
**KAUNAS UNIVERSITY OF TECHNOLOGY**

**FACULTY OF INFORMATICS**

# T120B166 Development of Computer Games and Interactive Applications

# Banisher

Group, Name and Surname:
IFF 7-11, Mantas Klimašauskas
Date: 2020-03-02

Kaunas, 2020

Table of Contents

# Table of Images

# Table of code

# Work Distribution Table:

| ***Name/Surname*** | ***Description of game development part*** |
|---|---|
| *Mantas Klimašauskas* | *Programming* |
| *Deividas Verbickas* | *Programming, level design* |

# Description of Your Game

Description of Your Game.

1. 3D or 2D? *3D*
2. What type is your game? *Action.*
3. What genre is your game? *Fighting.*
4. Platform? *PC.*
5. Scenario Description. *A fighting game where the player has the ability to enter a separate mirror dimension.*
   *The dimension is called the Exsilium and the one who can enter it are called outcasts (exsul). In the Exsilium you can see and touch (and even affect) the real world. Time moves slower in the Exsilium relative to the real world, which gives an advantage to anyone who is in it. In order to stay in the Exsilium the outcast must use their Vis (energy). The more matter the outcast takes to the mirror dimension the more Vis is being used. Also, the outcast uses more energy the further away they are from their physical body (in the real world). Vis is also directly related to stamina of the outcast in the real world.*
   *Inside the Exsilium outcasts can use their energy to acquire buffs, such as super strength. Most of the enemies encountered by the player do not possess the power to enter use the Exsilium to their advantage. But there will be bosses, who will be just as capable (or even stronger) than the player. Anyone who is damaged while in the mirror dimension cannot be killed and simply loses their energy. In order to regain the energy, the outcast must use special items.*

# Laboratory work #1

## List of tasks

1. Second level basic layout
2. Basic movement player physics
3. Add map details and lights
4. Active ragdolls, which try to stay upright (standing)

## Solution

## 1. Second level design

Created a small alleyway, that leads to a warehouse. The alleyway is surrounded by a few buildings. The warehouse has three holes in the walls as entrances.

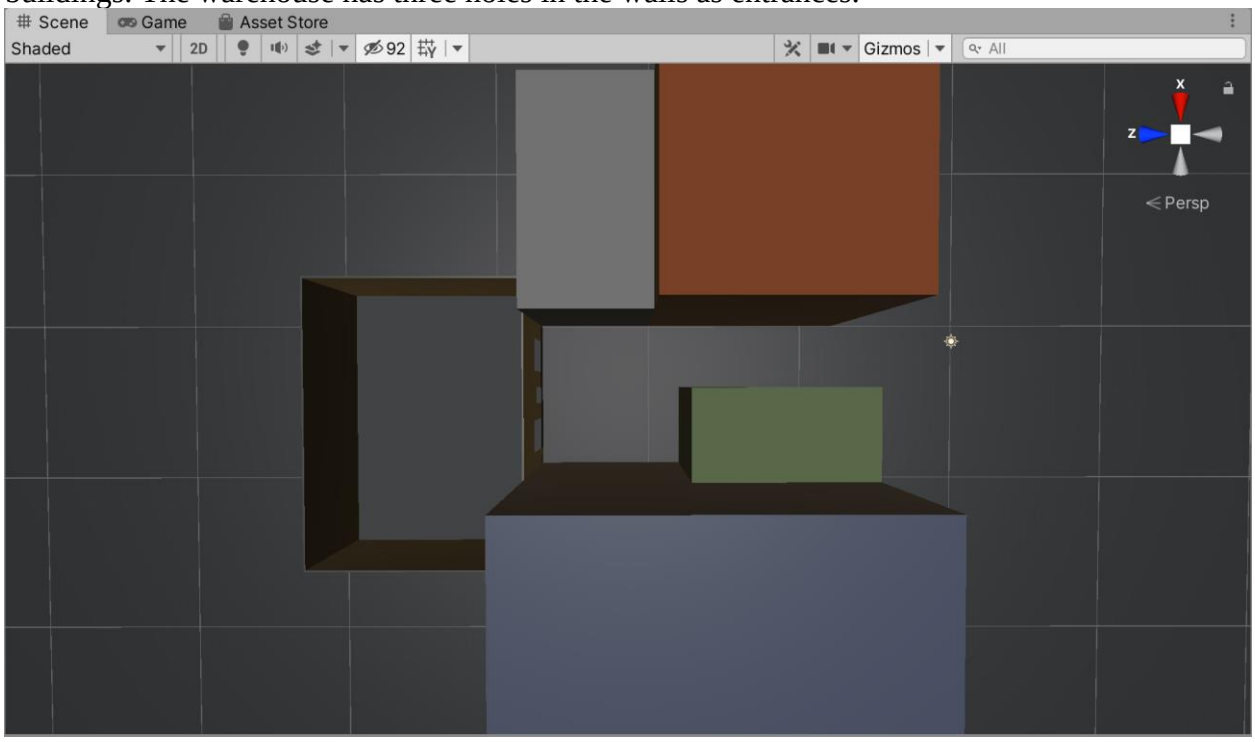



**Figure 1.** Second level layout

## 2. Basic movement player physics

Added a script for the SteamVR Player prefab, to enable basic physical ground detection for the player.

