VRisk
Project History

Jade Eames – Simone Farinelli

VRisk is a VR simulation designed for risk assessment which uses real-world data to recreate a disaster scenario where an urban area is struck by an earthquake.

In the simulation the user will need to navigate through the urban area and make it safely to a safe zone, while important data is being saved regarding the user's movement and pace.

The purpose of VRisk is to serve as a proof of concept, showcasing a potential product that focuses on validating the feasibility of it being transformed into a fully-fledged application.

The essential requirements needed for this proof of concept are the following:

- Movement with gestures.
- Urban Area Layout from data provided.
- Earthquake shaking.
- Buildings taking damage.
- Two safe-zones.
- Data Tracking.

This document will cover the history of the development process and share some insight on the underlying game-logic.

First Milestone

The goal of the first milestone was to have an initial mock-up with most of the essential features ready and running, to make sure we were taking the right path before advancing in development.

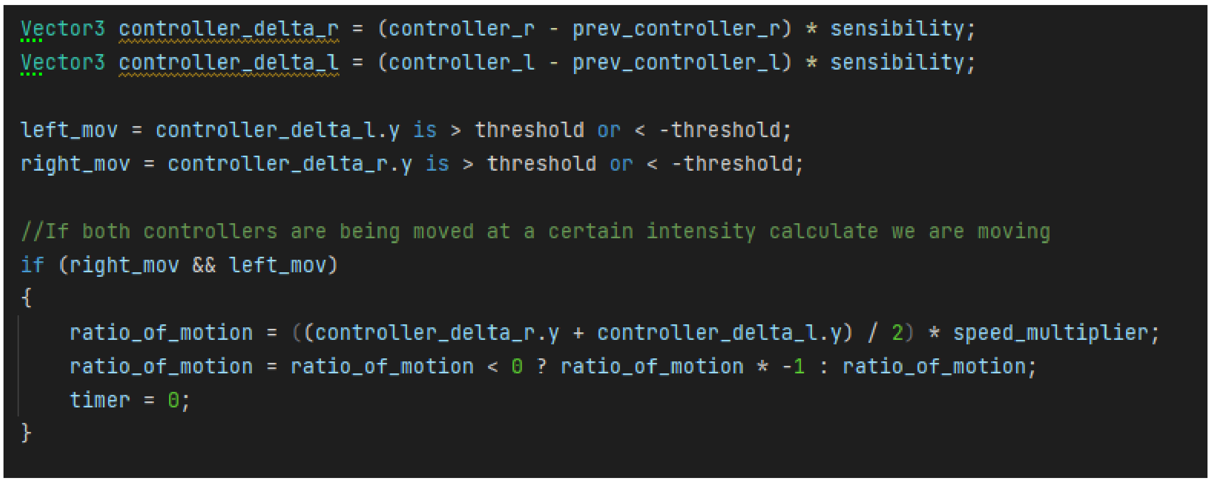
To reach our first milestone we worked on the following:

Movement:
Movement was the first component on which we worked on during the development of VRisk, being one of the essentials of the simulation therefore being implemented early.

We first started by coding the system that allows the player to move via gestures, we do this by reading the values of the accelerometers in the hand controllers each frame and using the last saved value to calculate how much the values have been altered between frames.

Then a threshold value can be used to allow movement to be detected only after a certain amount of motion, prevent involuntary movement.

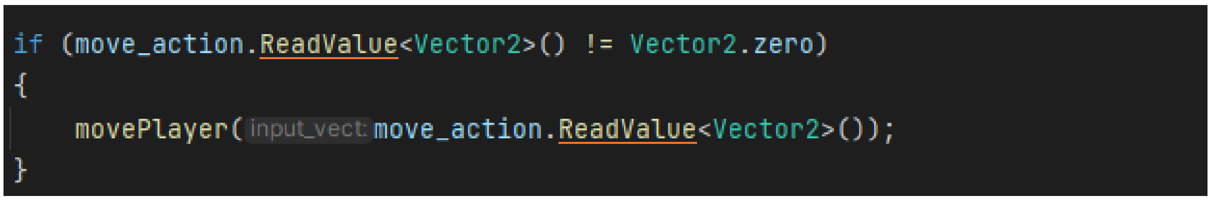
A sensibility value has also been implemented to allow the user to increase the magnitude of the movement, to allow for more sensible controls while still having a marginal threshold value.



```
Vector3 controller_delta_r = (controller_r - prev_controller_r) * sensibility;
Vector3 controller_delta_l = (controller_l - prev_controller_l) * sensibility;

left_mov = controller_delta_l.y is > threshold or < -threshold;
right_mov = controller_delta_r.y is > threshold or < -threshold;

//If both controllers are being moved at a certain intensity calculate we are moving
if (right_mov && left_mov)
{
    ratio_of_motion = ((controller_delta_r.y + controller_delta_l.y) / 2) * speed_multiplier;
    ratio_of_motion = ratio_of_motion < 0 ? ratio_of_motion * -1 : ratio_of_motion;
    timer = 0;
}
```

Later, we also added an in-game slider to adjust the sensibility (In milestone 3).

We also added the option to move with the thumb stick by reading its own data from the controller and using it to determine which direction is the stick pointing at, then move the player if the input is different than 0.

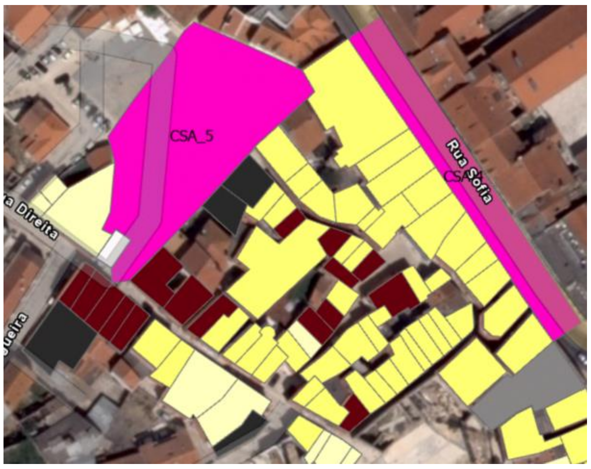


```
if (move_action.ReadValue<Vector2>() != Vector2.zero)
{
    movePlayer( input_vect: move_action.ReadValue<Vector2>());
}
```

Lastly, we added a fixed speed cap to prevent a movement method to move faster than the other, preventing invalid data.

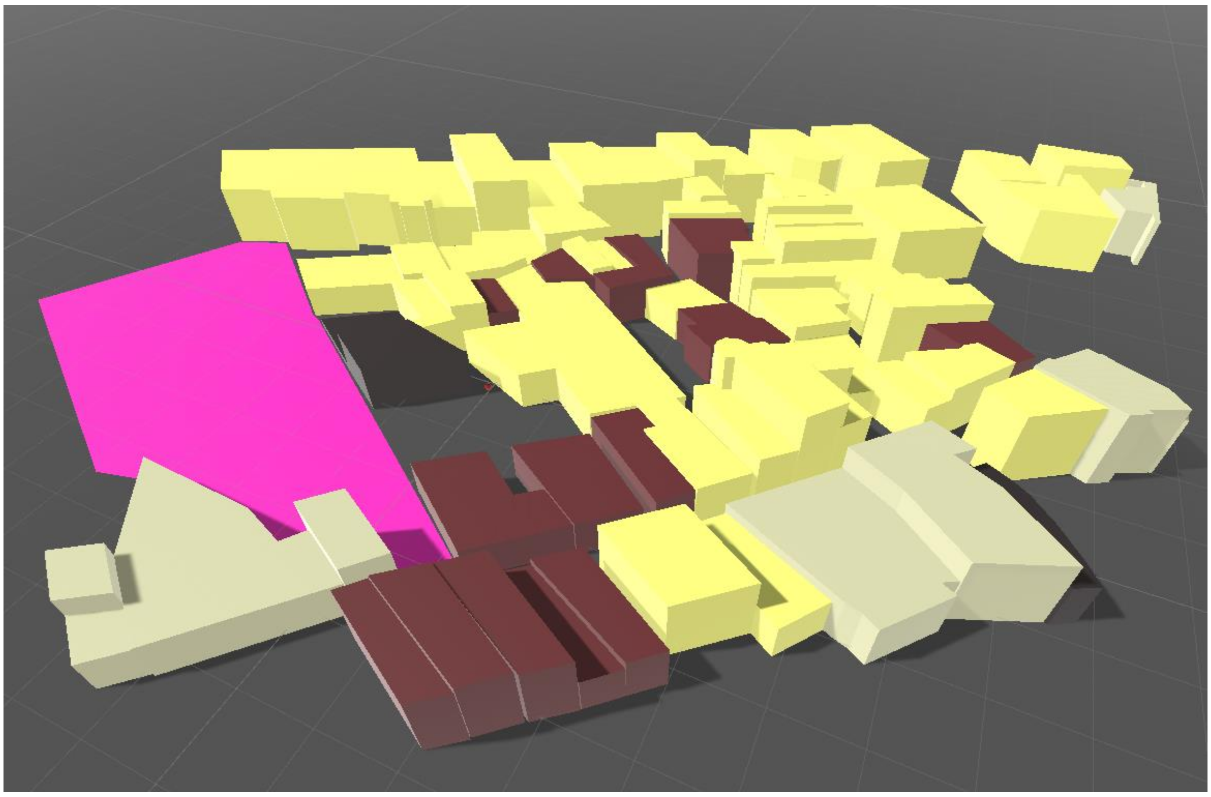
<u>Urban Area</u>:
Once movement was sorted, we started building the simulation area.

Using the provided map showcasing the danger level and layout of the urban area that we had to move into Unity, we were able to locate it on google earth to retrieve the measures, this allowed us to proportionally scale the buildings when inside unity.



As we did not had models yet, we built the map by using "boxes" the shape, size and height of the buildings from the google map data, Unity by default does not allow this, therefore we used a plugin called Pro Builder which allowed us to make this complex shapes.

We also coloured and tagged the buildings with their respective danger level; this will be helpful in telling them apart while they have no models or using the tag to quickly access them.

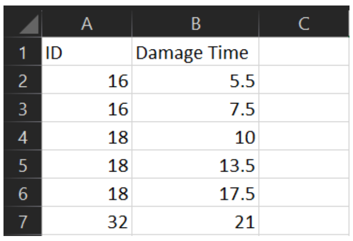


The map was then tested inside the headset, and we were able to do a bunch of other fixes to make it feel well proportioned.

<u>Timeline & Damaging Buildings</u>:
With the map laid out and the buildings in place we started working on a possible way to prompt the buildings to take damage, we decided to try with a .CSV functioning as a timeline.

This allows for the timeline to be edited in Excell, making quick changes easy and straight forward.

Our first pitch of the timeline was the following:



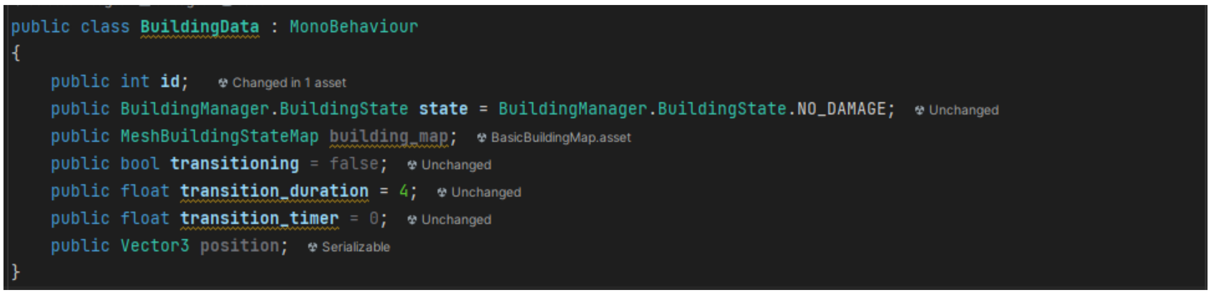
	A	B	C
1	ID	Damage Time	
2	16	5.5	
3	16	7.5	
4	18	10	
5	18	13.5	
6	18	17.5	
7	32	21	

The timeline would contain two data per row, and each row would be an entry that will prompt a building to take damage.

The first value, the ID would be what the timeline provides to locate a building, the second, being damage time, would be after how many seconds from the start that building would take damage.

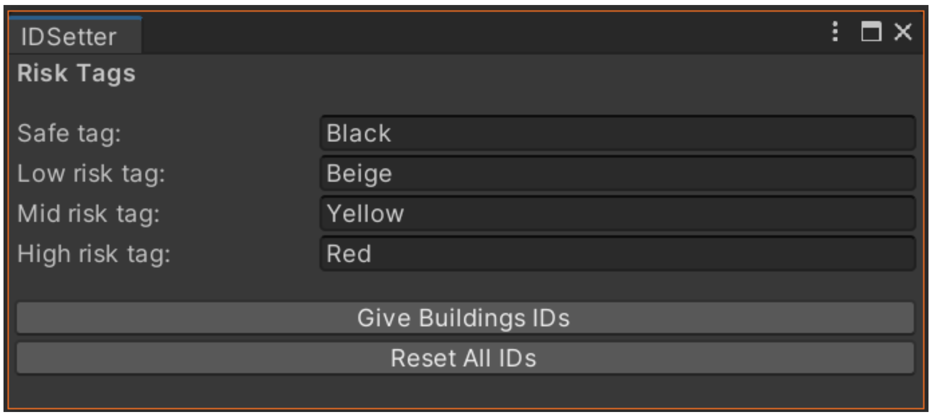
However, the use of IDs meant that we had to find a way to assign IDs to buildings in the scene.

To do so we created a script called “Building Data” that could be attached to a building, not only holding its ID but also valuable information that we may need further in development.



```
public class BuildingData : MonoBehaviour
{
    public int id;    Changed in 1 asset
    public BuildingManager.BuildingState state = BuildingManager.BuildingState.NO_DAMAGE;   Unchanged
    public MeshBuildingStateMap building_map;   BasicBuildingMap.asset
    public bool transitioning = false;   Unchanged
    public float transition_duration = 4;   Unchanged
    public float transition_timer = 0;   Unchanged
    public Vector3 position;   Serializable
}
```

Then we made a tool, which we called the "ID Setter" to quickly assign IDs to all the buildings, this tool would then evolve to become a very extensive and powerful tool by the end of the project, but its first iteration looked like this:

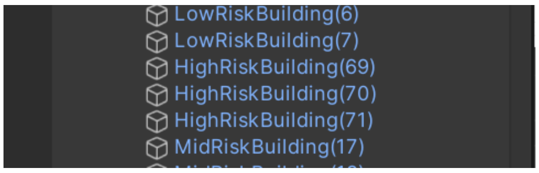


IDSetter

Risk Tags

Safe tag:	Black
Low risk tag:	Beige
Mid risk tag:	Yellow
High risk tag:	Red

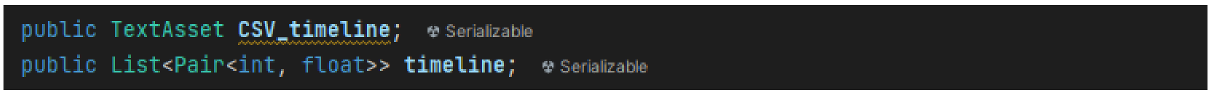
The ID setter would also take the risk tag of each building and rename it accordingly.



```
LowRiskBuilding(6)
LowRiskBuilding(7)
HighRiskBuilding(69)
HighRiskBuilding(70)
HighRiskBuilding(71)
MidRiskBuilding(17)
```

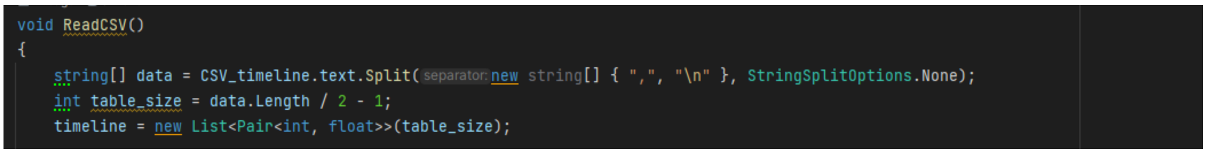
With the building's IDs in place, we started working on a script that would read the timeline, track time and prompt the buildings that need to be damaged, this script would the be called Timeline Manager.

