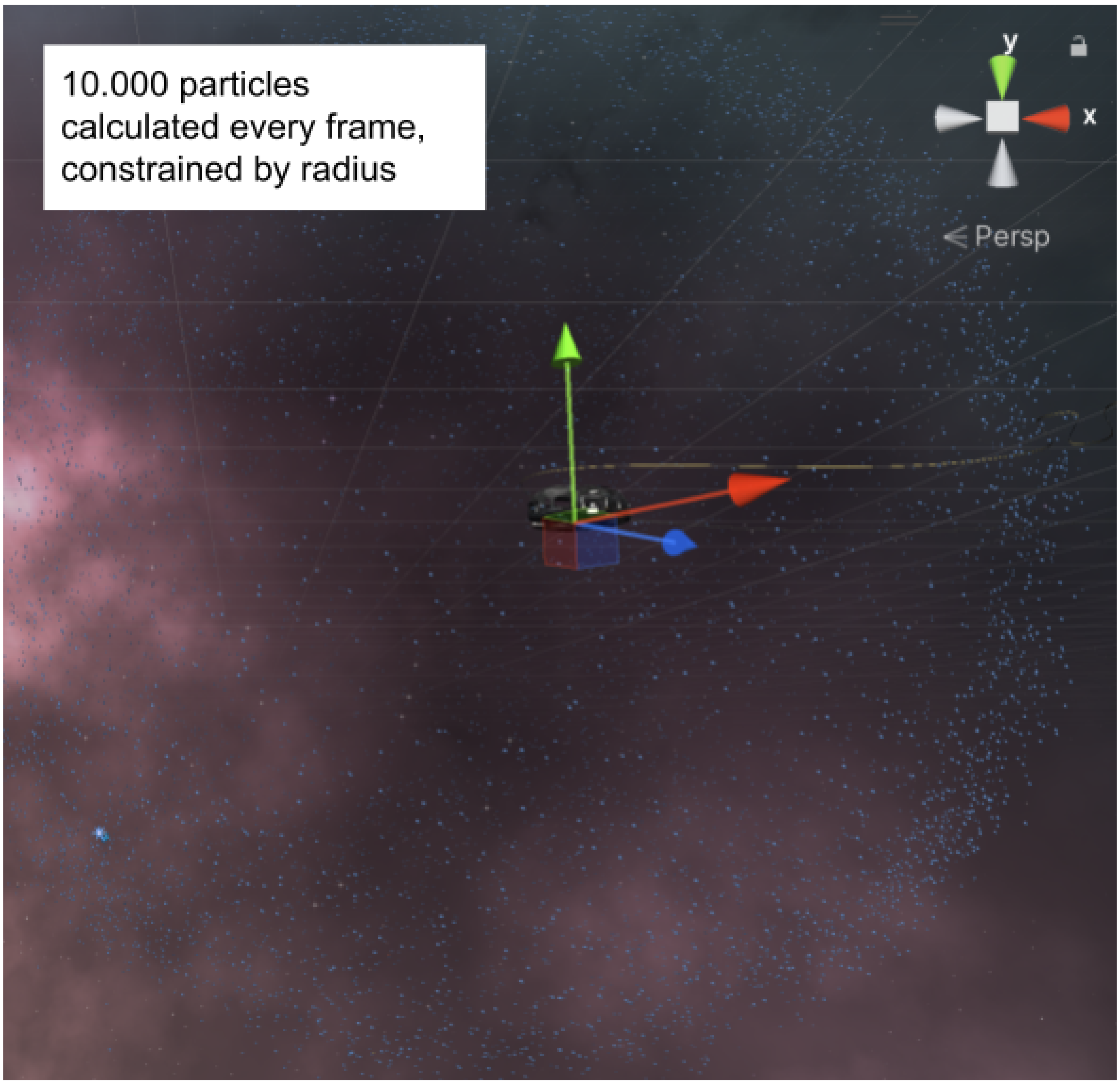
Alpha Release Gravity Gladiators

Graphical Effects

Gravitation Simulation for Stars

To create a more pleasant environment that fits the space theme we are aiming for, we add a Gravity simulation that is run with a compute shader and rendered with a geometry billboard shader. The calculations are simple: Newton's law of universal gravitation combined with Eulerian integration. The naive implementation runs with O(n^2). Optimizations were not necessary due to the small number of particles (1000 is enough to create a nice moving/living background, and does not affect performance significantly).
For added effect, the intensity of the light is controlled by a sine wave function.
Further optimizations could be to calculate only half of the particle interactions and apply the (mirrored) summed forces (n^2/2 + 2n) in a next pass, or use an approximation with, for example, the Barnes-Hut algorithm (O(n*logn)).


10.000 particles calculated every frame, constrained by radius
y
x
Persp

Many problems occurred that are not easily debuggable in a shader: Bill boards not showing due to wrong triangle orientation, NaN values due to division with zero, too large forces xx

$$F_{\mathrm{G}} = G \cdot \frac{m_1 \cdot m_2}{r^2}$$

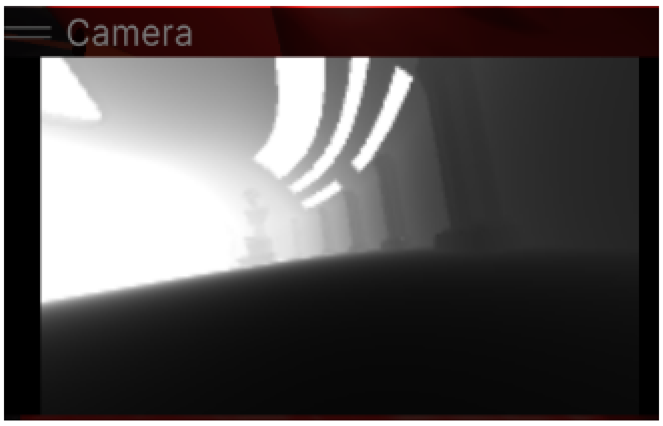
Newton Gravitation, $G = 6{,}673 \cdot 10^{-11} \frac{m^3}{kg\,s^2}$

blowing up the simulation, etc....

