowering the water we instead increase the height of the entire bathroom, hence simulating the effect with none of these problems and without any noticeable difference to the player. We used a **LERP** for this.

During each match the player is expected to get as high a score as possible by moving around with **W A S D** and fighting enemies. For each second alive, the player gets **1 point**, and for each enemy duckling killed the player gets **100 points** so as to incentivize the player to actually fight rather than run away for the match's duration**.** There are a total of **9 enemy ducks** alive at any given point. When one dies, a new one spawns in one of 9 possible spawn locations. This way, the bathtub never gets overrun, and the player never gets too overwhelmed to the point that the game becomes impossible to survive. After the game ends the top **10 highest scores** saved are shown, alongside the score the player managed to achieve in that match. We had initially implemented this feature by creating and storing a custom .txt file, but changed to **Unity's PlayerPrefs** class rapidly, since it can be used to store values that must persist through different play sessions, be them scores or even settings.

**A trailer video can be found at - https://www.youtube.com/watch?v=y7F35J31UTc**

## 1.2 Weapons & Inventory

The player has access to two weapons which shoot forward out of the duck's mouth - a **Bubble Beam** which the player activates pressing and holding the **Left Mouse Button**. As the beam is used the player's **Water Tank** gets depleted and if it fully empties, the player must wait until it's at 25% full to be able to shoot again. Anytime the player stops shooting this tank starts filling up gradually. It takes a few seconds of being hit by this weapon for an enemy to be killed. This weapon was implemented by shooting a **beam of particles with collisions enabled**. Each particle deals **damage** and **slows down** the **enemy**. The second weapon is a **Water Balloon**. The player has a limited number of balloons in their inventory which get consumed when thrown. By pressing and holding the **Right Mouse Button** the balloon's trajectory is shown to help the player aim. Releasing this button will then throw a balloon. These balloons kill enemies in **one shot**, or are destroyed when they hit the water, exploding into a **splash of particles**. To get more balloons the player must ram **wooden toy chests** scattered around the map. These reward the player with 3 water balloons, as they're **destroyed** and respawn after sometime of being destroyed. All of these collisions (both against treasure chests, alongside with enemies bumping the player, or the player's weapons hitting are verified through **OnCollisionEnters**, checked by either the player, enemy or water balloon scripts).

## 1.3 Enemies

There are two different enemies in **Tub Wars**:

- **Enemy Ducklings**
  - These enemies spawn in random spots and wander around the bathtub until the player comes too close, after which they will begin to chase. They were implemented as **NavMesh Agents**. A navigation mesh is baked on top of the "playing field" which takes into account the positions of obstacles, hence informing these agents where they can and can't move. When the player comes close the agent's target is set to the player and they automatically compute the shortest path, taking into account obstacles. When wandering around, they simply pick a random target within a radius, travel to it, and then swap out, so as to simulate this behavior.
  - These enemies *Quack* at random intervals (producing both sound and a <QUACK> onomatopoeia), possess a health bar, displayed on top of them, and, when they touch the player, they **explode into particles** dealing a full heart of damage to the player. If killed they award the player with 100 points.

- **Enemy Toy Boats**
  - Scattered around the map are toy boats which have a cannon on their broadside. Should the player come into their cone of vision, the cannon will shoot out a water balloon aimed at the player (offset by taking into account the player's moving speed). These projectiles shoot in a **parabolic trajectory always at 45º degrees** (picked both due to simplicity and because an initial angle of 45º allows us to shoot the balloon further than if we were to use any other angle). If the balloon hits the player it will deal half a heart of damage. These enemies cannot be killed or moved and are meant to be maneuvered around.
  - As an indicator to the player of where the balloon will roughly land, as a balloon is shot, a **realtime red spotlight** is instantiated (and then destroyed when the balloon is destroyed) on top of the boat's target's position. This also adds some depth and anxiety to the player, alongside allowing us to showcase Unity's RealTime Lighting (and above all, it just makes for a really cool effect).