This first iteration of the Timeline Manager uses a TextAsset to serialize the .CSV inside the project and then use a list of pairs to store ID and Damage Time:



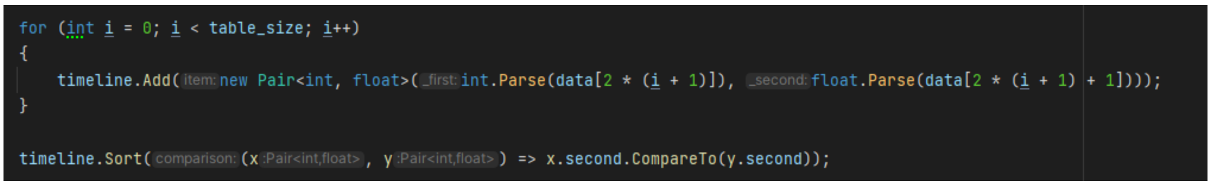
```
public TextAsset CSV_timeline;   Serializable
public List<Pair<int, float>> timeline;   Serializable
```

We have chosen to use a .CSV because its internal values are separated by commas, as the file format suggests; This allows us to easily split the text and retrieve the single values inside Unity, since TextAssets have a built-in functionality to be split.



```
void ReadCSV()
{
    string[] data = CSV_timeline.text.Split(separator: new string[] { ",", "\n" }, StringSplitOptions.None);
    int table_size = data.Length / 2 - 1;
    timeline = new List<Pair<int, float>>(table_size);
```

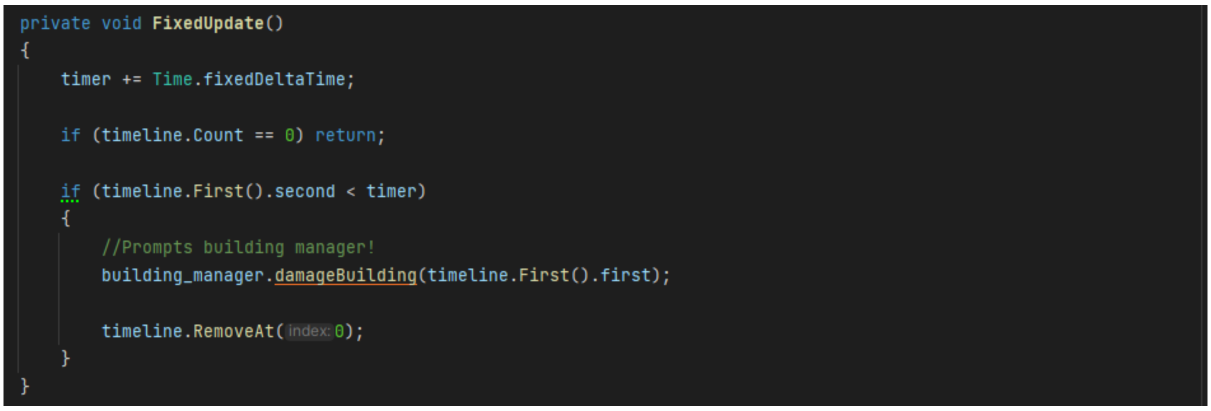
Following, all the data is moved inside the timeline and is then sorted by time, this is important as it will allow some script optimization that will be discussed shortly.



```
for (int i = 0; i < table_size; i++)
{
    timeline.Add( item: new Pair<int, float>( _first: int.Parse(data[2 * (i + 1)]), _second: float.Parse(data[2 * (i + 1) + 1])));
}

timeline.Sort( comparison: (x :Pair<int,float> , y :Pair<int,float> ) => x.second.CompareTo(y.second));
```

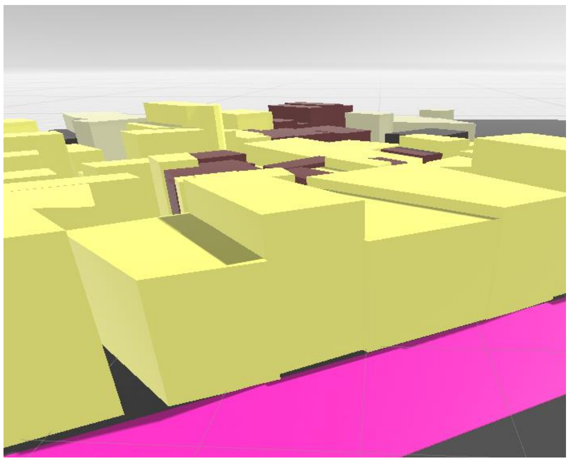
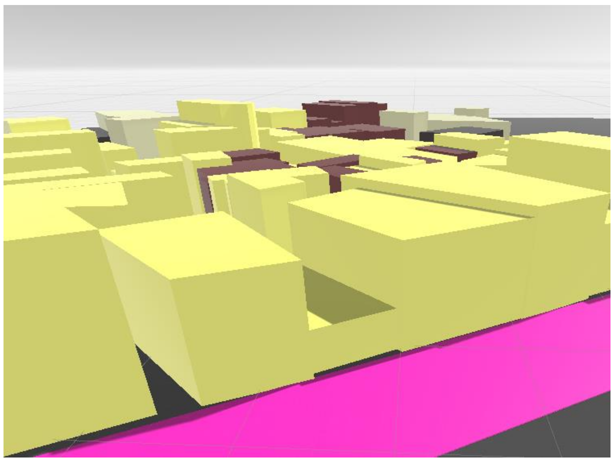
After the timeline is sorted and saved, a timer starts alongside the scene, every time it gets increased the script will check if the "Damage Time" value of the first entry of the timeline is lower than the timer, if it is, it will damage the building at the ID specified in that entry and then remove it (the entry).

The sorting is what allows this to work, as all the entries will be in chronological order, once the first is removed, the next one will move as the new first entry. This lets the script check that first entry only once.



```
private void FixedUpdate()
{
    timer += Time.fixedDeltaTime;

    if (timeline.Count == 0) return;

    if (timeline.First().second < timer)
    {
        //Prompts building manager!
        building_manager.damageBuilding(timeline.First().first);

        timeline.RemoveAt( index: 0);
    }
}
```

When a building is prompted, we run a script that decreases the height of that building, this works as a temporary visual clue to showcase damage until the particles and modes are in place.

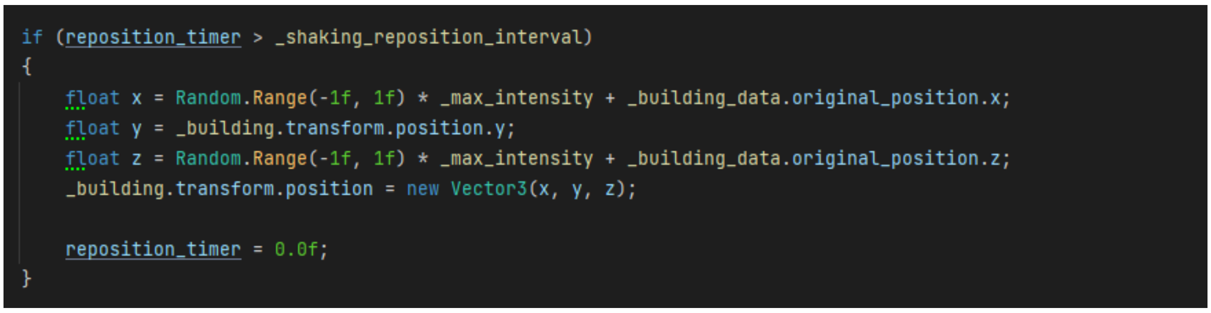



<u>Shaking</u>:
The last component we worked on for the first milestone is the implementation of the earthquake's shake, the goal was to add a shake that would feel somewhat similar to a real earthquake.

Firstly, we worked out a way to shake buildings, to do so we made a script that takes the position of a building and quickly alters it slightly every frame, this gives a fluid shaking effect, although it results too intensive and unnatural.

Given the promising result of using this method, we expanded it trying to fix the evident problems, first we added an “Intensity” variable that would alter how much the position of the building changes every time it shakes, and secondly, we added a “reposition interval” variable that would act as a timer between each relocation, instead of doing it each frame.

This revealed to work great, and the result is a shake that can be easily tweaked and has a realistic feeling to it, unfortunately to demonstrate we are limited to providing a snippet of the code instead of a GIF/Video.

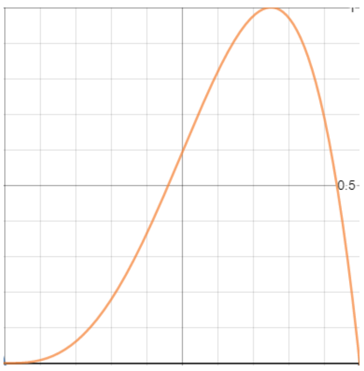


```
if (reposition_timer > _shaking_reposition_interval)
{
    float x = Random.Range(-1f, 1f) * _max_intensity + _building_data.original_position.x;
    float y = _building.transform.position.y;
    float z = Random.Range(-1f, 1f) * _max_intensity + _building_data.original_position.z;
    _building.transform.position = new Vector3(x, y, z);

    reposition_timer = 0.0f;
}
```

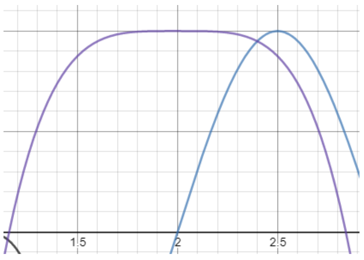
Next, to enhance it, we decided to implement an intensity curve. This feature enables precise adjustments of the earthquake's intensity throughout its duration.

A finely tuned curve would allow a smoother transition with a gradual increase and decrease in intensity at the beginning and end of the earthquake.

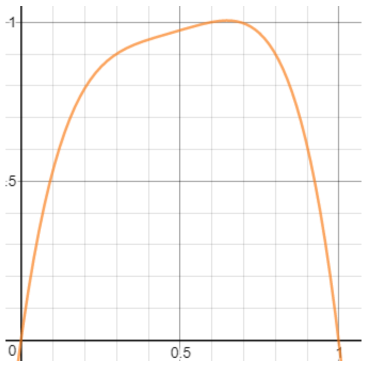
We tried different intensity curves with different outcomes:



With this intensity curve the results were promising but far from realistic, the earthquake would have a very slow ramp-up to then quickly die down once reached peak intensity.



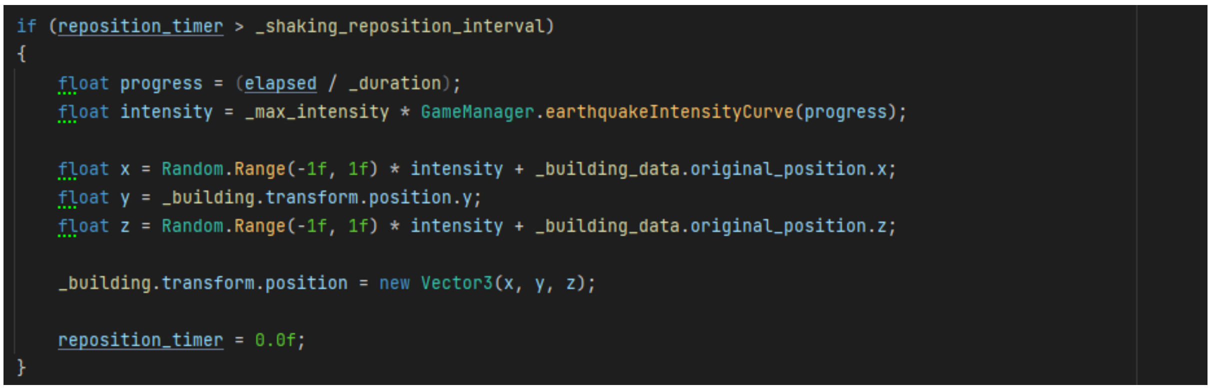
This next curve (In purple) felt more realistic and gave a better feeling to the shaking, however the ramp-up and slow-down were bland and predictable.



This was our last iteration, it provides a quick ramp-up to a strong quake while slowly increasing until reaching peak intensity, then it would first decrease slightly then quickly die off.

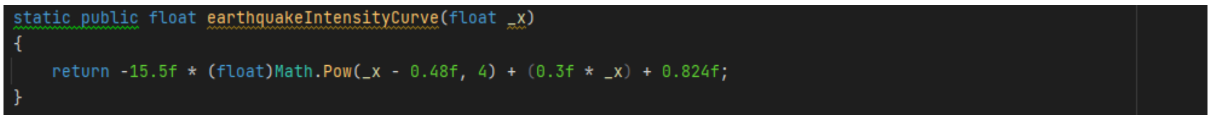
While far from being very realistic as a real quake, it gave the shaking a procedural and more natural feeling.

To implement this inside Unity, we modified the shaking script to track progress through the earthquake, then use this to determine where in the curve the quake is to then modify its intensity accordingly.



```
if (reposition_timer > _shaking_reposition_interval)
{
    float progress = (elapsed / _duration);
    float intensity = _max_intensity * GameManager.earthquakeIntensityCurve(progress);

    float x = Random.Range(-1f, 1f) * intensity + _building_data.original_position.x;
    float y = _building.transform.position.y;
    float z = Random.Range(-1f, 1f) * intensity + _building_data.original_position.z;

    _building.transform.position = new Vector3(x, y, z);

    reposition_timer = 0.0f;
}
```

The function returning the intensity is the following, it uses the progress percentage to return an intensity from 1 to 0 in relation to the curve provided.



```
static public float earthquakeIntensityCurve(float _x)
{
    return -15.5f * (float)Math.Pow(_x - 0.48f, 4) + (0.3f * _x) + 0.824f;
}
```

Second Milestone

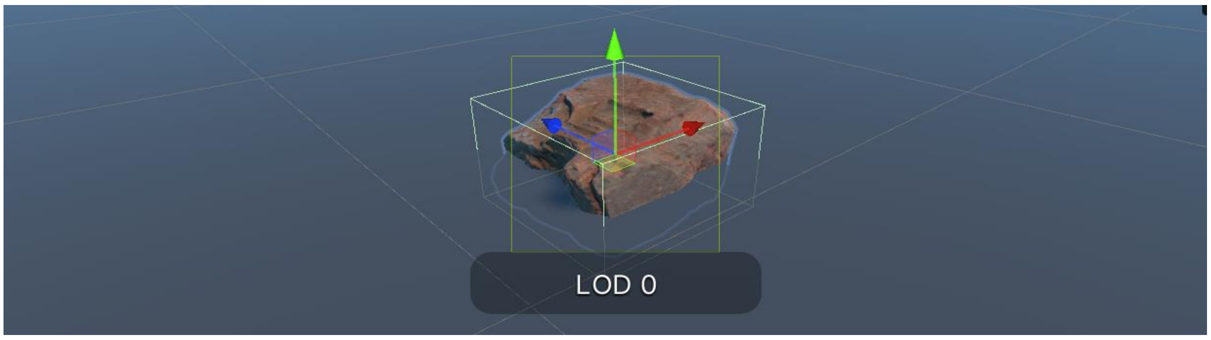
After the first milestone was successful, our goal for the second milestone was to start to turn the project into something more than a mock-up, we implemented menus and other features to enhance the experience such as sound, particles, debris, menus and more.

To reach our second milestone we worked on the following:

Debris:
We started working onto enhancing the experience, adding falling debris during and after the earthquake buildings was our first choice.

Before developing the specific tools, we downloaded a free debris model from the internet (license free) as we did not have a 3D artist yet and used it as our main debris model.

To increase performance, we also added a LOD (Level of Detail) system that would render the model in lower quality when far away.



To manage debris, we created a debris manager alongside a visual tool that would allow us to tweak debris and its generation.