Experimenting with navigation

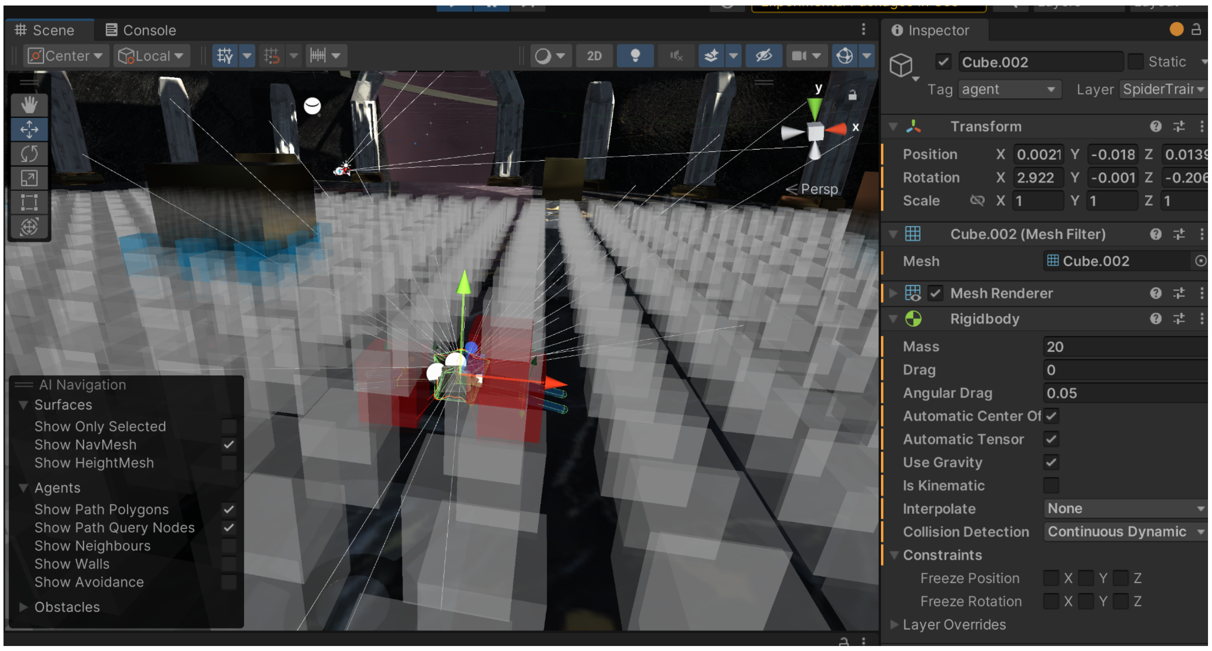
The current way of navigating in Unity is to use NavMesh, a package that uses the assigned mesh to calculate pathfinding for 3D objects (it takes size into account).
It's performant and easy to set up; a major limitation is that it doesn't handle unconnected areas well (as with flying agents) and only computes the shortest path, but not the fastest, resulting in very predictable movements.
A big problem for us is that with the changing rotation of the level, the precomputed navigation points are useless: The fastest way to get from A to B, for example, is then to jump off and let the slope and gravity accelerate you, instead of trying to follow a linear path.
To achieve this, a lot of time was spent experimenting with different sensor types and reward functions.
The process was cumbersome due to long training times (5 hours min to see first results, 20 hours for whole training). Simple bugs were again a major issue and caused many issues.
Only a more involved debugging system helped (printing out every information).

Depth Ray Sensor



The agent needs to know the shape of its environment. Our first implementation using Unities Raycast was way too slow as the rays were computed on the main thread and slowed down the training significantly.
The second attempt was to use a custom sensor type that uses CNNs to compress the rays; a script shoots a ray through each vertex of a UV sphere and maps the hit distance back into the texture map by the corresponding UV coordinates. This custom sensor produced a usable map, but the performance problems persist..
The third iteration/idea was to use a depth pixel shader. The first attempt was computed in the vertex shader, which produced strange artifacts when the vertices were clipped.
The pixel shader worked as intended.

Voxel Sensor



```
AI Navigation
▼ Surfaces
    Show Only Selected
    Show NavMesh             ✓
    Show HeightMesh
▼ Agents
    Show Path Polygons       ✓
    Show Path Query Nodes    ✓
    Show Neighbours
    Show Walls
    Show Avoidance
▶ Obstacles
```

```
✓ Cube.002                         Static
  Tag agent              Layer SpiderTrair
▼ Transform
  Position      X 0.0021  Y -0.018  Z 0.0139
  Rotation      X 2.922   Y -0.001  Z -0.206
  Scale         X 1       Y 1       Z 1
▼ Cube.002 (Mesh Filter)
  Mesh                  Cube.002
▶ ✓ Mesh Renderer
▼ Rigidbody
  Mass                  20
  Drag                  0
  Angular Drag          0.05
  Automatic Center Of   ✓
  Automatic Tensor      ✓
  Use Gravity           ✓
  Is Kinematic
  Interpolate           None
  Collision Detection   Continuous Dynamic
▼ Constraints
    Freeze Position     X  Y  Z
    Freeze Rotation     X  Y  Z
▶ Layer Overrides
```

However, this was not enough to get the agent to navigate. It was clear that it needed more local or global information to have any chance of computing a path.
For this we used the Unites Grid Sensor, which detects tags on a layer and stores them in a png file for use in a CNN.