## 1.4 Freecam Mode

Since it's hard to notice all the small details and graphical fidelity of the game during **Tub Wars**' hectic gameplay, a freecam mode was created. This can be accessed via the main menu and basically unlocks the home camera, allowing it to be moved freely with **W A S D** and aimed with the mouse (W always moves the camera towards where its pointing) and swaps out the epic music for a smooth jazz relaxing the player and allowing them to take in the scenery. Whilst in this mode the player can **take pictures** by pressing **C** - these are then stored as .png in the ***TubWars_Data/pictures*** folder. This feature was implemented using **Unity's ScreenCapture.CaptureScreenshot** class. To leave this mode, press **ESC**.

# 2 Graphics & Effects

## 2.1 Universal Rendering Pipeline

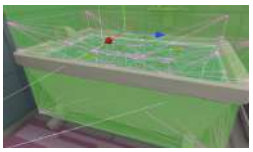
Between **HDRP** and **URP**, the team decided to go with **URP**. We did it, both because all members were more accustomed to it, but also because it allows us to easily and quickly create optimized graphics, with a plethora of supported features allowing us to touch on all required aspects of the project (from shader graphs to global illumination), and some more. There were, however, some limitations, notably, we initially thought that **URP** supported **Lens Flares** natively, but found out this was not the case. Even still the team managed to achieve this effect in other ways.

## 2.2 Global Illumination & Ambient Occlusion

Firstly, for global illumination, we utilized **Unity's LightMaps** due to their performance and good results. This was a bit tricky, since our scene, although technically static relative to itself, does move up as the game goes on. We basically had to mark all our bathroom objects as **static**, **generate the lightmap**, and then attach them to a non-static parent and uncheck them as static so that they could be moved via script. We generated a **lightmap** with **64 direct samples, 512 indirect samples**, **64 environment samples**, **2 max bounces**, a resolution of **32 texels per unit** and overall max size of **512**. These settings were chosen after a lot of trail and error and they seemed to be the perfect balance of performance (in terms of how storage-heavy the final generated lightmap was) and appearance. We also toggled on **Ambient Occlusion** before generating and increased a bit the **Indirect Intensity** in order to make *color bleeding* a bit more noticeable. Since our scene takes place indoors, we also made it so the skybox was fully black, rather than Unity's default dark blue.

Before baking we added **7 light sources** - 5 lamps on the ceiling and 2 windows. We basically created two different materials (one for each type of light source) and enabled them to be **Emissive**, with the lights emitting a whitish light and the windows a more orange light with less intensity.

Since the objects that appear during gameplay are definitely not static (i.e the player, enemy ducks, and so on), before baking we also added **Light Probes** that encased the bathtub and enveloped all of these objects. This way, our moving objects would also be influenced by the lightmap (the way it works is the light probes store the light information and then during runtime moving objects get this information from the nearest probes).



Img 2.2.1 - Our LightProbe group responsible for lighting our moving objects during runtime

Since we also wanted to play around with **Realtime Lighting**, and since it fit our gameplay nicely, we made it so when an enemy ship shoots, a **red spotlight** is spawned indicating where the shot will land. This gives the player some visual indication so they can avoid the shot, and blends quite well with the water. This spotlight is promptly destroyed when the balloon pops.

Img 2.2.2 - Realtime lighting (left), Ambient Occlusion and slight color bleeding (middle), overall scene with lightmap applied (right)

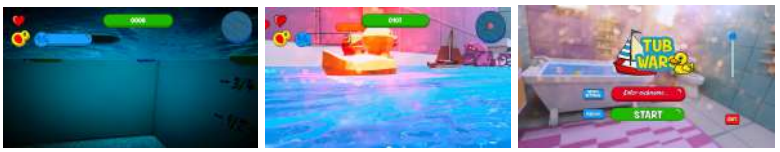
## 2.3 Post Processing & Lens Flares

There are 4 **Post Processing Volumes** spread throughout the bathroom which get enabled or disabled as needed. The way these works is through camera collision. When the camera is detected as being within the volume, the propper effects come into play. We have an **UnderWater** and **AboveWater** volumes used during gameplay, a **HomeEffects** volume for when the player is on the main menu and a

**FreeCamEffects**volume for the free cam mode, which is basically a toned down, less bloomy version of the HomeEffects profile.