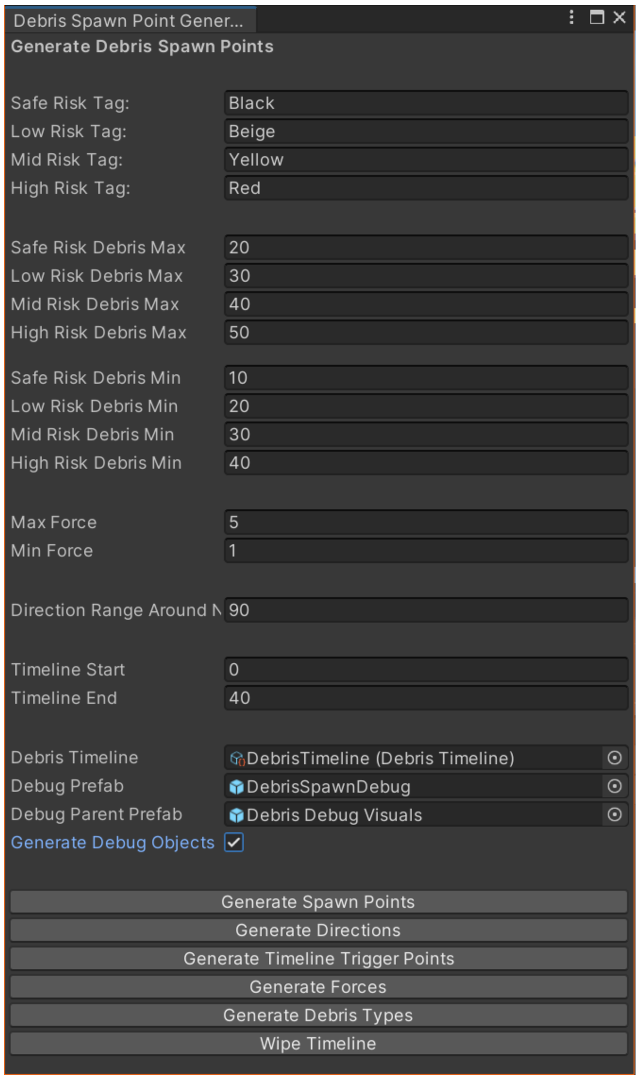
To make the system flexible with all types of meshes, the tool will use an object's mesh to map where the debris would spawn, then the system would take a random point on the surface of the mesh and use it as a possible spawn point for debris.

However, due to the project's nature in collecting data, this would cause the debris generation to be random between each simulation, we fixed this issue by adding a second timeline that would make debris behaviour deterministic, unlike the first timeline introduced previously for buildings, this is internally generated by our debris editing tool and only serves as an internal sheet for debris spawning, it can't be edited by the user.

As one of the last steps, we optimized performance by adding pooling, which is a technique used to save resources, a fixed number of objects (is this case, debris) is created and maintained, then when one needs to be spawned, it gets borrowed from the pool rather than being created on runtime.

We finalized it by adding specific settings to tweak pooling, alongside the possibility to specify the tag on the buildings, maximum force upon spawn, angle of spawn and the timeframe of the earthquake.

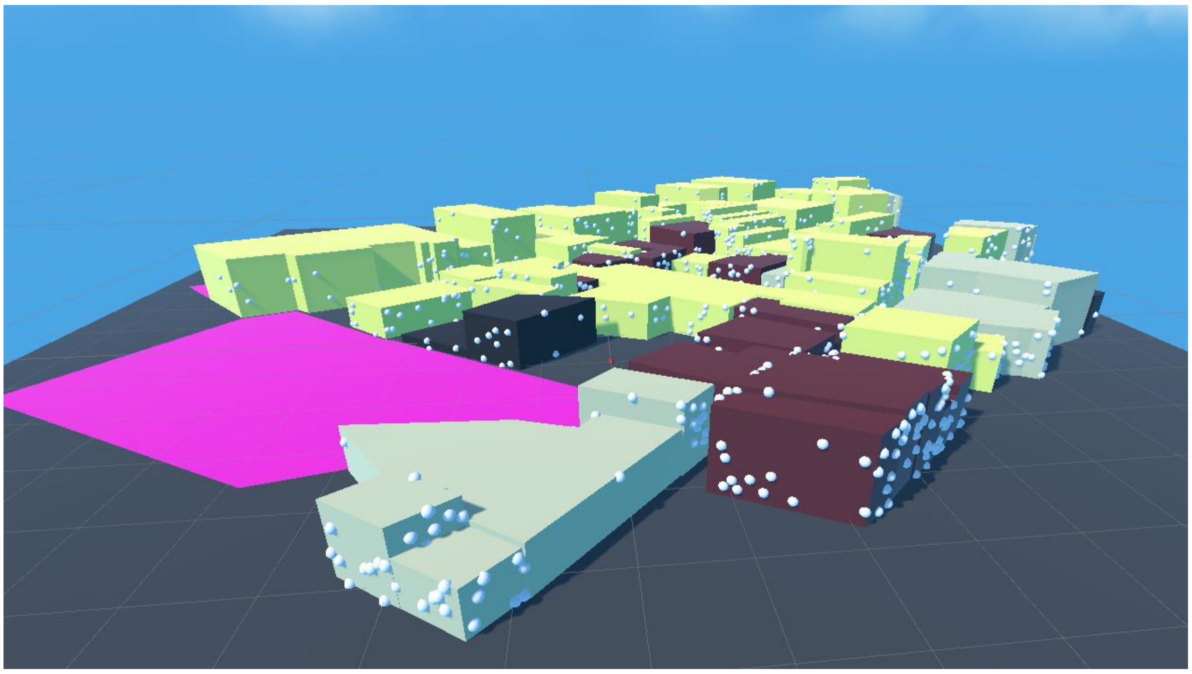
The first iteration of the Debris Data Editor tool looked like the following:



Generate Debris Spawn Points

Safe Risk Tag:	Black
Low Risk Tag:	Beige
Mid Risk Tag:	Yellow
High Risk Tag:	Red
Safe Risk Debris Max	20
Low Risk Debris Max	30
Mid Risk Debris Max	40
High Risk Debris Max	50
Safe Risk Debris Min	10
Low Risk Debris Min	20
Mid Risk Debris Min	30
High Risk Debris Min	40
Max Force	5
Min Force	1
Direction Range Around N	90
Timeline Start	0
Timeline End	40
Debris Timeline	DebrisTimeline (Debris Timeline)
Debug Prefab	DebrisSpawnDebug
Debug Parent Prefab	Debris Debug Visuals
Generate Debug Objects	✓

Alongside the options to generate the debris data are also present some debug features, these are visual clues that we added for development to visualize where on the mesh of the buildings the debris were going to spawn.

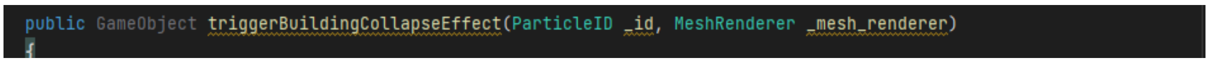


Lastly, by adding a collision box to the debris and the player, we can detect when a debris has fallen on the player, and then use a script to raise a flag inside our data folder, we'll use this further on in development to determine if the player has survived through the simulation.

Particles:
To handle particles easily, we made a particle manager, the requirement was for the system to be able to apply particles to any game object in the scene.

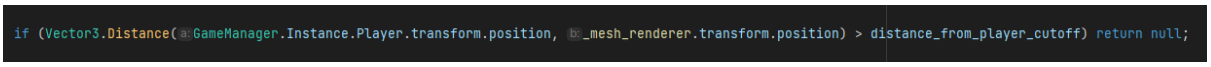
Before we worked out how to apply particles to objects, we spent some time on optimization, this is because in Unity particles are summoned using particle emitters, and having too many in the scene would cause performance issues, once again, we used pooling to pool the particle emitters, which would also limit their amount.

We quickly discovered that using the mesh renderer of an object to trigger particles would work very well with the buildings, therefore we created a function that would focus in doing so.



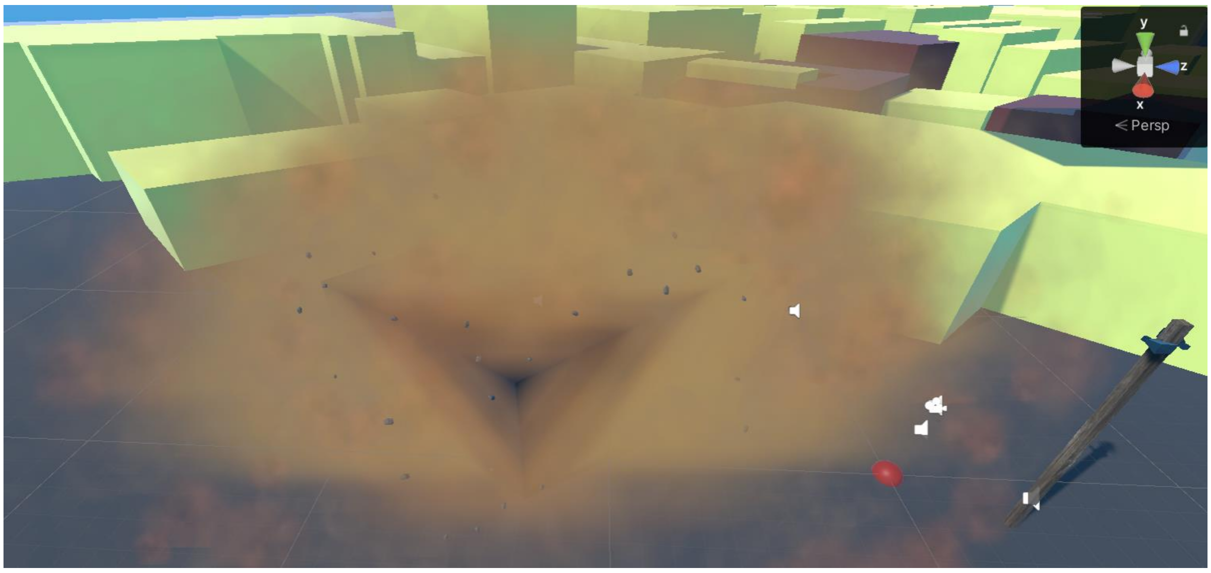
```
public GameObject triggerBuildingCollapseEffect(ParticleID _id, MeshRenderer _mesh_renderer)
{
```

Having a dedicated function to apply particles to buildings being damaged also allowed us to add some optimization, such as keeping track of the distance between the player and the building that would trigger the particle effect; If the player is out of range, the effect won't be triggered.



```
if (Vector3.Distance(a: GameManager.Instance.Player.transform.position, b: _mesh_renderer.transform.position) > distance_from_player_cutoff) return null;
```

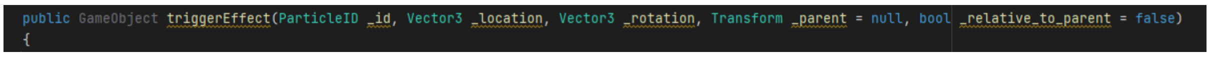
Lastly, we spent some time tweaking the particles and the function to obtain a visually pleasing effect that would help to cover eventual mesh changes when updating the damage of the buildings.



The edge of the mesh would often be visible, but this would not turn out to be an issue once the models are in place.

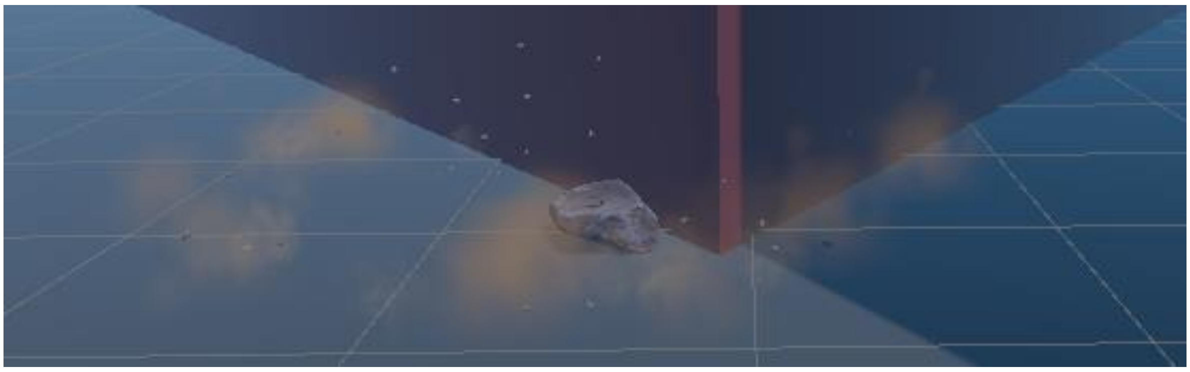
With the function to apply particles to the buildings in place, we started working on a function to apply particles anywhere in the scene or to other game objects.

We achieved this by creating a function that would take a point in 3D space to summon the particles at, then can also use additional parameters such as rotation and parent object to inherit properties such as position and rotation to the object we desire to attach the particles to.



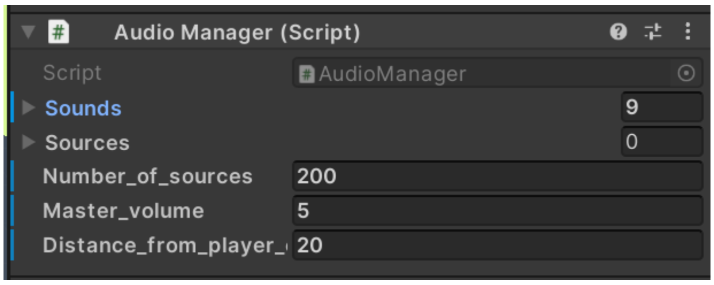
```
public GameObject triggerEffect(ParticleID _id, Vector3 _location, Vector3 _rotation, Transform _parent = null, bool _relative_to_parent = false)
{
```

This worked well with the debris, as we can now easily tweak where in relation to the fallen debris the particles will spawn.



<u>Audio</u>:
Audio is a fundamental component for a simulation, and our goal was to integrate spatial audio. This approach will allow us to create a three-dimensional sound environment, emulating a surround sound system for a more immersive experience.

We created an Audio Manager that uses Unity’s audio sources, a source is an audio emitter that can be placed in the 3D environment.

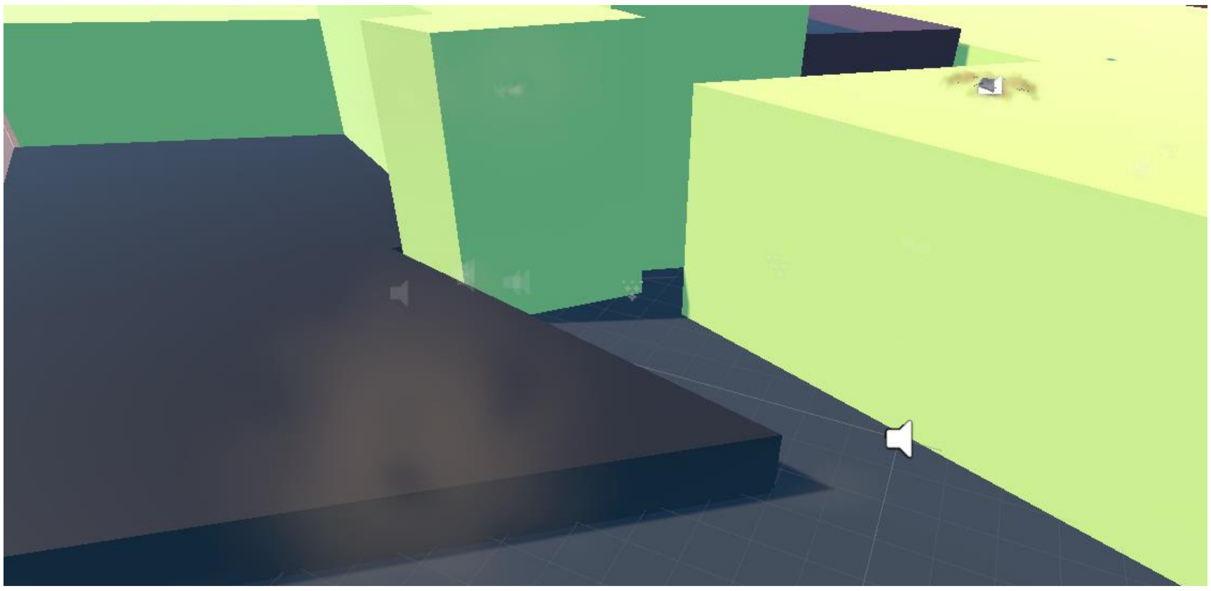

Audio Manager (Script)

Script	AudioManager
Sounds	9
Sources	0
Number_of_sources	200
Master_volume	5
Distance_from_player_	20

The system works by pooling (Once again!) a predefined number of sources in the scene, then via a function the audio manager can be prompted to attach one of the sources onto an object in the game scene and reproduce a sound.