When the player moves over a collider, the script will push (teleport) the player up, to adjust the real-world ground to the object the player character in the game is standing on.

When the player steps off an object, a rigidbody attached to the Player prefab is enabled (is kinematic is turned off) and the player falls, until the ground is close enough.

To detect the ground below the player, a raycast with a limited distance is casted down. If the raycast hits something, that something is considered the ground, and the player's position is adjusted accordingly. If the raycast doesn't hit anything, the attached rigidbody is activated for the player to fall.

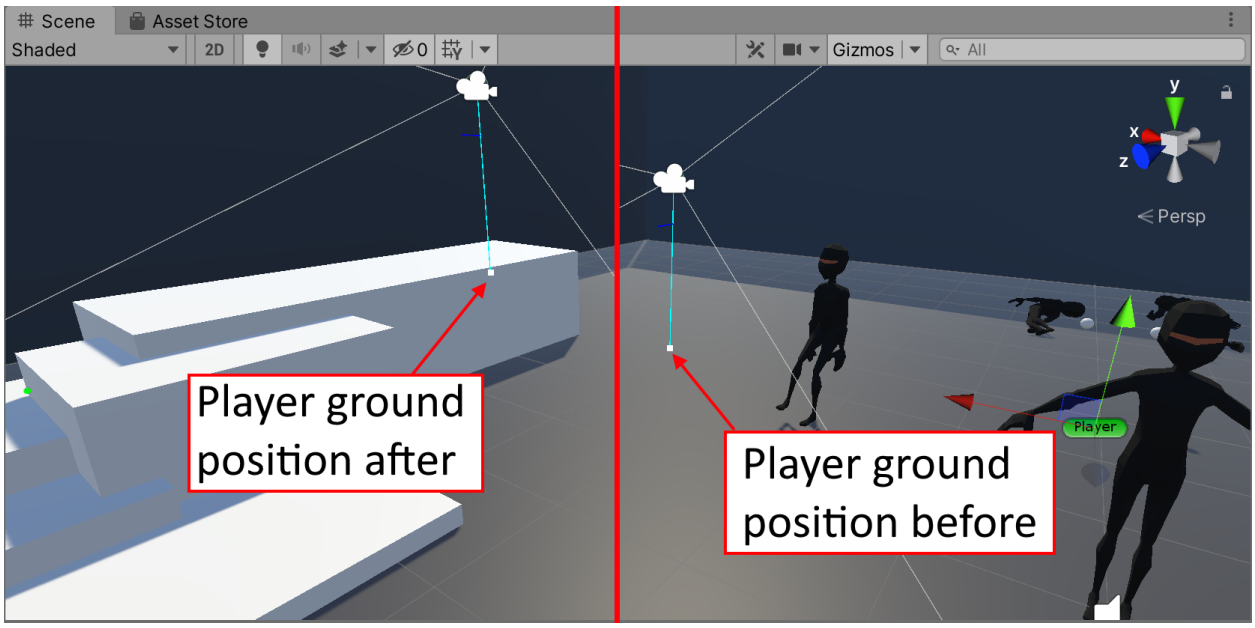



**Figure 2. Player ground position before moving above the white box and after**

## 3. Second level design

Added doors and ramps to some of the buildings. Also, added an indoor second-floor office, with windows. The windows are made with a transparent material. Also added a few pillars inside the warehouse.

For lighting, added five spot lights inside the warehouse, change the direction of the directional light (the sun) to make it an evening setting.

Also made a few prefabs, such as doors, ramps, lights.