- **HomeEffects**
  - Has **ToneMapping** (ACES)**, White Balance, Color Adjustments** to increase saturation, **Depth of Field** (using Bokeh) with a fixed focal length of 51 and an aperture of 6, **Lens Distortion** to add a bit of "fish eye effect" and very pronounced **Bloom** with **Lens Dirt**. All these effects would be unusable during gameplay since due to how much we cranked them, they would be too distracting to play with.
- **FreeCamEffects**
  - Has an identic profile to **HomeEffects** but with less bloom, less lens distortion and the **Depth of Field**'s focus distance is adjusted in runtime (Dynamic Depth of Field), with a fixed aperture of 3.4.
- **UnderWater**
  - Has **Vignette** (which increases as the camera goes deeper), **Chromatic Aberration**, **Channel Mixer** which influences the impact of each color in the mix, **Film Grain**, **Lift Gamma Gain** and **Split Toning** (which influences the colors of shadows and highlights)
- **AboveWater**
  - Employs **Tonemapping (***ACES*), **White Balance**, **Color Adjustments** to add some *saturation and post exposure*, **Bloom** with Lens Dirt and slight **Shadows Midtones Highlight** to add some more depth

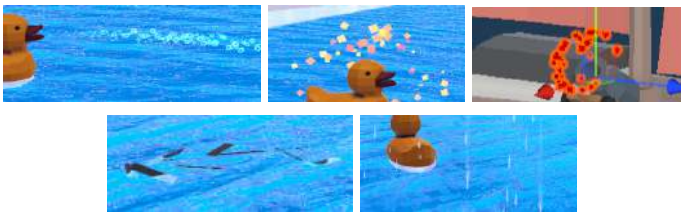
Unfortunately, we discovered that **URP** does not support **Lens Flares** (https://docs.unity3d.com/Packages/com.unity.render-pipelines.universal@11.0/manual/universalrp-builtin-feature-comparison.html). A similar effect, however, can be simulated with **Bloom Post Processing**, by adding a **Lens Dirt** texture and giving it some intensity. The result is nigh indistinguishable with the right texture and very performance friendly (since all it's doing is overlaying a PNG image on top of the bloom and making it glow).




Img 2.3.1 - The Underwater, AboveWater and HomeEffects volumes.

## 2.4 Particle Systems

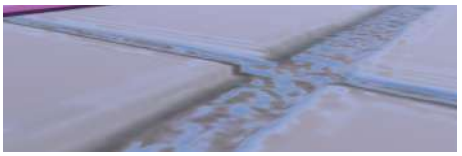
**Tub Wars** is filled with particle effects. Namely, there are particles for when **the ducks move**, a **duck dies** (which uses, as an emission shape, the duck's model itself), a **cannon fires**, a **water balloon explodes** (whether it be on the water or on a duck). These were all created using **Unity's Particle System** and made to invoke a "low poly" look, hence each particle's render is a simple unlit square. The particle system for when **a chest is destroyed** uses, instead, an actual wooden board mesh as a particle. The duck's **bubble beam** is also made up of particles which actually have **collision**, allowing us to detect when they hit an enemy duck and deal the appropriate damage (alongside their render being an image of a bubble which worked quite well for the effect we were going for). Finally, if the player dunks the camera in the water and then brings it back out, they'll see **water droplets and trails** going across their screen for immersion. This effect was also done using a Particle System. We simply check if the camera's position is underwater (which works since the water isn't actually draining, rather it's the bathroom thats increasing in height overtime) and raise a flag so that the next time the camera rises, the particle system, which is placed in front of the camera at all times, is played.



Img 2.4.1 - Some of the particle effects. In order from left to right, top to bottom - bubble beam, duck death, cannon fire, chest destruction and water droplets

## 2.5 Bump Mapping

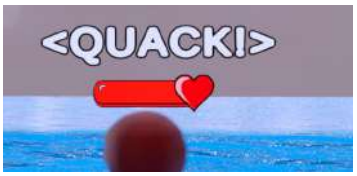
Both the walls and the floor's textures were taken from **CG0Textures**. These came with the texture itself, but also a **normal** and a **heightmap**, which was then used to create the bump mapping on their respective materials.



Img 2.5.1 - BumpMapping as seen on the floor

## 2.6 Billboarding

A billboarding effect can be seen in the enemy ducks. At random points in time an onomatopoeia "<QUACK>" will show up on their heads which will have this effect. Alongside this, their health bar (which is always displayed) also showcases this. The way this effect was implemented was via a script - *Billboard* - which takes swaps the object it's attached to's rotation to be the same as the camera.