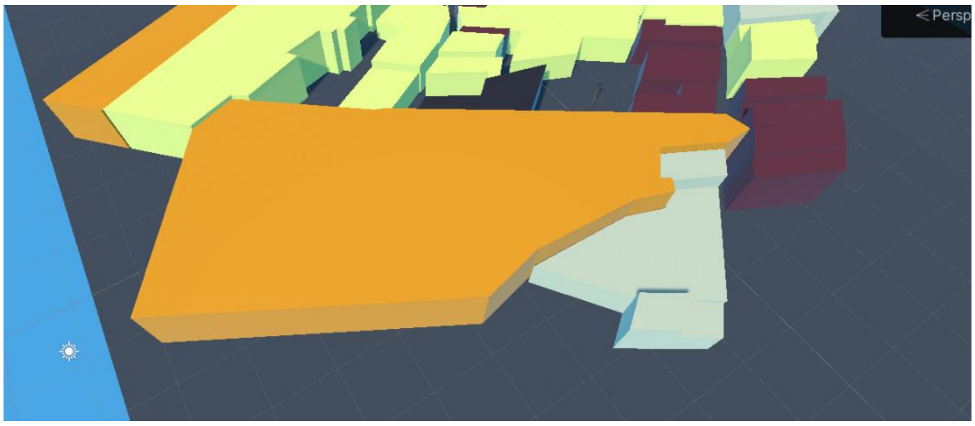
This allows for any object in the scene that needs to produce a sound to be able to easily prompt the audio manager that will attach a source to the object playing the desired sound in 3D space.

In the scene editor, using gizmos the audio sources can be easily seen, this allowed us to determine if the system was working correctly.

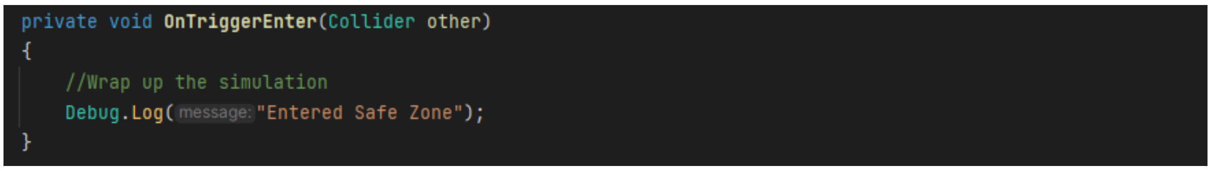


The audio system initially also had an EAS (easly warning system) feature that would allow a choosen siren to warn the player of the incoming quake, allthough this was scrapped as the location where the simulation data have been taken from does not have one.

<u>Safe Zones</u>:
The end goal for the simulation is for the player to reach the safe zones safely and wrap up the experience, to achieve this we started by outlining the safe zones using a Pro Builder Mesh:



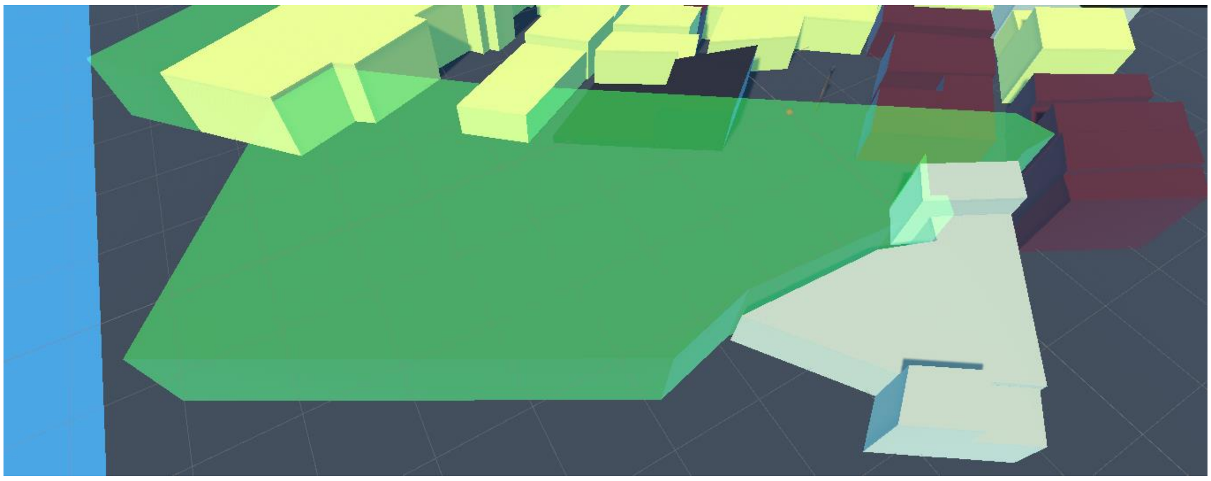
Using a visible object allows for a better understanding of the zone that will function as a safezone, the next step is to make the area non-solid and add a trigger to flag if the player has entered the area, this can be easily done by turning on the option “Is Trigger” in the safe zone’s collider options, then via a custom script we can add functionality for when this happens.



```
private void OnTriggerEnter(Collider other)
{
    //Wrap up the simulation
    Debug.Log( message: "Entered Safe Zone");
}
```

We also added a visible particle effect and sound to signal the entrance in the safezone, but the function to end the simulation will still remain unimplemented until the system to manage the menus and scenes is implemented.

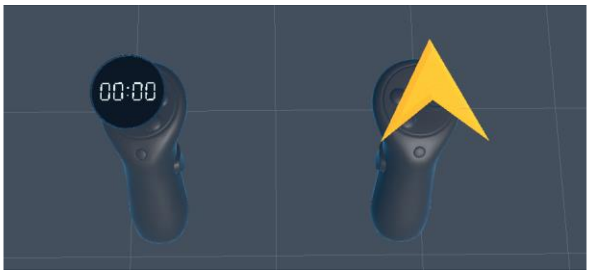
Later we recolored the safe zone and made it semi-transparent to leave a visual clue for the player to see.

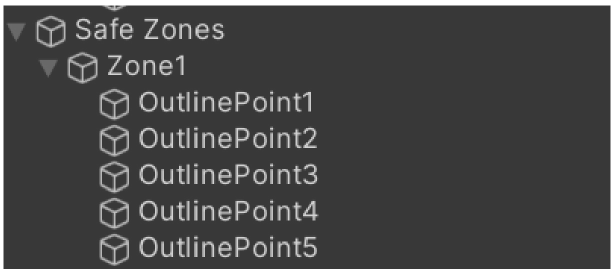


However, while the visual clue helps in locating the safe zone when on sight, it is still difficult to navigate to one trhough the map, specifically when the earthquake is happening, we addressed this issue by implemented a navigation system, which will direct the player to the closest safezone.

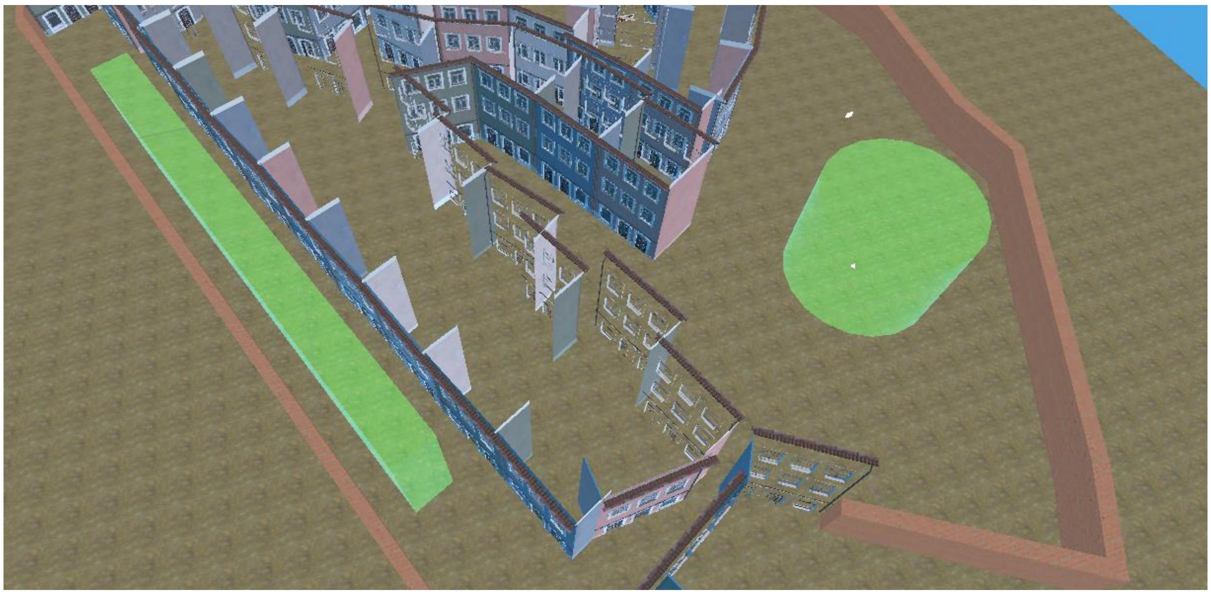
The system provides a navigational arrow on top of the player’s right hand controller, which will always point at the closest safezone. To allow for better precision we populated the safe zones with points that will mark its outline for the arrow.

While off topic, we used the same system that adds the navigational arrow to add a timer on the left hand controller for the player to keep track of time.





Later in development we adjusted the size of the safe zones to make them less obnoscious and more user friendly, the following picture is from the final product.



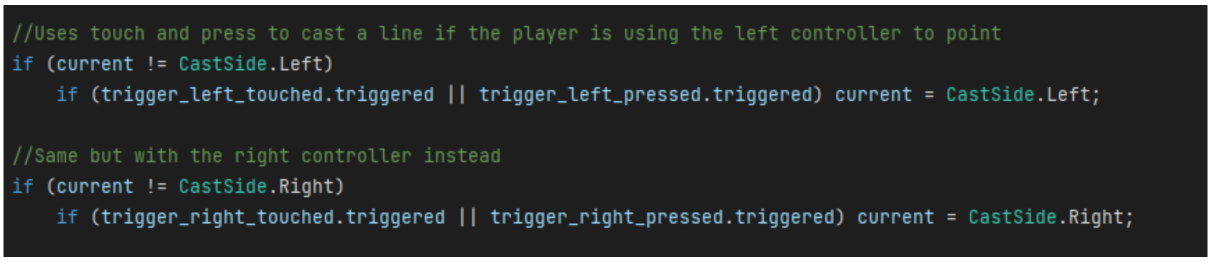
<u>Menus and Software Loop</u>:
To wrap up the second milestone we worked towards having a functioning software loop and interactive menus that the player can comfortably use in VR.

A software loop is the ability to cycle through all the scenes within the application, when doing so each scene will revert to its default parameters when loaded, this prevents from having to manually clear the data each time a scene is loaded/unloaded.

Before creating a system to tie all the different scenes together, we created all the necessary missing scenes and started work on menus in each one of these, we used the default Unity canvas for the menus, but it needed tweaking as it does not natively support VR pointing devices such as the Oculus controllers.

We addressed this issue by replacing the default input module with the one officially supported by the headset (XR UI input module); However, this comes with a huge downside: this input module only supports one interact button, which are the back triggers of the controllers, giving us no option to change it.

To then allow controllers to correctly cast a ray to the menus to use as a pointing device, we made a script that checks for user interaction with the trigger buttons, such as resting the fingers on top of them, this prompts the script to enable the ray casting on that controller and disables the other one.

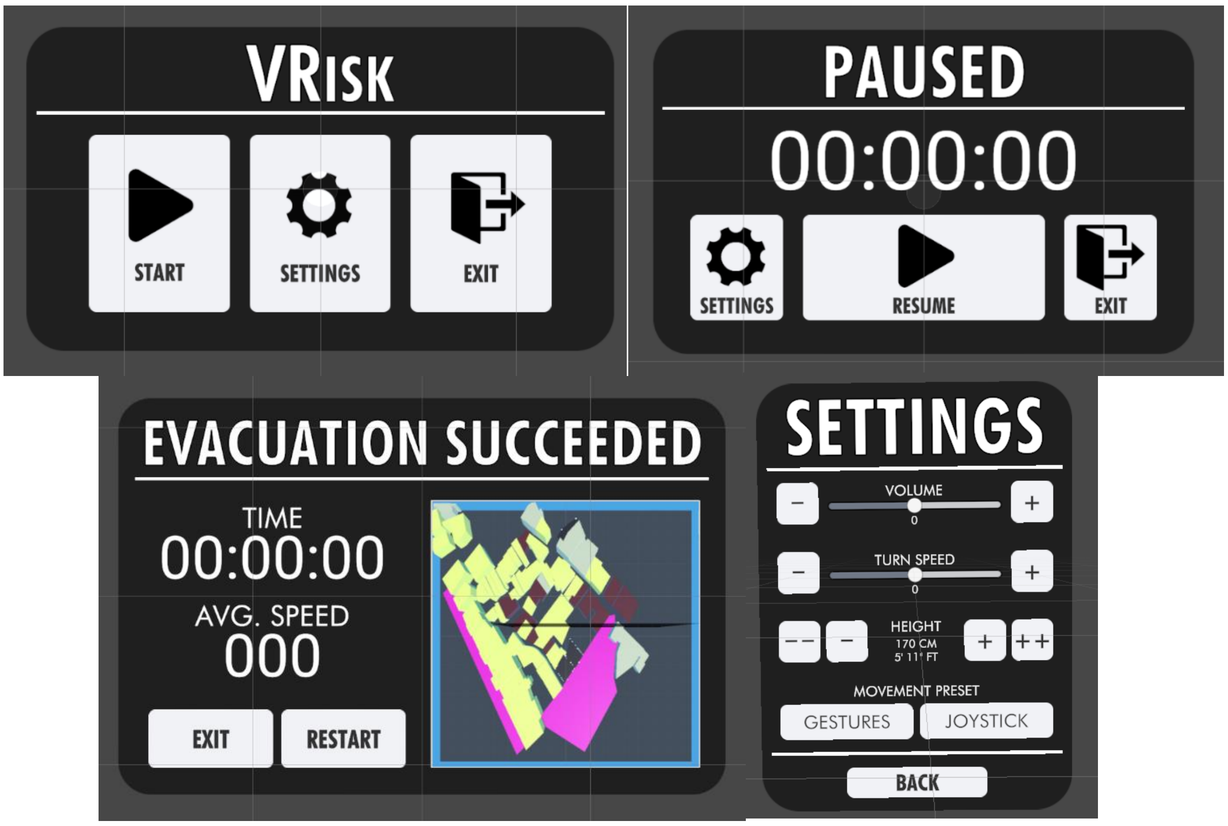


```
//Uses touch and press to cast a line if the player is using the left controller to point
if (current != CastSide.Left)
    if (trigger_left_touched.triggered || trigger_left_pressed.triggered) current = CastSide.Left;

//Same but with the right controller instead
if (current != CastSide.Right)
    if (trigger_right_touched.triggered || trigger_right_pressed.triggered) current = CastSide.Right;
```

With that done, we only miss working menus to be interacting with.

We started creating menus for the application, such as the main menu, pause menu, end menu and a settings menu too, however almost each of those menus had big changes trhoughout development as the state of project evolved, these first iterations had many unimplemted feature that were only planned at the time of their making.

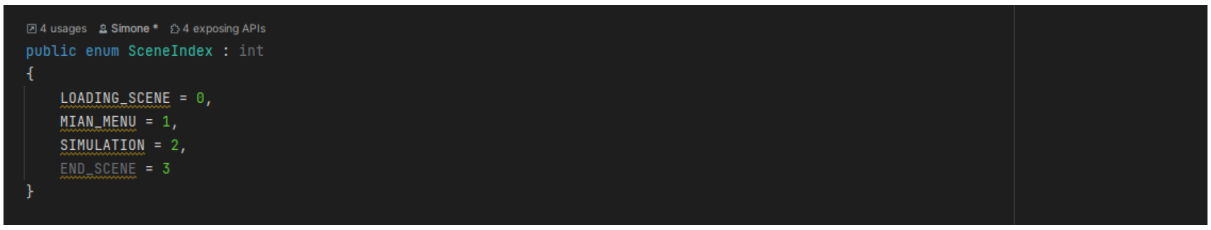
The first iterations of the menus are shown here, in order: Main menu, Pause menu, End menu and the options:



We also linked them to their respective controllers and tweaked their look and feel, the lack of animations caused them to look boring, we downloaded the tool “Lean Tween” and added some smooth animations.

With the menus in place in each of their corresponding scene, we starting to adress how to transition seamingly between scenes, the first thing we had to take care off is how to keep data stored between scenes since when loading a new one the previous gets unloaded and loses all the data that is being stored in it, we fixed this by creating a Unity Scriptable Object named “GameData”, a scriptable object keeps its data in place between scenes.