Impulse Sensor

Another idea was to further optimize the collision detection of the legs with the ground. Instead of a binary TouchGround, we implemented a "TouchSensor" that maps all impulses from physics collisions from the surface into a cubic map, which it then feeds into a CNN.

The mapping was quite complex, since normal colliders do not store information about the UV coordinates at impact. My solution to this problem was the following:

- Fitting an AABB to the shape of the collider using the Unity BoundingBox function.
- Determine the position of the pulse
- map "ClosestPoint" to the surface with Unities and then assign the point to the corresponding surface with a simple Cube Mapping switch
- save the unfolded cube in a texture

The mapping works, but whether it makes running smoother/more optimized could not be tested yet.

Results and Outlook

Other ideas include storing the normal of the surface hit in an image (polar coordinates or storing the distance in the normal vector as a length) and using 3D CNN (which are not available in United MLagents, but could be added by very roundabout changes under the hood).

With these three sensor types, the agent actually trains itself a useful level of collision avoidance within the environment.

Additional Map and Enemies Changes

Missiles

The spider enemy still moves at a somewhat lower speed, so we've introduced a long-range attack for it: He fires small missiles in bursts that follow the target with a Quaternion PID controller (available in the Asset Store). Euler PID controllers are not suitable for this task due to their obvious singularities, while quaternion PIDs are too complex to implement for this project.

It creates an entertaining experience where the player tries to find cover to avoid being hit by the barrage of flying arrows. The missiles are 100% physics controlled and therefore compatible with all other features of the game.

Cover

To allow the player to further interact with the map, we added cover (large heavy cubes). These are the main problems for agents when traversing the map.

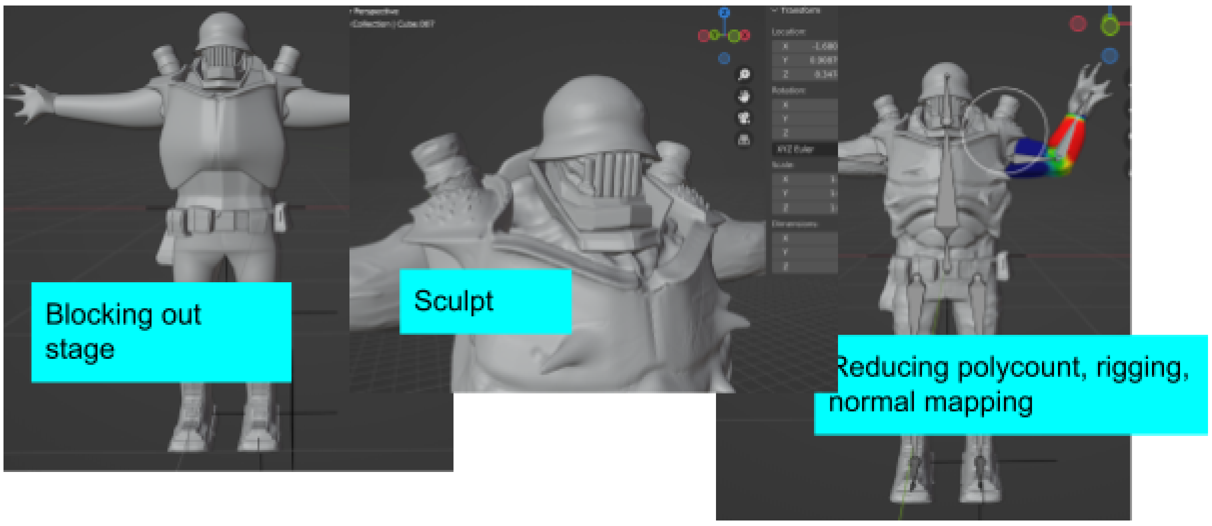
However, our approach has solved this problem. The spiders visibly make an effort to avoid the blocks, even if they are not always successful. This fits with the need to reach the player at a reasonable time.

Humanoid characters and other enemy types

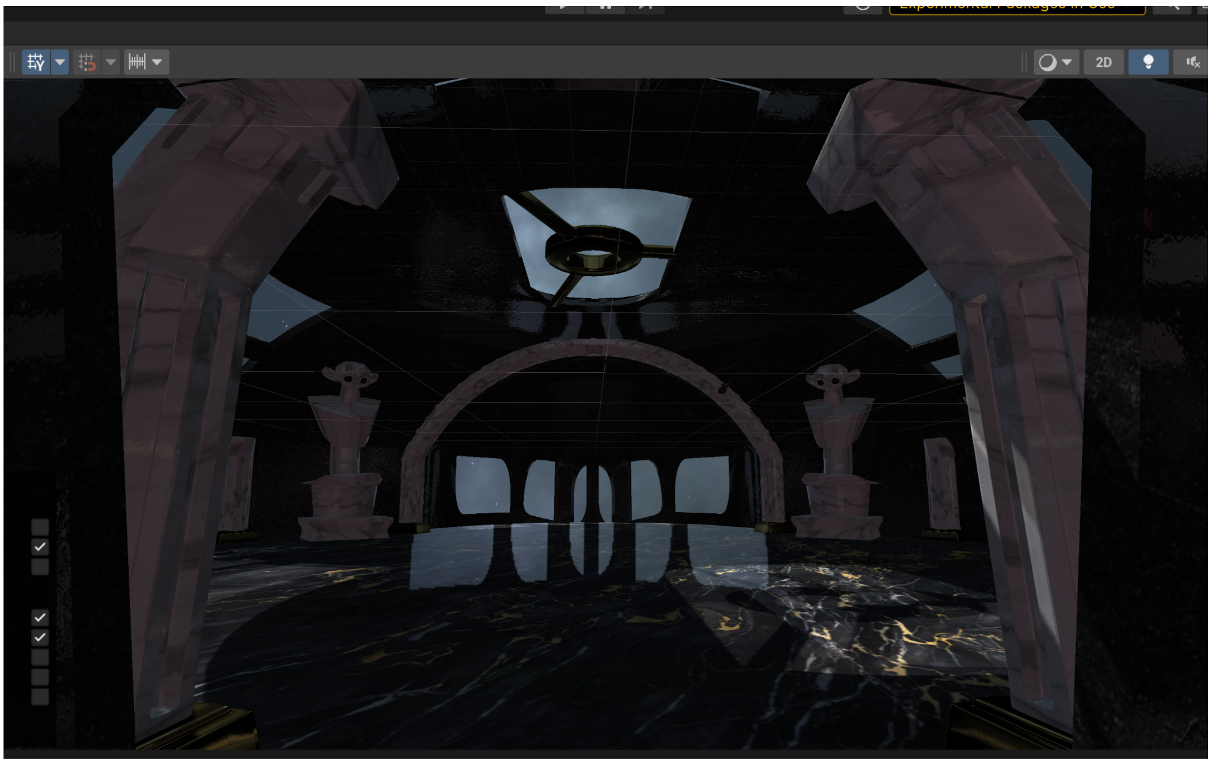
Figuring out navigation was quite a problem, costing us a lot of time and blocking the implementation of other features such as water simulation, limb separation, and other enemy types.