Img 2.6.1 - The billboard effect making it so the health bar and the <QUACK!> always face the camera

## 2.7 Reflections

Reflections were done in two manners. For the bathroom mirrors, which the player can only really observe during free cam mode, we used a **Baked Reflection Probe**. We did this because the bathroom scene is entirely static, seldom for what's within the tub, so we can just bake the reflection map offline and not have the performance hit of having to compute reflections in real time for something that the player won't actually see during gameplay (which is when non-static objects actually appear in the bathroom).

Alongside this environment mapping, however, we also have **Realtime Reflection Probes** which were placed within the submarines, giving its metallic-material lain windows reflections which get computed whilst the player is playing. We quickly found out this had a heavy impact on the performance, and as such, added the option within the main menu for the player to swap the quality of these reflections. On **low** reflections are only 256x256 textures and these reflection textures are updated over fourteen frames (with **Time Slicing** setting on **Individual Faces**). This makes it so the reflections are pretty low res and "stuttery", but it does improve performance. Whilst on **medium** reflections are 512x512 textures which update over 9 frames (**All Faces at Once)**. And finally on **high** reflections are 1024x1024 textures which update every single frame making them smooth and very high rez but very much performance unfriendly.

Img 2.7.1 - Baked Reflections (left) and Realtime Reflections (right)

## 2.8 Water Shader

For our **water shader** we started by following a cartoony water tutorial by *Brackeys* (https://www.youtube.com/watch?v=Vg0L9aCRWPE) , but then swapped over to ***PolyToots Water shader tutorial*** (https://www.youtube.com/watch?v=zD6GV6bZenM). This was done due to the sheer amount of features this tutorial taught, alongside its introduction to the **Amplify Shader Editor** tool - a Unity addon which provides a more indepth and feature rich node-based shader editor with support for

“functions” (akin to Unity’s sub-graph shaders) which help organize our graph. Our water shader is therefore an “Amplify Surface Shader Graph” with custom functions. It features **Caustics** through *Voronoi Noise* and taking into account the depth, **Displacement** to simulate waves using a custom *displace texture*, customizable water **color** with the addition of LERPing **highlights** meaning the water’s color transitions slightly depending on its height, **Refractions** that take into account the maximum depth (a value that sets how “translucid” the water is, created using a *depth mask*) and **Foam Edges** around objects on the water’s surface. There’s a plethora of values to customize allowing us to get our water to look exactly how we want it to. Since the scene takes place in a bathtub we decreased the overall wave height and speed (to simulate more stagnant water), increased the caustics size, made the refractions really noticeable and gave our water a “cartoony” very light blue color.

Img 2.8.1 - Our shader graph

One thing our water does not have, however, is physics, in the sense that waves don’t actually make objects on the water’s surface go up and down. To simulate this effect we created a custom **WaterBobbing** script and attached it to all our objects on the water’s surface. This script utilizes a sinusoidal function to alter these object’s *y* positions so that they appear to be bobbing slightly with the waves, hence adding more depth to our scene.

Img 2.8.2 - Showcase of some of the water’s features - refraction, waves, caustics, slight edge foam and watercolor lerping

## 2.9 Dynamic Depth Of Field

To add some more dynamism to our **freecam** mode we implemented a **Dynamic Depth Of Field**. By default, during this mode, a PostProcessing volume is activated which spreads throughout the entire bathroom. This volume has effects like color correction, bloom and **Depth of Field**. The way we make it dynamic was by attaching, tou the moving camera script, a method which shoots out a raycast from the camera towards it’s forward direction. This ray has a limited range. If this raycast hits an object (the entire bathroom scene was outfitted with colliders for this), then the depth of field’s **focal distance** is swapped out to the distance between the camera and the intersected object. This way the camera will focus on the object it is looking at, and blur all rest. Furthermore, the focal distance transition is done using a **LERP**, making it so focusing appears smooth. If no object is intersected, then the focal distance is set to the maximum (meaning unless things are really far away, they won’t be blurred out).

Img 2.9.1 - Showcase of the Dynamic Depth of Field focusing close objects (left and middle) and faraway objects (right)