Once created, we started using it to store the menus and scene data, we also added a “NextScene” variable, this will be at the base of the transition system that we are about to create.

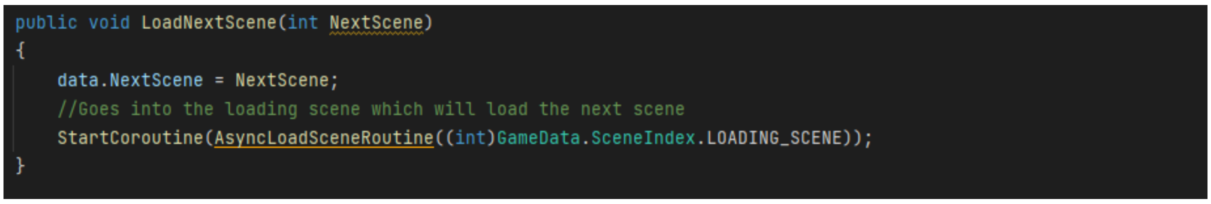


```
4 usages  Simone *  4 exposing APIs
public enum SceneIndex : int
{
    LOADING_SCENE = 0,
    MIAN_MENU = 1,
    SIMULATION = 2,
    END_SCENE = 3
}
```

We started working on the transition system, this system will be made of a loading scene and a transition manager.

The loading scene is a separate scene that we created to specifically load the next scene, it is empty exept the logo of VRisk, this is because when loading a complicated scene, such as the Game Scene, it could cause the application to freeze or lag, which would be very unpleasant in a visually cluttered enviroment. Once the loading scene is loaded, it will read the "Next Scene" value and load that specific scene.

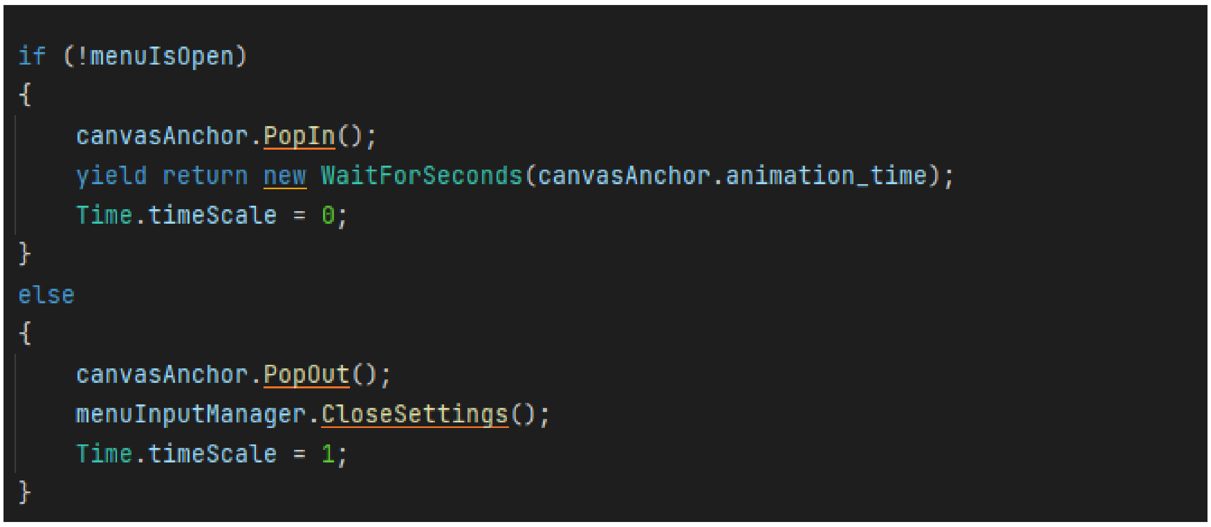
The transition manager is a prefab that can be placed on each scene, it has as script to fade the screen to white, then set the desired "Next Scene" value and load the loading scene.



```
public void LoadNextScene(int NextScene)
{
    data.NextScene = NextScene;
    //Goes into the loading scene which will load the next scene
    StartCoroutine(AsyncLoadSceneRoutine((int)GameData.SceneIndex.LOADING_SCENE));
}
```

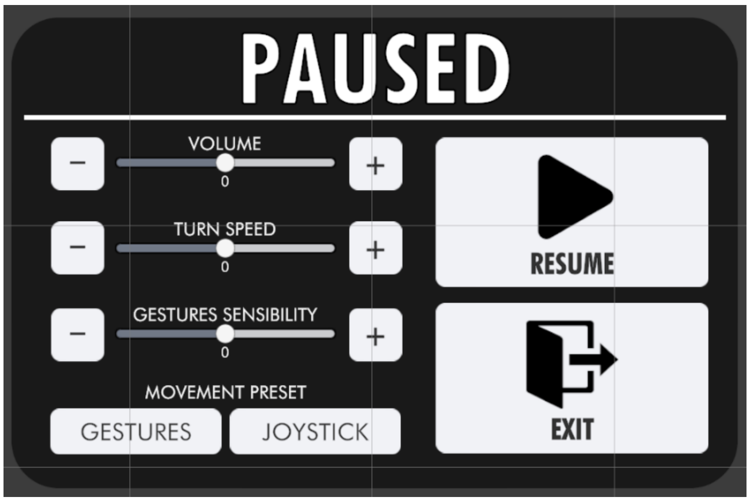
The last thing we worked on for the menus was to integrate the pause feature when opening the pause menu, this was done easily as Unity has a built-in option to do so.

This works by setting the "Time.timeScale" of the project to 0 when pausing the game, this stops the components from updating, freezing everything in place; When resuming we set the value back to 1.



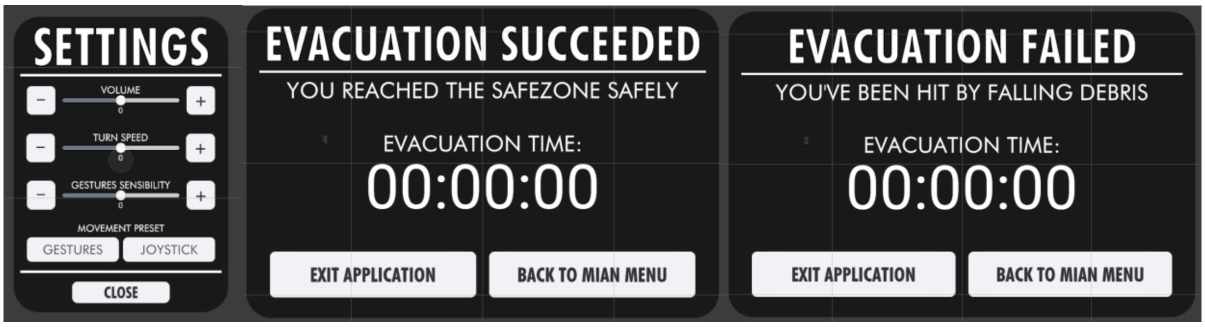
```
if (!menuIsOpen)
{
    canvasAnchor.PopIn();
    yield return new WaitForSeconds(canvasAnchor.animation_time);
    Time.timeScale = 0;
}
else
{
    canvasAnchor.PopOut();
    menuInputManager.CloseSettings();
    Time.timeScale = 1;
}
```

However, pausing the application when opening the pause menu would also cause the menu's animation and the settings pop-up window to not show correctly anymore, we addressed this creating a new version of the pause menu with all the needed information of the same page.



As mentioned, throughout development the only menu that has remained unchanged is the Main Menu, the settings and end menu had to be changed too to implement functionality that has made its way in and remove old uninplemented one.

The settings menu included new options such as gesture sensibility, and the End menu now has two states, one for when the player survived the simulation and another one if not:



SETTINGS

EVACUATION SUCCEEDED

YOU REACHED THE SAFEZONE SAFELY

EVACUATION TIME:

00:00:00

EVACUATION FAILED

YOU'VE BEEN HIT BY FALLING DEBRIS

EVACUATION TIME:

00:00:00

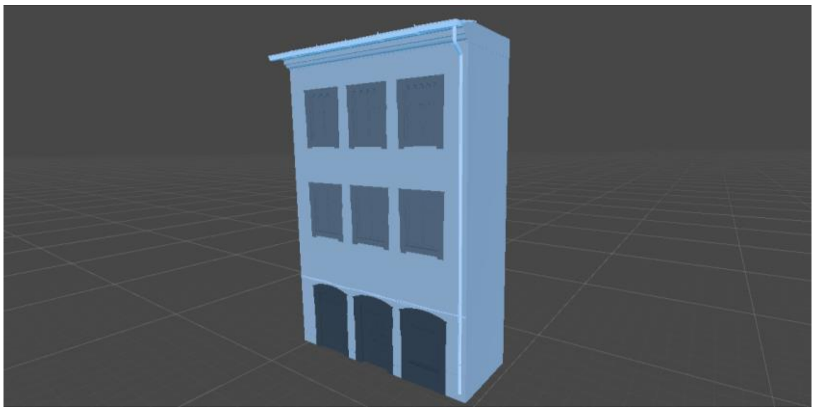
Third Milestone

The completion of the second milestone meant that the project has reached a state where most of the functionality related to the simulation is now in place and playable, with this in mind we focussed the third milestone on enhancing the experience accessibility with a tutorial and visually with models, then worked on how to record data from the player's actions.

One of the main goals for data logging that was to have a way to replay the actions from a player's run in an external software outside the headset.

The Debris and Building models were realized by our 3D artist, trying to mimic the style of buildings present in Coimbra.

__**Modelling**__:
After receiving the models from our 3D artist, we imported them into Unity to check if the model was loading correctly, and unfortunately the first models provided would prevent colours from loading correctly.



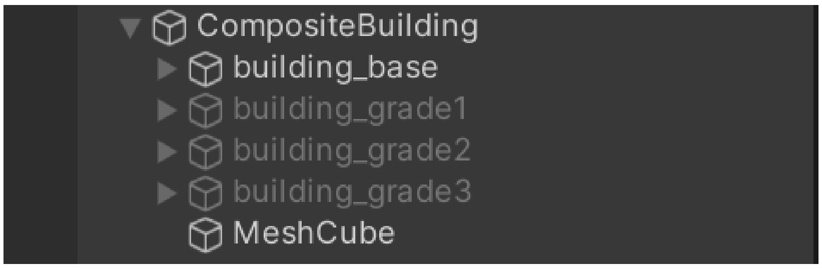
Fortunately, our 3D modeler pushed a fix out in no time, and we were able to keep working, the new iteration of the models fixed the colours but at the expense of the complexity of the model, which requires more resources to be rendered.

Upon successfully integrating the models, we focussed on how to develop a script that would enable the buildings to transition between different states at runtime.

Knowing that each damage state corresponds to a distinct model, we needed a system that would swap models on the fly during runtime, however, this approach posed a challenge: the dynamic loading and unloading of models could potentially tank performance, which is already been a worry considering the complexity of the models.

To work around this issue, we once again employed the concept of pooling, we designed a Game Object called "composite building" which encapsulates all possible damage state models simultaneously and has them always loaded.

“Composite Building” is the name we’ve given a fully modelled building with added functionality.

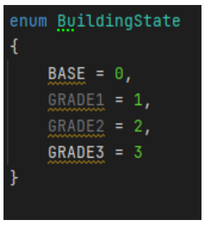


```
▼ CompositeBuilding
    ▶ building_base
    ▶ building_grade1
    ▶ building_grade2
    ▶ building_grade3
      MeshCube
```

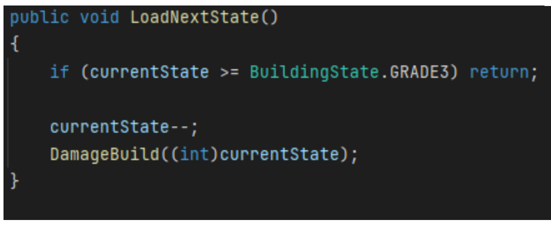
Alongside this, we created a script capable of deactivating and activating each of the models inside, to simulate a specific damage state.

In Unity deactivating a model allows it to remain loaded in the scene without having any major impact on performance, as its renderer and scripts become inactive.

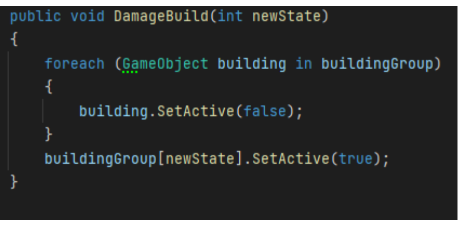
We then updated the script so that each of the damage state selectable is an Enum entry, this allows external scripts to interface with each building and request to activate a specific damage state, or in case of the timeline manager, we made a function that once prompted will apply one grade of damage to the building.



```
enum BuildingState
{
    BASE = 0,
    GRADE1 = 1,
    GRADE2 = 2,
    GRADE3 = 3
}
```

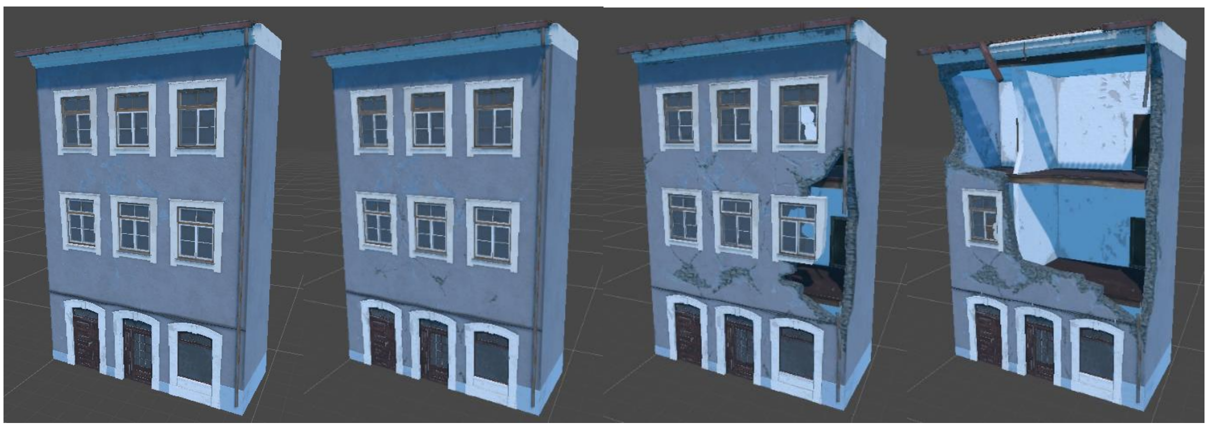


```
public void LoadNextState()
{
    if (currentState >= BuildingState.GRADE3) return;

    currentState--;
    DamageBuild((int)currentState);
}
```



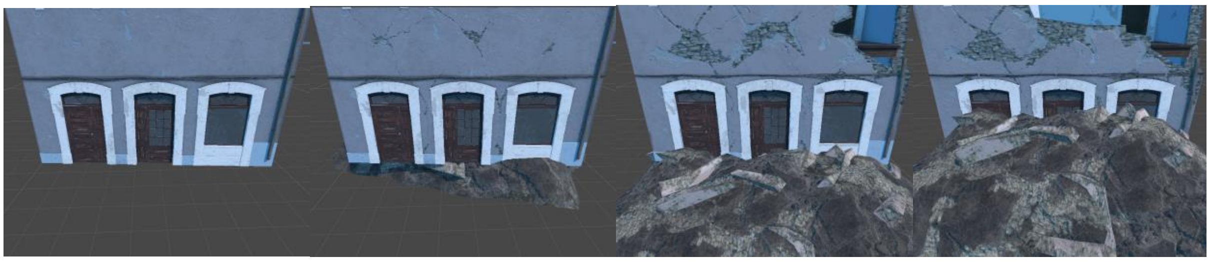
```
public void DamageBuild(int newState)
{
    foreach (GameObject building in buildingGroup)
    {
        building.SetActive(false);
    }
    buildingGroup[newState].SetActive(true);
}
```

Each of the promptable damage state looks like the following:

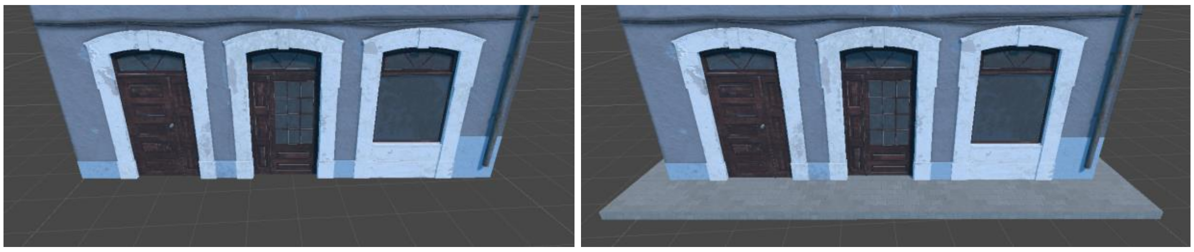


This provides instant transition between damage states while maintaining optimal performance.