We modeled and rigged our next enemy in high-poly, which will be ready by the next release.

The lack of know-how in using Blender was once again a big problem, but the amount of things learnt is immense.



Design of the map



he map is also created by us in Blender. It has a waste space and no verticality for the player to move on. The Design is chosen to be darkish to make the enemies more frightening and less visible.

Target spawning

Target spawning (the targets the Agent tries to reach) should cover a large area of the map to avoid blind spots that the player could run away from.

Therefore, a mesh was created that covers the ground and its triangles are sampled to find a spawn point. The triangles are then barycentrically interpolated to find a point in between. The triangles were all of similar size, so weighting by size was not necessary.

Roller Coaster Speed

A weighted average and numerical derivative of position and rotation were used to calculate the speed and rotational velocity of the entire map/ train (moving in the update).

Bc the Bezier movement of the map does not deliver the actual 3D Velocity Vector, we had to approximate the direction. Only taking the last position minus the current and dividing it with the frame Time was not suitable due to too small numbers (termination of sum).

Weapons

For this milestone we have introduced 3 new weapons, for a total of 5. However, in the end we decided to have only 1 weapon, this one having a main and secondary fire. The main fire is always equipped and can be used by pressing the left mouse. The secondary fire can be one of 4 modes, which can be cycled by pressing E. A weapon wheel for more intuitive swapping was planned but was not ready for this milestone. This was done like this for 2 main reasons. 1: We are not experienced modelers and creating models for each unique weapon would have taken a much longer time than just one and 2: We liked the idea of having one main and secondary fire mode and this seemed like an intuitive way to achieve that. On that node, the new secondary modes added are:

1. Machine Gun: The more conventional of the weapons, it works by shooting projectiles automatically by holding the shooting button. Each projectile applies forces to the player and enemies, so while each one is relatively weak, a stream of them can easily topple an enemy and push the player a decent distance.

2. Grenade Launcher: It shoots a bouncing grenade that explodes after a delay. The grenades do not explode on contact with enemies and bounce away instead, balancing its high damage potential with it being difficult to aim precisely.

3. Singularity: Shoots a projectile which flies straight ahead while the attack button is held. Once it is released, the projectile detonates, dealing damage and pulling in all objects towards it. High potential for displacing enemies but relatively low shooting rate and damage potential.

Our priority when designing weapons was to create interesting ways for the player to mess with the enemies in more ways than simply shooting and killing them. We did add a more conventional weapon in the machine gun in case this is more intuitive to use, but for the most part we wanted each one to feel unique, allowing for more creativity.

In the future we want to add more incentives for players to swap between weapons actively. One idea we had which is not yet implemented is to add shields to enemies of the different weapon colors. These shields would need to be broken by dealing damage to them with the associated secondary weapon, otherwise the enemies will take reduced damage while the shields are active.

Weapon Visuals

A big focus of this milestone was put towards making sure that beyond being functional, the weapons actually have unique visuals. While each uses the same model, some effort was put towards making each feel different. For instance: Each secondary mode has a color associated with it, and there is a light on the weapon that shines based on the color of the secondary mode currently equipped. Also, while the model is relatively simple, it has some moving parts that allowed us to create unique shooting animations for each secondary mode (and the main fire as well). Finally each projectile also has some particle effects associated with them, making them feel different. These were, however, downloaded as assets and not made by us. For the next milestones we want to focus on adding self-made particle effects for the projectile impacts/explosions, but also for the projectiles themselves to be rendered as particle effects.

Tutorial level

We created a small scene that is meant to work as a tutorial, particularly for our prospective play-testers in the next phase. This is a very simple scene, as it had to be done quickly, but it offers a small description of what each weapon (or mode) does and provides the opportunity for them to experiment with each one without enemies coming after them. This is meant to be used as a quick introduction to the mechanics before jumping into the first level proper.

Main and Pause Menus

We've implemented the initial version of the main and pause menus. The pause menus main purpose is to give the player the capability to pause the game in order to let them take a break, quit the game, and adjust settings.

Main menu is where the player starts the game. Additionally, the player is able to adjust certain settings such as volume. It also has an option to quit the game before it starts.

Wave Spawner

Since each level represents a train compartment, we needed a way to represent multiple groups of enemies coming after the player in this cursed train ride. Wave spawner is capable of spawning multiple different enemies with varying numbers in predetermined locations in a random way.

This is achieved by providing a pool of multiple predetermined spawn points. During gameplay, one of these spawning points are randomly chosen.