We applied the same pooling method to sedimentary debris, each of the damage state has its own model in front of the building, activating/deactivating any state will cause by inheritance his debris to be activated/deactivated too.



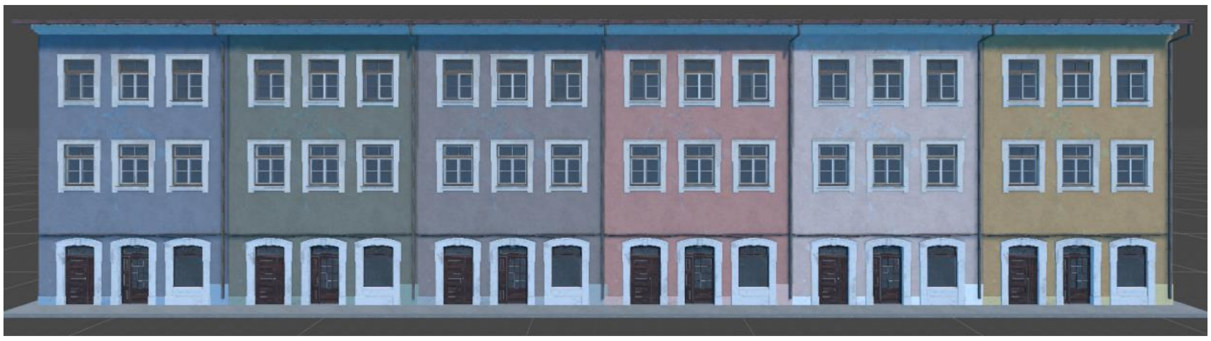
We also made a function to allow a pavement to be enabled too, however we did not use this in the simulation but turned out to be a nice addition in the tutorial.



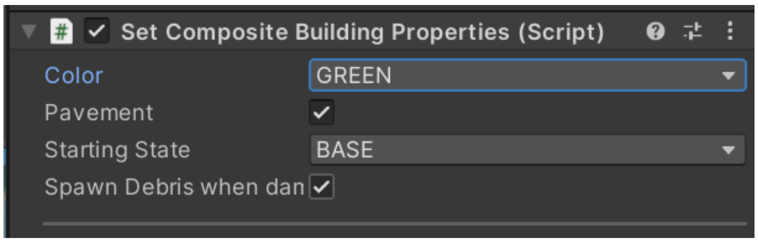
To wrap up the Composite Building prefab, we worked on improving its look by adding the option to select multiple colours, the system works by changing the albedo of the material of the building, this allows for the material to keep its properties but to change its colour.

We used street view in some of Portugal most famous cities and sampled a set of colours from the street view pictures themselves, the colours we've picked are based on how frequently we've seen that colour repeated across the city's buildings and how well it would fit in the simulation.

The colours we have picket are the following:



Lastly for ease of use, we made a quick editor script that allows to change a building's properties easily from the editor.

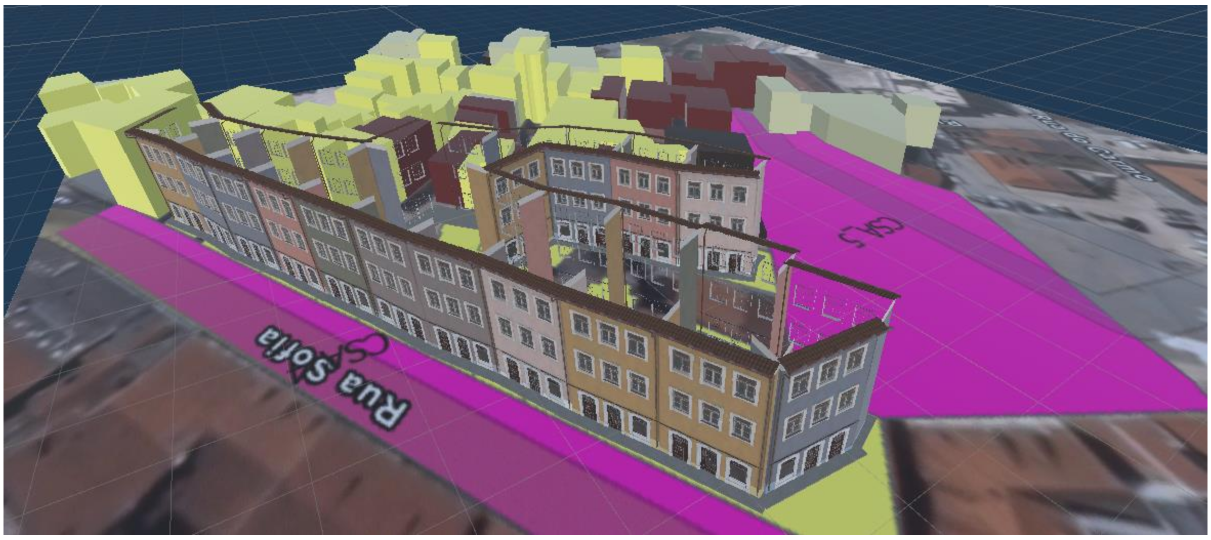


Set Composite Building Properties (Script)

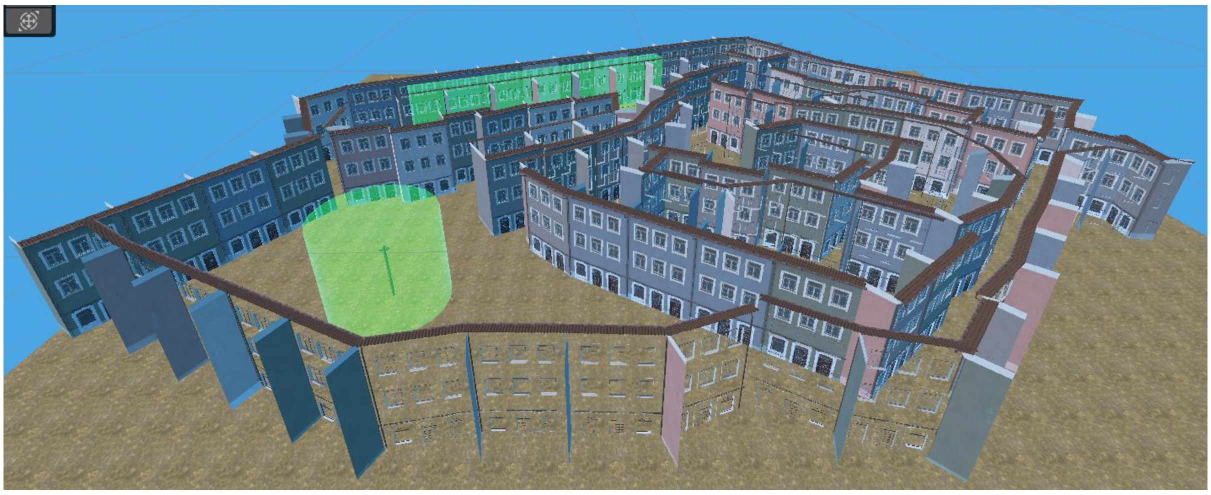
Color	GREEN
Pavement	✓
Starting State	BASE
Spawn Debris when dan	✓

With the composite buildings now ready to go, we finally started to rework the map to use the new models.

We referred to the supplied risk-map to accurately populate the map with new buildings, maintaining the city layout and proportions as closely as possible. However, adjustments were necessary due to the fixed dimensions of the models; This is how it looked like during the modelling process:



Once finished, this was the result:

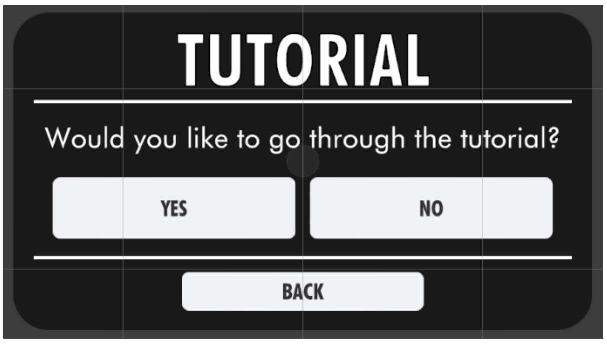


Unfortunately, later on we had to compromise on the size and the level of detail during the optimization phase (Covered below).

<u>Tutorial</u>:
VRisk is an application meant to be playable by a wide audience, this includes individuals who may not have previous experiences with VR devices and may lack the knowledge on how to interact with this kind of device.

We address this regard by trying and implementing a tutorial to teach the player the basics on how the simulation works and how to move/interact with it.

We started by creating a new window in the main menu that will ask the player if they wish to access the tutorial before starting the simulation, if they chose to do so, the tutorial will load.


TUTORIAL

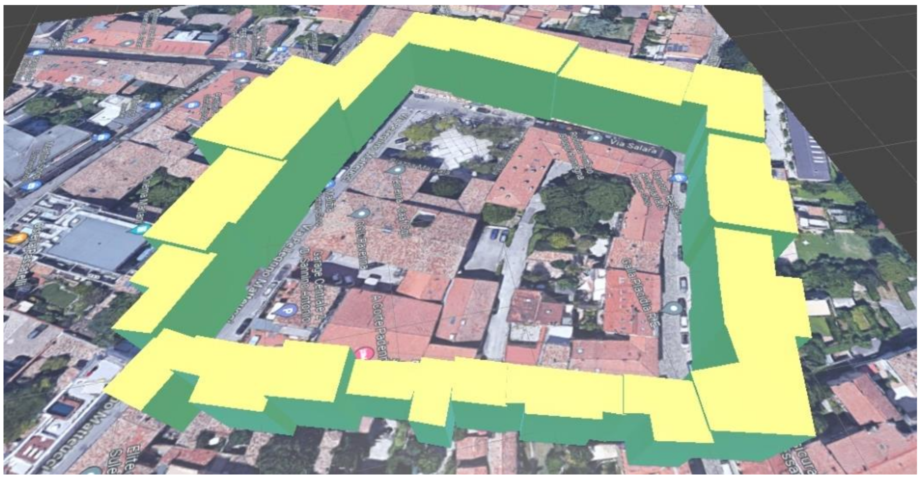
Would you like to go through the tutorial?

YES NO

BACK

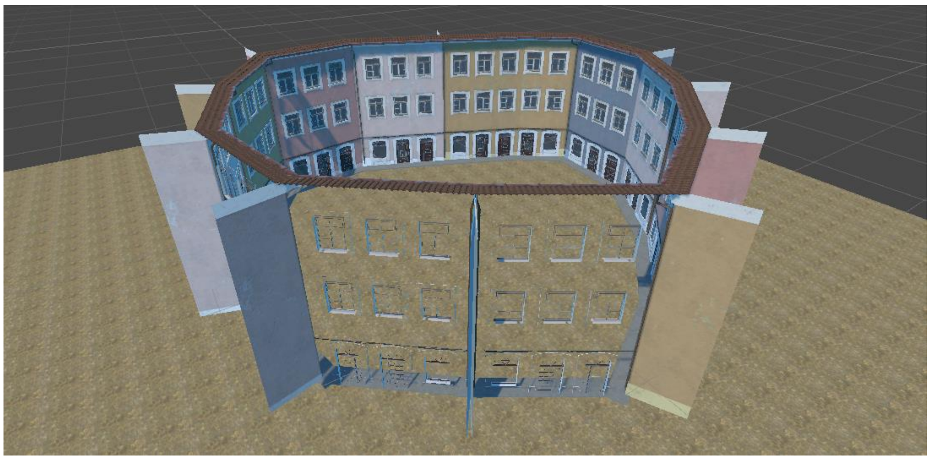
We built the tutorial in the same scene as the main menu, our vision was to have an area representing a small plaza, we did not want anything more complex like hallways or narrow corridors, keeping it simple should help new users to familiarize with the controls before entering the urban area of the simulation.

As an inspiration for the plaza's size and shape, we used a district area in the city centre of Ravenna (Simon's hometown), this district is part of the historic city centre located nearby Saint John's church and has a set of connecting roads which form a square-ish shape, which should work well in the tutorial area, using some satellite images, we blocked it out:



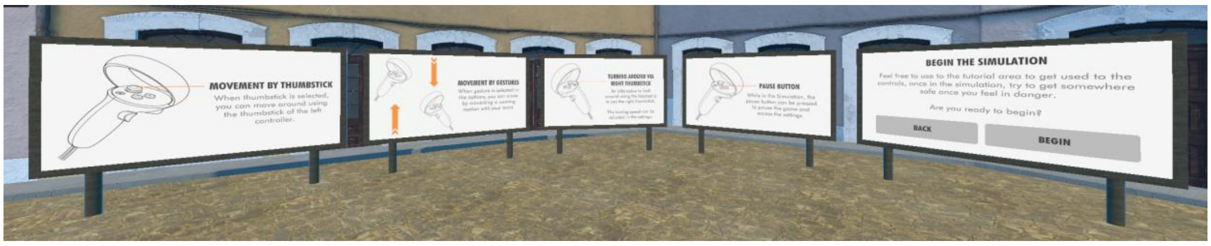
This resulted to be just about right to offer the right amount of space to try the controls and to move around, we also tried to add a loading animation that would make the tutorial appear from beneath the player, but this would cause motion sickness, hence we scrapped it.

As the size was right, we modelled the tutorial, and once again unfortunately we had to alter the shape to accommodate the fixed size of the buildings.



The tutorial so far is accessible and navigable, but it still does not provide any kind of information to the player about the simulation.

We approached this firstly by setting up a set of billboards in the area that would have both images and text to teach the player the basics of the controls, this involved movement by thumbstick, gestures, looking around pausing and tweaking settings.



MOVEMENT BY THUMBSTICK

When thumbstick is selected, you can move around using the thumbstick of the left controller.

MOVEMENT BY GESTURES

PAUSE BUTTON

BEGIN THE SIMULATION

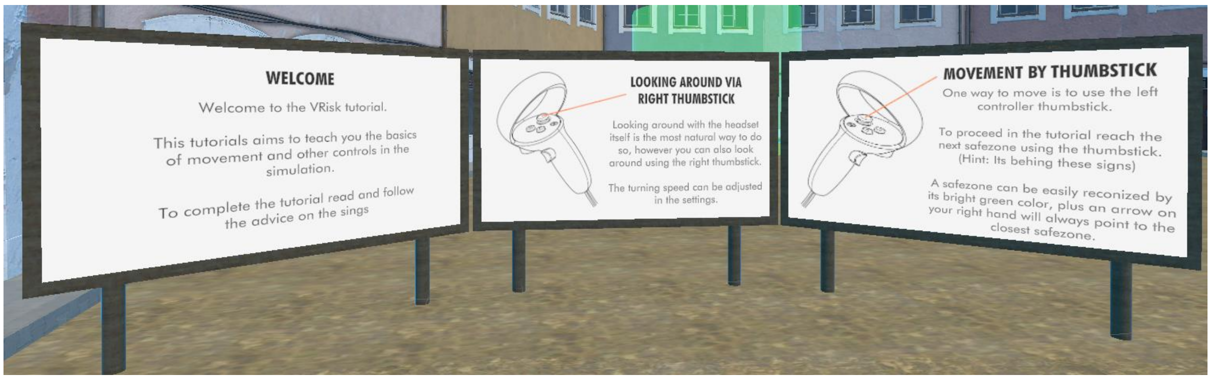
Feel free to use to the tutorial area to get used to the controls, once in the simulation, try to get somewhere safe once you feel in danger.

Are you ready to begin?

However, after some feedback we realized that this approach wasn't effective.

This iteration of the tutoriual would not push the player in exploring or trying the different movement methods, and most importantly it would not teach how to locate and interact with safezones.

We decided to keep the use of billboards but to rework the tutorial entirely, now the player will be placed in front of a set of billboards that would teach them how to locate and interact with safezones, alongside teaching how to look around and moving around with the thumbstick.



WELCOME

Welcome to the VRisk tutorial.

This tutorials aims to teach you the basics of movement and other controls in the simulation.

To complete the tutorial read and follow the advice on the sings

LOOKING AROUND VIA RIGHT THUMBSTICK

Looking around with the headset itself is the most natural way to do so, however you can also look around using the right thumbstick.

The turning speed can be adjusted in the settings.

MOVEMENT BY THUMBSTICK

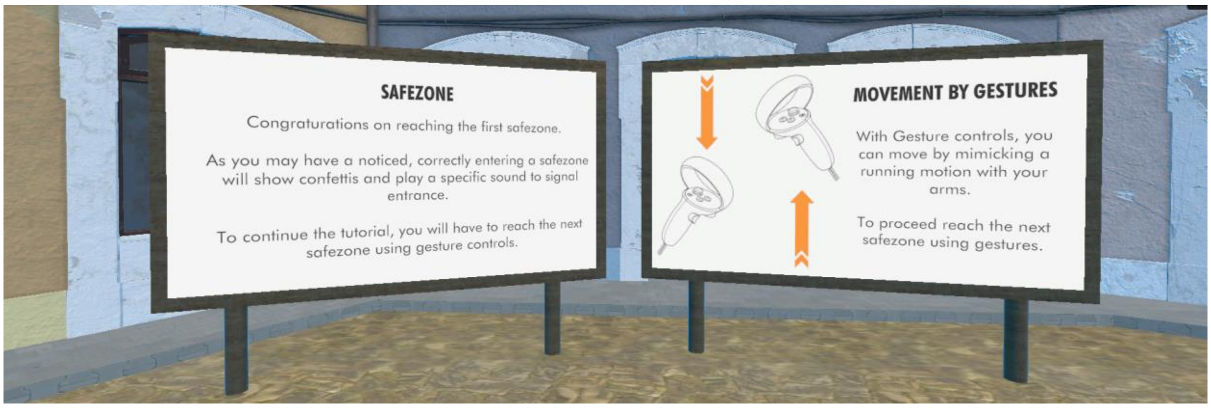
One way to move is to use the left controller thumbstick.

To proceed in the tutorial reach the next safezone using the thumbstick. (Hint: Its behing these signs)

A safezone can be easily reconized by its bright green color, plus an arrow on your right hand will always point to the closest safezone.

To proceed in the tutorial, the player has to move and navigate to the safezone using the thumbstick, as the gesture controls will be disabled.

After reaching the first tutorial safezone, a new set of billboard will appear that will teach the player how to move using gestures and will also lock the player movement method to this kind of movement, the player will then have to reach and interact with the next safezone to continue.



SAFEZONE

Congraturations on reaching the first safezone.

As you may have a noticed, correctly entering a safezone will show confettis and play a specific sound to signal entrance.

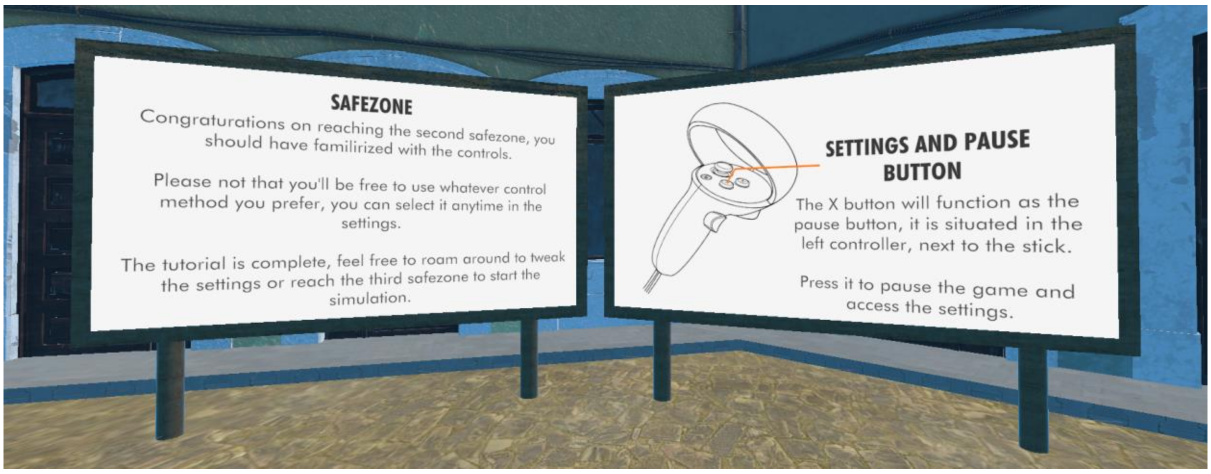
To continue the tutorial, you will have to reach the next safezone using gesture controls.

MOVEMENT BY GESTURES

With Gesture controls, you can move by mimicking a running motion with your arms.

To proceed reach the next safezone using gestures.

After reaching the second tutorial safezone, another set of billboards will appear that will teach the player how to pause and use the settings menu, it will also explain to the player that their movement method can be changed any time within these settings.


SAFEZONE

Congraturations on reaching the second safezone, you should have familirized with the controls.

Please not that you'll be free to use whatever control method you prefer, you can select it anytime in the settings.

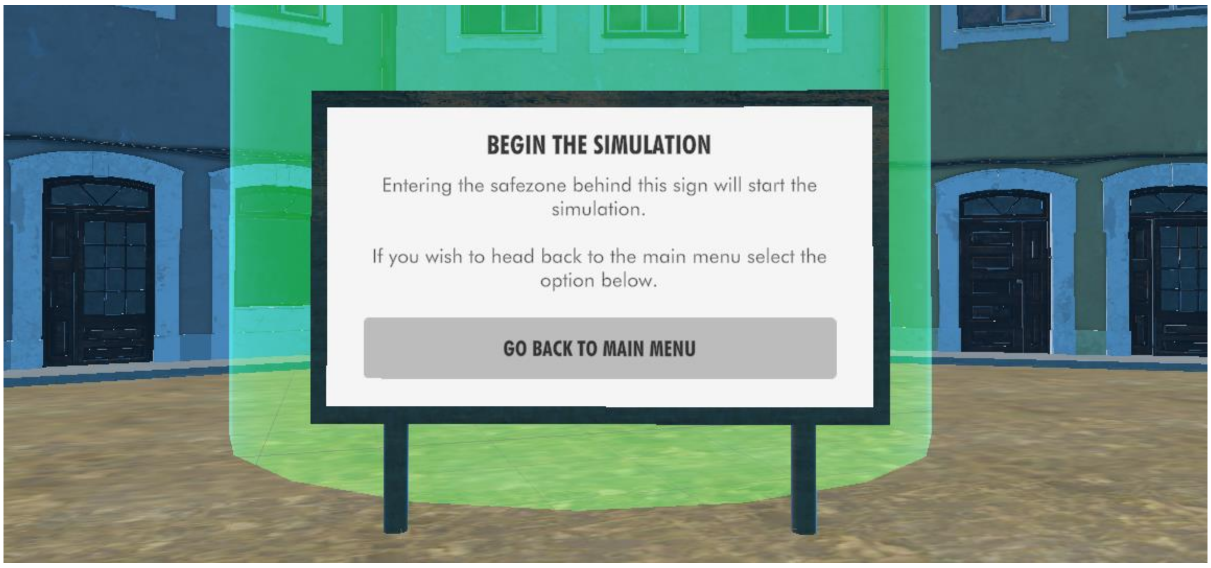
The tutorial is complete, feel free to roam around to tweak the settings or reach the third safezone to start the simulation.

SETTINGS AND PAUSE BUTTON

The X button will function as the pause button, it is situated in the left controller, next to the stick.

Press it to pause the game and access the settings.

A new safezone will spawn shortly after, and the player to proceed will have to reach it, once inside it will start the simulation, while the billboard outside allows the to return to the menu.

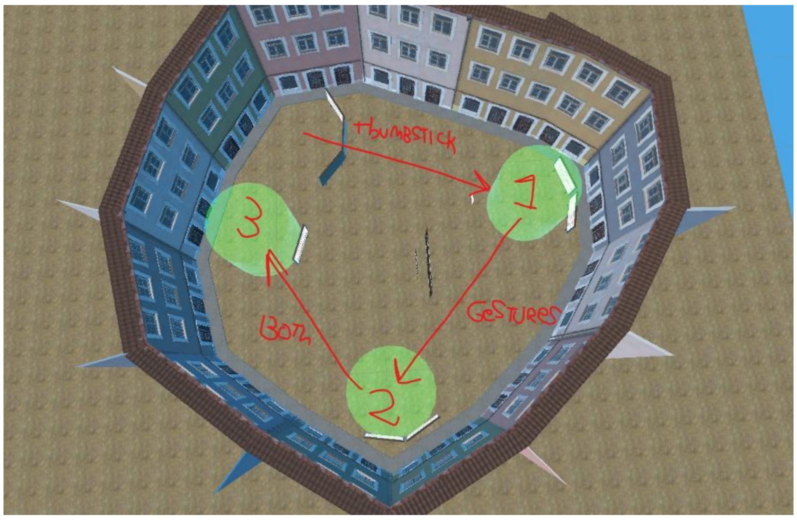

BEGIN THE SIMULATION

Entering the safezone behind this sign will start the simulation.

If you wish to head back to the main menu select the option below.

GO BACK TO MAIN MENU

This approach while still far from perfect, Is vastly more effective than the previous one, as now the player will have to interact with safezones and try the different control methods to proceed.



User Registration and Data logging:

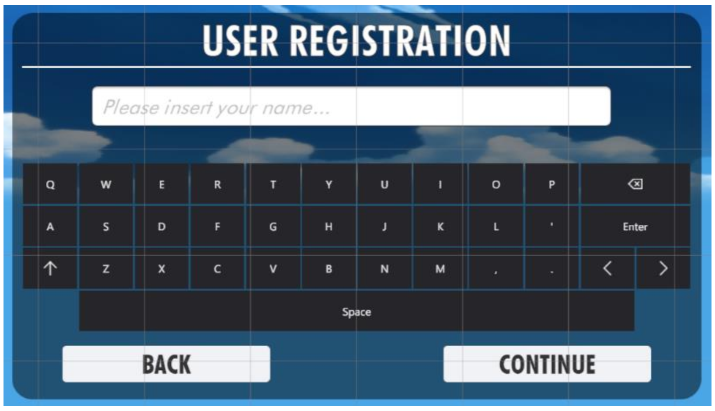
We are now approaching the completion of VRisk, with the tutorial wrapped up the simulation now has complete functionality and is fully playable, however, there remains an essential aspect that we have yet to integrate: the functionality to log data from player playthroughs.

This is crucial to conduct risk assessment; our goal is to implement a system that will be able to track the player's position and rotation in real-time and display this data in a replay viewer.

The replay viewer should then feature the simulation's full map and a time slider, allowing users to review and analyse the player's position and orientation at any given moment in the timeline.

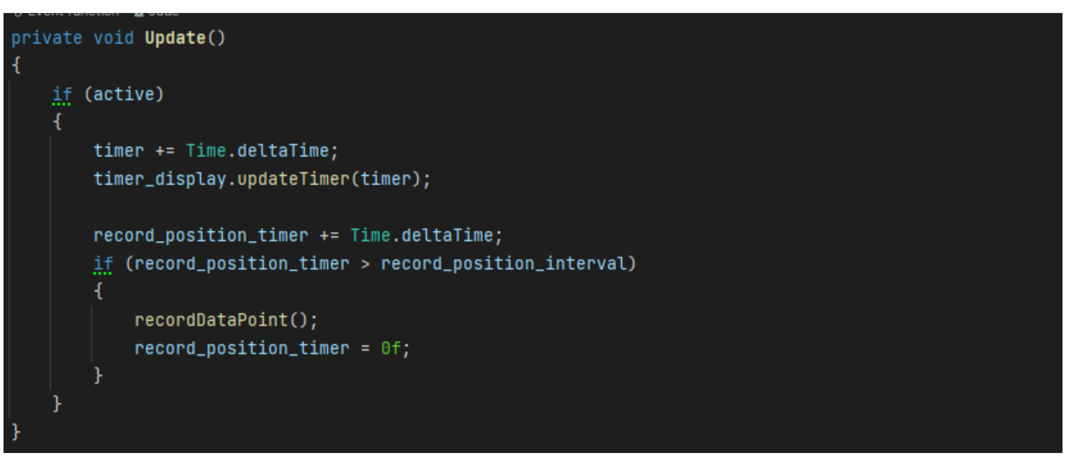
However, the samples of data collected during these sessions require specific naming for effective recognition and analysis, this is because participants will play the simulation and then be asked to complete a survey, it is essential for the names on the surveys to correspond with the data taken. To facilitate this, we have implemented a user registration tab that prompts players to enter their name before starting the simulation, this name will then display in the replay viewer when loading the data in.

We implemented this by adding a new window in the main menu containing a virtual keyboard, this will pop up after the player pressed "Start" and will ask for their name.



Before being used as the name of the file, it will be stored inside a variable store in GameData, cleared each time the user will go back to the main menu.

To track the player's movement and rotation, we created a script called "DataTracker", it will start automatically when the player enters the simulation; By using a reference to it, the script will sample the position and rotation of the player at every desired interval, determined by a time variable.

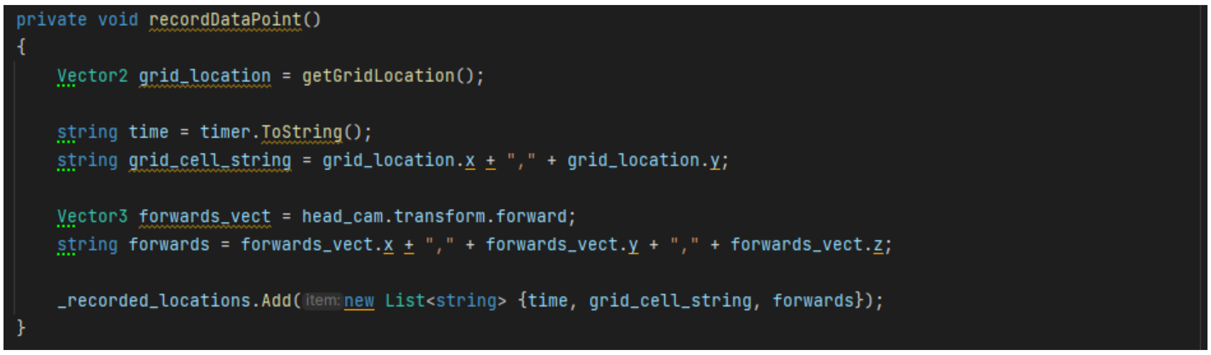


```
private void Update()
{
    if (active)
    {
        timer += Time.deltaTime;
        timer_display.updateTimer(timer);

        record_position_timer += Time.deltaTime;
        if (record_position_timer > record_position_interval)
        {
            recordDataPoint();
            record_position_timer = 0f;
        }
    }
}
```

Each of the samples recorded will contain:

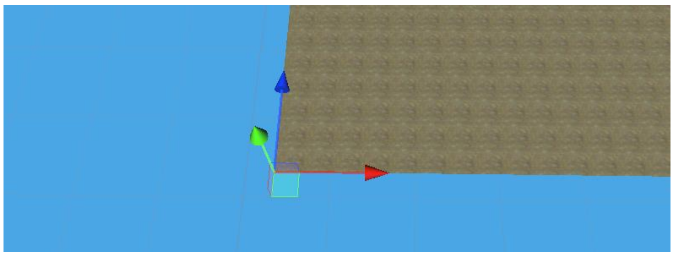
- The time when that sample has been taken.
- Grid X and Y coordinates to represent the position of the player in a top-down view.
- Camera vector to represent the rotation of the player.

Each sample taken is saved inside a list of recorded samples, called "_recorded_locations", but not saved to file yet.