Waves are predetermined. This is to ensure that each wave is balanced and gives the player a challenge in a controllable and reasonable way. A wave can spawn enemies of different kinds. Currently, it is possible to spawn all enemies in a wave at the same time or spawn them in successive manner with some time delay. The customizable wave spawner interface is prepared such that in order to provide the maximum amount of flexibility for experimentation and balancing purposes.

Original goal was to entirely randomize the spawn points as well. However, we’ve had difficulties with certain edge cases such as when enemies spawned too close to the wall, they would clip into the wall which would be detected as a collision and resolved in the next frame. This would occasionally send the newly spawned flying. In order to get to a working version of the wave spawner, we decided to put the entirely randomized version of the spawn points into our extra targets.

Win Condition

If there are no more waves left and the player has defeated enemies, then the level is completed and a victory screen is shown.

Movement Cues and Sound

Something we had in mind since the beginning, was how to make the player aware of incoming movement. Since the movement could be very extreme we thought it would be quite frustrating to get thrown around without a warning, so with no counterplay possible. That’s why we needed to find a solution, to make the player aware of the next slopes or curves. We had a lot of ideas, like making the whole level distort before the movement, or having a birds eye view on the track, but for now we stuck with two simpler solutions, we wanted to try out.

Miniature Model

The first one was to make a miniature model of the level part of the player's HUD and make it rotate like the level does. Of course it wouldn't help a lot if it would just mimic the exact movement, because at that point the player would already be in the air, but it had to be a few steps ahead of the real roller coaster. So we used a little 3D model that would check the position of the roller coaster and be a configurable amount of seconds ahead of the actual level. This worked well, but we had the issue that it was hard to determine the orientation of the 3D model. The player couldn’t tell in which direction the roller coaster was facing. So we added hints, by adding some graffiti like arrow to the texture of the 3D model. This helped a lot to know where the level was actually facing, but we still had an issue with the UV mapping which made the solution pretty much obsolete. Since the arrow was showing towards the opposite direction on the other side. Fixing the UV mapping is still in progress, but for now we want to know how helpful this tool is for the player. If it actually helps to anticipate the sudden movements, or is just a gimmick.

Alert Signs

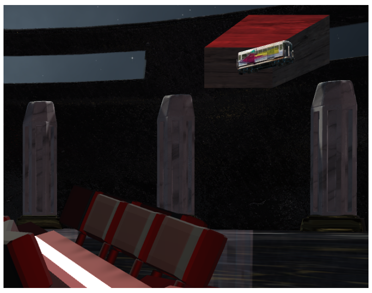
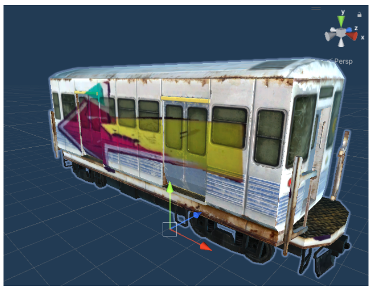
The second idea is even simpler. It basically is an alert system used right before something extreme happens. We used road signs as visual warnings, since they should be well known by most people and thus need no explanation. Additionally to the sign we used a sound cue that warns the player about incoming changes. This should alert the player enough to be ready for the maneuver. The signs obviously don't give as much information as the miniature model thus and definitely doesn't look as cool, but maybe it is enough for the player and the information overload of the 3D model doesn't help in the situation. The idea was to test those two movement indicators separately, but for now both are implemented.

Sound

We also started with the sound design, which helps a lot making the game feel more alive. First we started with the warning sounds and the gun sounds, since those are the most important for the gameplay itself. Then we added some background music to the main menu and the actual gameplay level. Since there is sound we also need a volume control, otherwise the players would blow their ears off. For now there's only a master volume you can adjust, but sliders for each, music, effects and dialog are already in the making. In the future we want to add different background music to each level and add some more dramatic music when a boss spawns, or even remove the alert sounds and just make the sound queue by using different music.

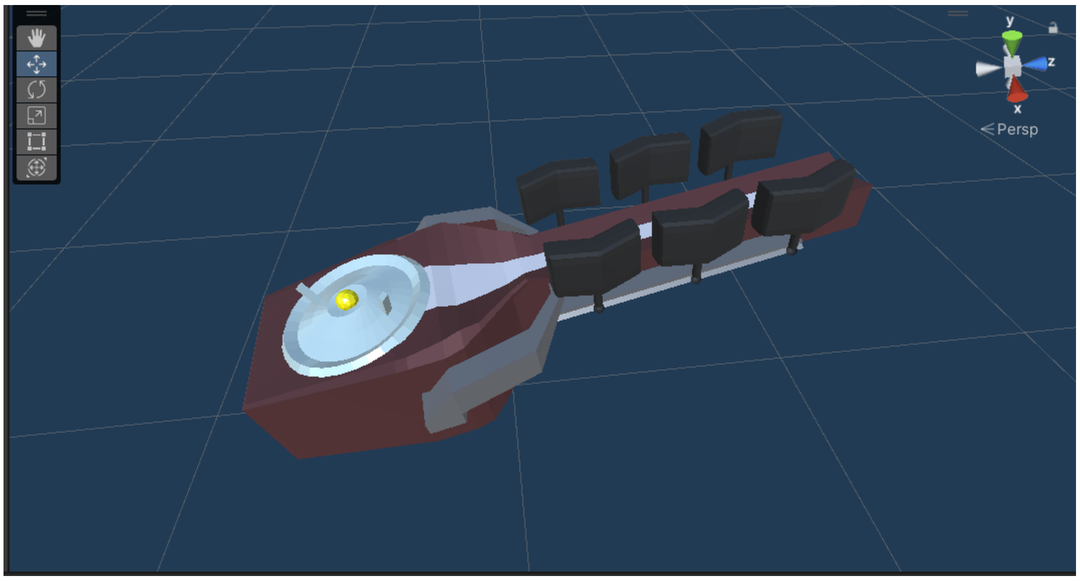
Images

Movement Indicator Model:

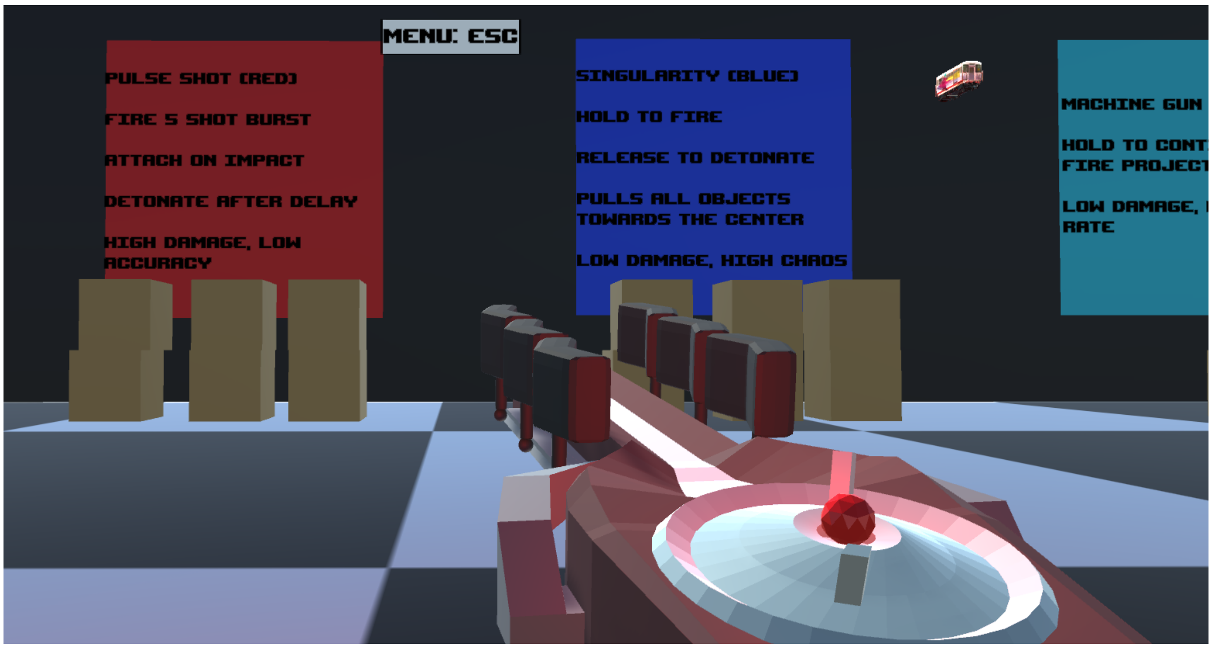



Movement Indicator Signs:

The Main Weapon:



Tutorial Level:



PULSE SHOT (RED)

FIRE 5 SHOT BURST

ATTACH ON IMPACT

DETONATE AFTER DELAY

HIGH DAMAGE, LOW ACCURACY

SINGULARITY (BLUE)

HOLD TO FIRE

RELEASE TO DETONATE

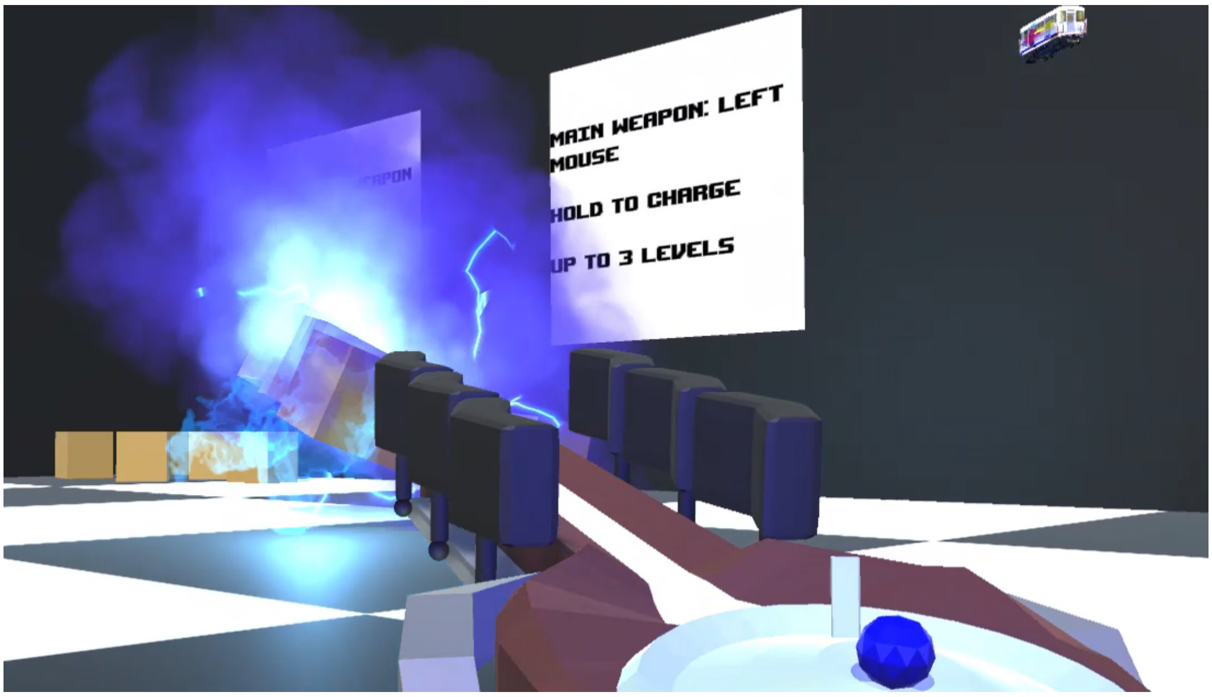
PULLS ALL OBJECTS TOWARDS THE CENTER

LOW DAMAGE, HIGH CHAOS

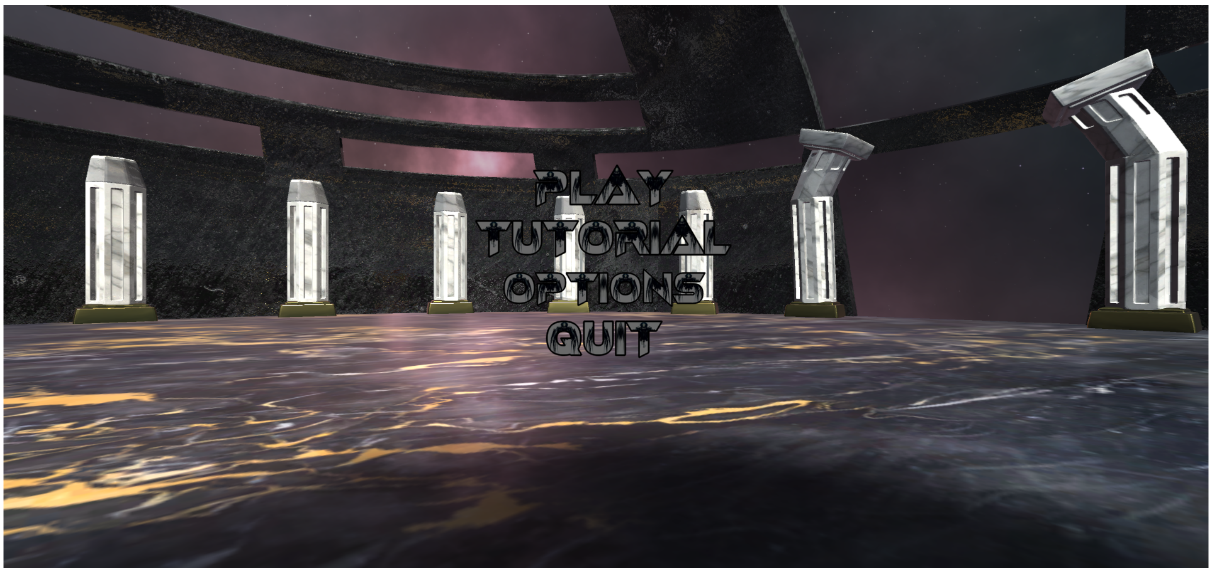
MACHINE GUN

HOLD TO CONT FIRE PROJECT

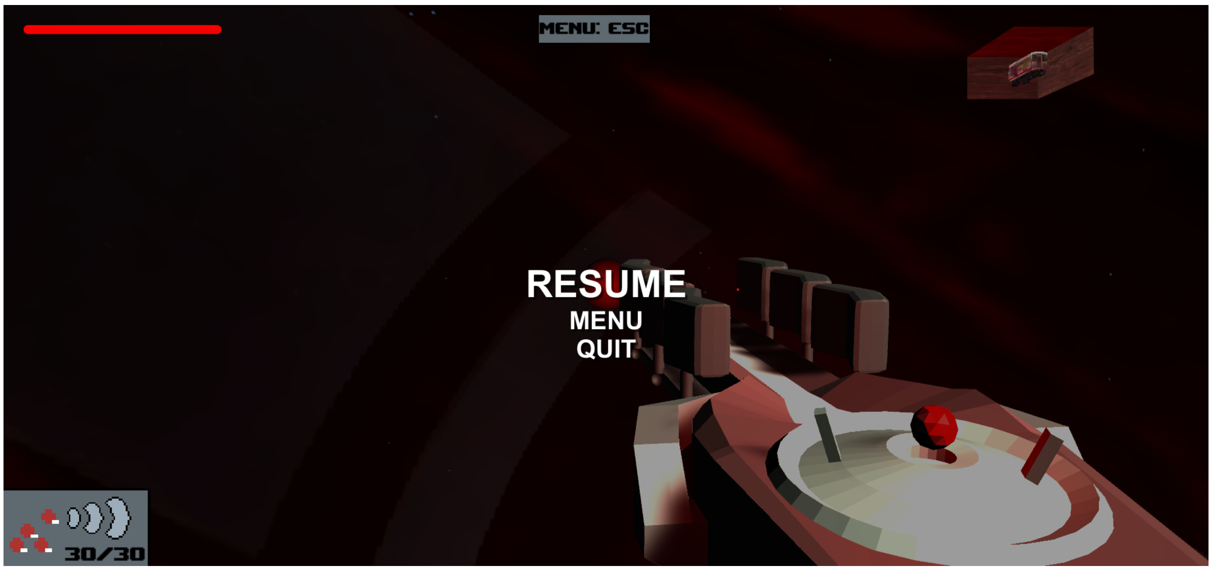
LOW DAMAGE, RATE



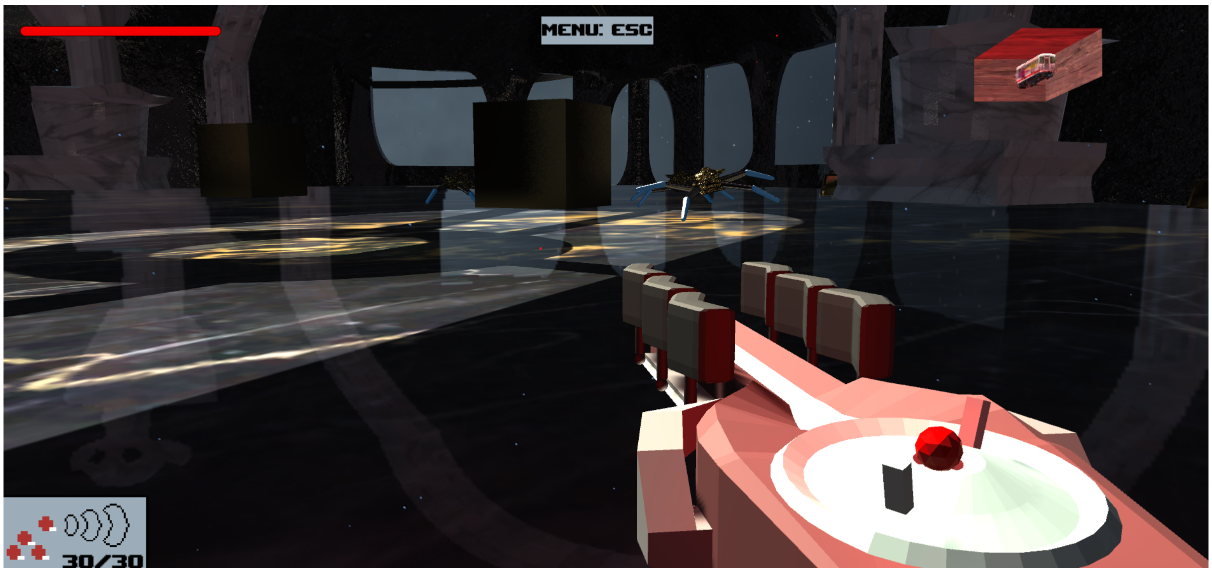
Main Menu:



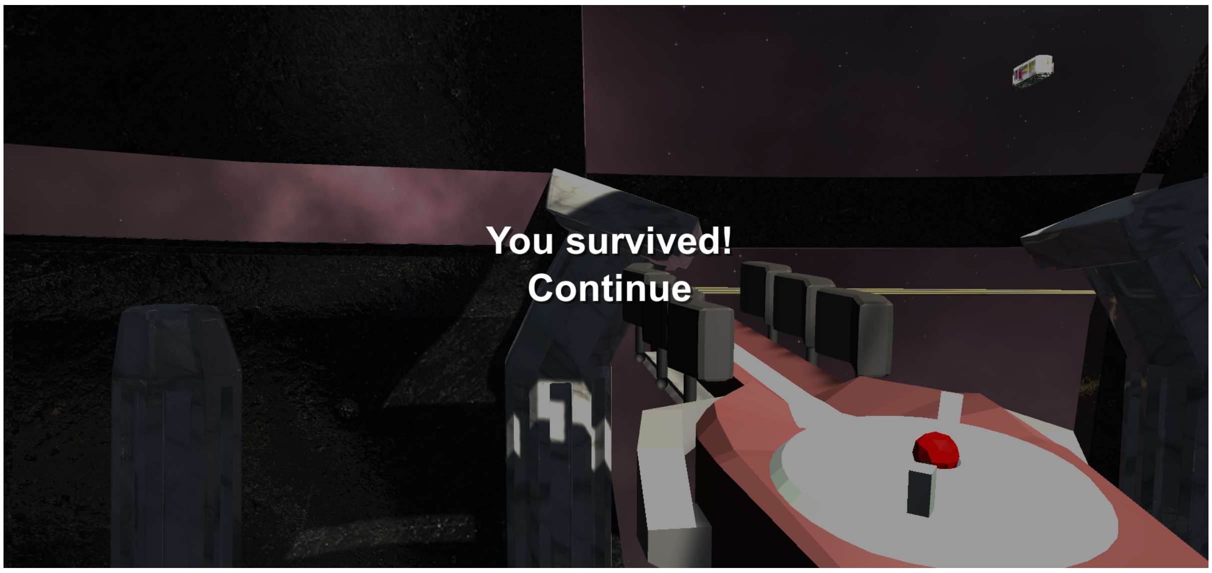
Pause Menu:



Wave Spawner:



Win Screen:



Goal Statuses

Functional Minimum (Layer 1) Goals	Goal Statuses
One type of enemy to shoot at that reacts in physically plausible way	Completed
Basic gun play	Completed
Basic levels	Completed
Rapid movement of the environment (only a few like e.g. rotation)	Completed
Rigidbody physics	Completed
Keyboard and mouse control	Completed

Low Target (Layer 2) Goals	Goal Statuses
More complex level movements (g-force, free fall, sudden turns, acceleration)	Completed
Weapons have accurate knockbacks and recoils (such as rocket jumps)	Completed
Physically-based player controls (impulses, force, momentum)	Completed
Ability to control level movement (to some degree)	Delayed
Basic UI	Completed

Desirable Target (Layer 3) Goals	Goal Statuses
Sound design (movement of the roller coaster, music, weapon sounds etc.)	In progress
Different Projectile weapons (push enemies, break things, cannon balls, spikes etc.)	Completed
Different enemy types	In progress
Water physics	In progress

Breakable objects (glass, boxes, destructible environment)	Delayed
Nice looking UI	In progress

High Target (Layer 4) Goals	Goal Statuses
Story and characters	In progress
Physics based puzzles (environmental, use the movement of the level)	Delayed
Good weapon effects and animations	In progress
Nice art and assets	In progress
Well trained AI for bosses	Delayed
Game controllers (such as Xbox and Playstation controllers)	In progress

Extra (Layer 5) Goals	Goal Statuses
Cutscenes	Work on it not started yet
Multiplayer / co-op modes	Work on it not started yet
First-player animations (arms, legs, doom-like kills for enemies)	Work on it not started yet
More fluid simulations etc.	Work on it not started yet
Shot off body parts move after they fall to the ground etc (for example like zombie arms).	Work on it not started yet
Weapons that use fluid dynamics like a water jet gun etc.	Work on it not started yet
Weapon progression such as adding mobility skills to weapons (crossbow provides a hook etc.)	Work on it not started yet