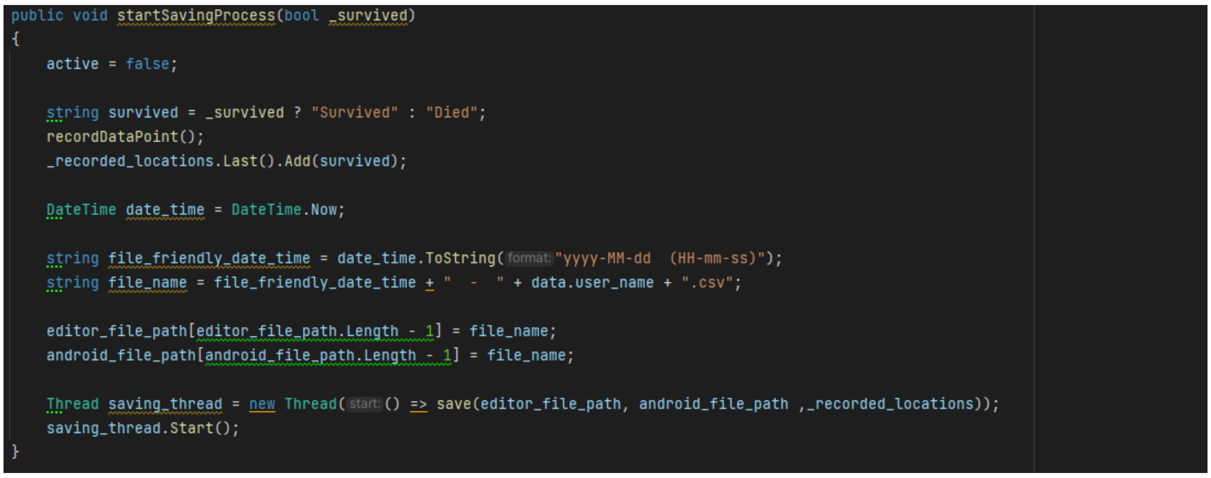
```
private void recordDataPoint()
{
    Vector2 grid_location = getGridLocation();

    string time = timer.ToString();
    string grid_cell_string = grid_location.x + "," + grid_location.y;

    Vector3 forwards_vect = head_cam.transform.forward;
    string forwards = forwards_vect.x + "," + forwards_vect.y + "," + forwards_vect.z;

    _recorded_locations.Add( item: new List<string> {time, grid_cell_string, forwards});
}
```

To make the grid position of the player as precise as possible, the location of the player is taken in relation to the map origin, which is determined by an empty game object in the bottom left corner.



Sampling will continue until the player reaches a safe zone or gets hit by debris, when that happens the script will stop the sampling of data then start the script that will save this to file, this script is called "startSavingProcess".

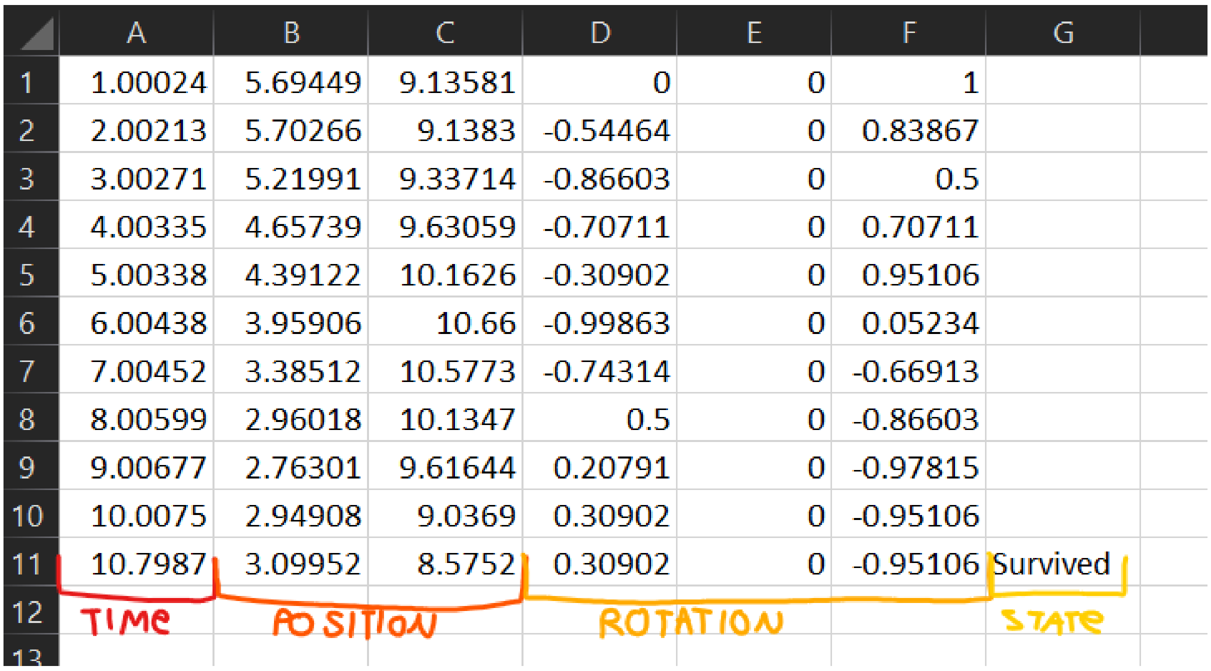
When called, this script will save the state of the player with either survived (if in a safezone) or died (if hit by derbis) in the last sample recorded, next it will start the saving process by moving all the samples inside a .CSV file in the chosen directory to then rename the file with the player's name and the date and time of the playtrhough.



```
public void startSavingProcess(bool _survived)
{
    active = false;

    string survived = _survived ? "Survived" : "Died";
    recordDataPoint();
    _recorded_locations.Last().Add(survived);

    DateTime date_time = DateTime.Now;

    string file_friendly_date_time = date_time.ToString( format: "yyyy-MM-dd  (HH-mm-ss)");
    string file_name = file_friendly_date_time + "  -  " + data.user_name + ".csv";

    editor_file_path[editor_file_path.Length - 1] = file_name;
    android_file_path[android_file_path.Length - 1] = file_name;

    Thread saving_thread = new Thread( start: () => save(editor_file_path, android_file_path ,_recorded_locations));
    saving_thread.Start();
}
```

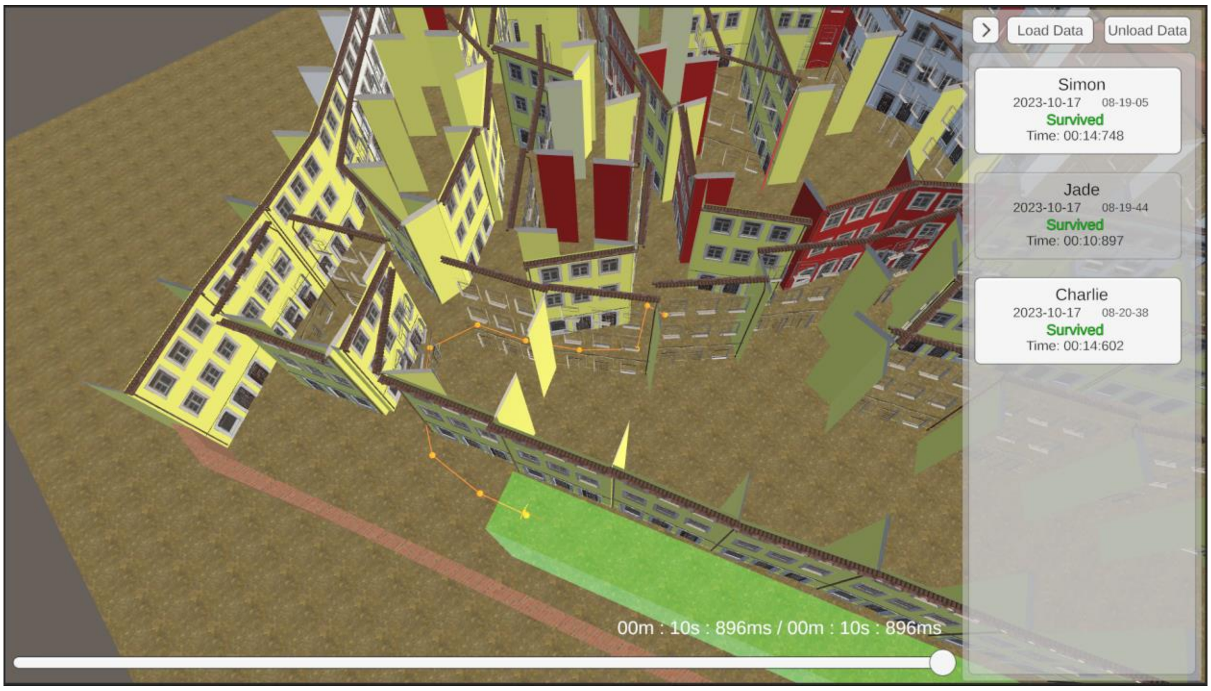
The total time of completion will not be explicitely saved but deducted form then last taken sample, as it will happen when entering a safezone or getting hit.

For a look behind the scenes, that’s how replay files look like:



	A	B	C	D	E	F	G
1	1.00024	5.69449	9.13581	0	0	1	
2	2.00213	5.70266	9.1383	-0.54464	0	0.83867	
3	3.00271	5.21991	9.33714	-0.86603	0	0.5	
4	4.00335	4.65739	9.63059	-0.70711	0	0.70711	
5	5.00338	4.39122	10.1626	-0.30902	0	0.95106	
6	6.00438	3.95906	10.66	-0.99863	0	0.05234	
7	7.00452	3.38512	10.5773	-0.74314	0	-0.66913	
8	8.00599	2.96018	10.1347	0.5	0	-0.86603	
9	9.00677	2.76301	9.61644	0.20791	0	-0.97815	
10	10.0075	2.94908	9.0369	0.30902	0	-0.95106	
11	10.7987	3.09952	8.5752	0.30902	0	-0.95106	Survived
12							
13							

When loaded into the Scrub Through, the application we made to view replays, it will read the samples and recreate an accurate path of the player’s actions.



Simon
2023-10-17 08-19-05
Survived
Time: 00:14:748

Jade
2023-10-17 08-19-44
Survived
Time: 00:10:897

Charlie
2023-10-17 08-20-38
Survived
Time: 00:14:602

00m : 10s : 896ms / 00m : 10s : 896ms

This wraps up the third milestone and last milestone.

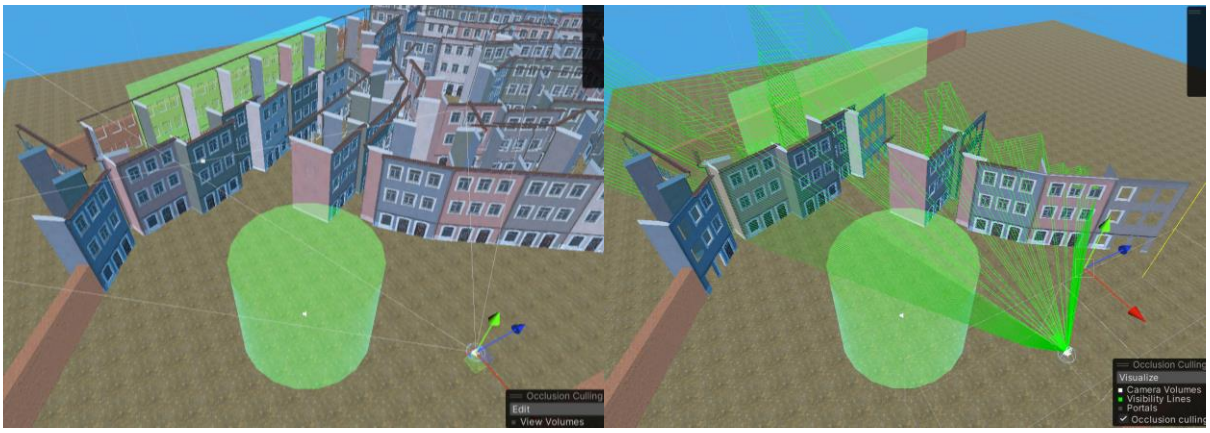
Optimization

One last thing to do was to optimize the project for the headset, this took un quite some time given the hardware of the Oculus Quest 2, but we were able to achieve a playable experience.

In the worst-case scenario, we had in improvement from 5 frame per second to a solid 25 at the lowest, following are the techniques that we used to achieve said performance improvement.

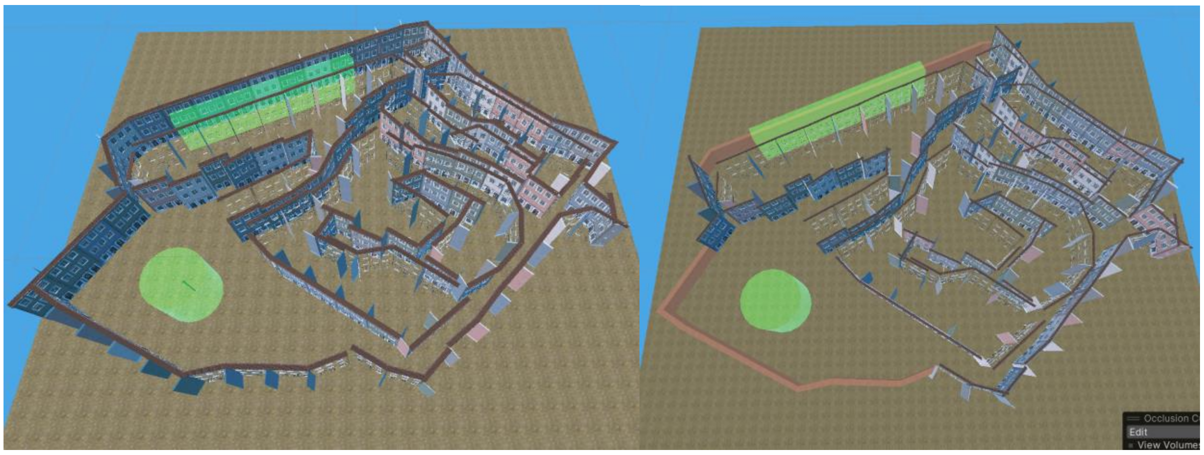
Occlusion Culling:
Occlusion culling is a technique used in 3D graphics rendering to improve performance. It involves not rendering objects that are blocked by other objects and therefore not visible to the camera. This reduces the computational load, as only visible elements are processed.

Considering the complexity and number of models in the simulation scene, this allowed us to achieve an impressive performance uplift, below are two images showcasing what is rendered during runtime without and with occlusion culling, the white lines are the camera viewport, and the green lines are the occlusion culling rays:



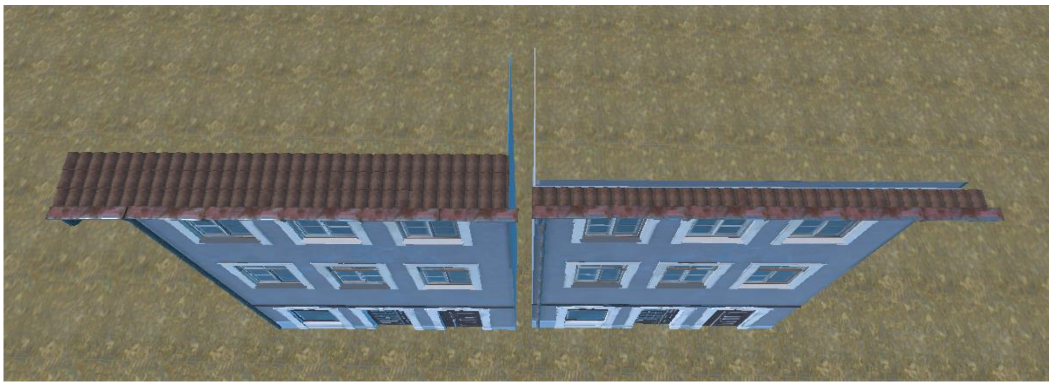
Removal of unnecessary models:
Despite having occlusion culling, big open areas still require the application to render a high number of buildings, we addressed this by removing some parts of the map that were less relevant and replace non-shaking buildings with walls.

Below is the map before and after removal of unnecessary models:



Model Optimization:
The last tweak to the models we made was removing 2/3 of the roof tiles, the more than halved the draw calls in the scene and gave another performance boost while being unnoticeable from the player's point of view.

Below is the model before and after the removal of the extra roof tiles:



Code Optimization:
The following are some of the other changes in the codebase that we made to improve performance:

- Localized shaking: Buildings that are far and hardly visible from the player wont shake.
- Removal of collapse additional shake: A building collapsing would case the buildings nearby to shake harder, this was unnoticeable and therefore removed.
- Simplification of light rendering: Removed shadows and diffuse lighting.
- Added LOD (dynamic level of details) to particles: Particles far away will be less detailed or not rendered.
- Reduced Debris Lifespan: Debris disappear faster.