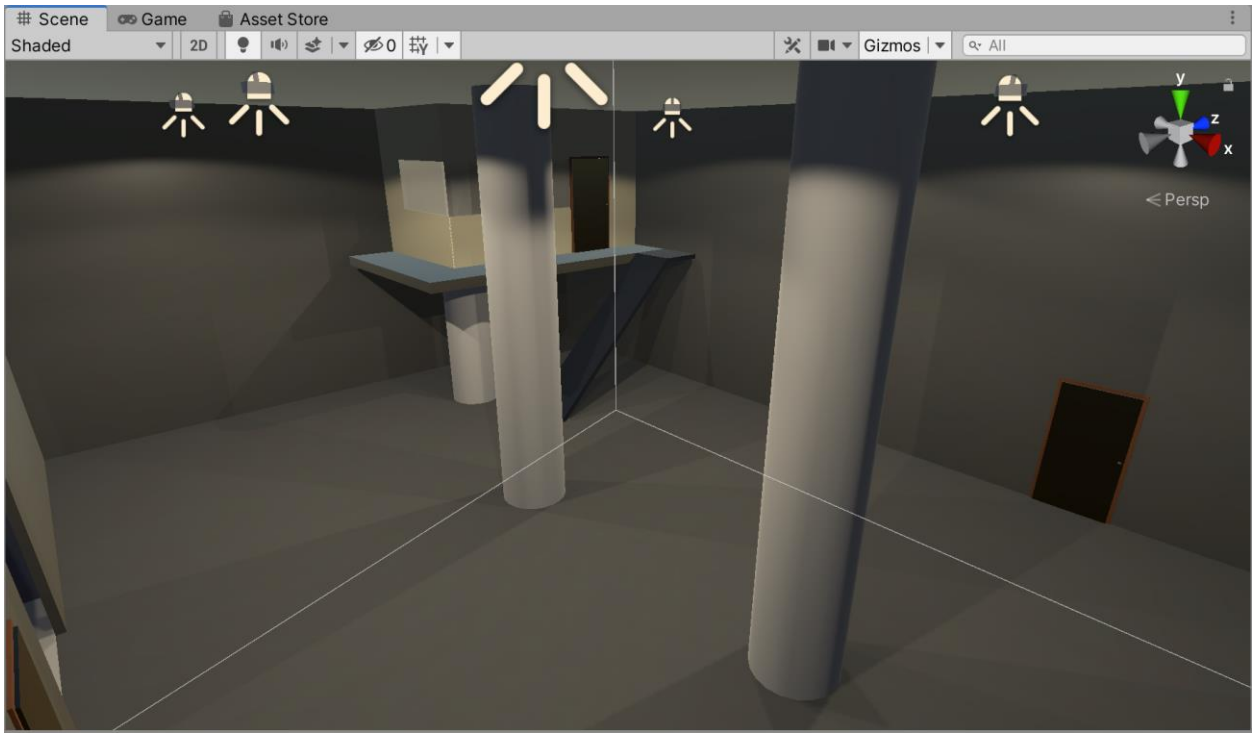



**Figure 3.** Inside the warehouse

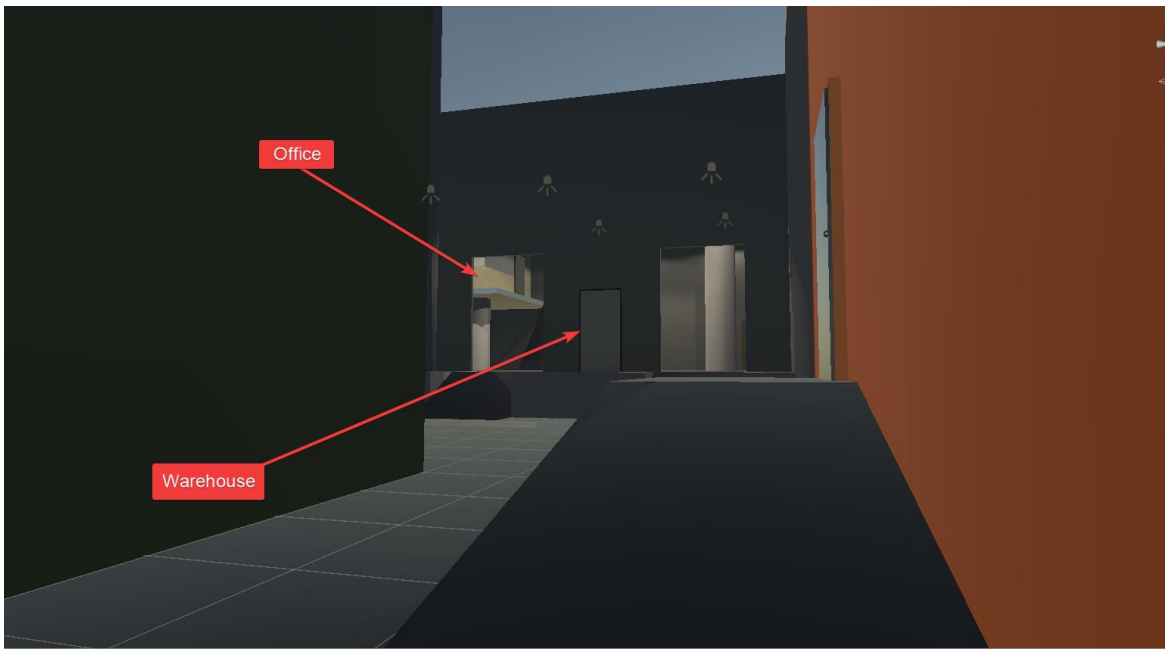



**Figure 4.** Looking into the warehouse through the alleyway

## 4. Active ragdolls, which try to stay upright (standing)

Started the work on active ragdolls (ragdolls, which try to physically move to certain positions), for using them on enemies and the player body.

To create the ragdoll itself, a free asset* from the Unity Store for a rigged humanoid character model. Then, using the Unity's built-in ragdoll creation tools (GameObject->3D Object->Ragdoll…) setup the character joints and rigidbodies of the character.

The actual active part of the ragdoll is composed of multiple custom scripts.

The first script is used to push and rotate rigidbodies to a target position and rotation.

The second script is used to calculate the wanted position and rotation of a few parts of the ragdoll. Then using the first script, the ragdolls certain parts are pushed and rotated towards the calculated target spots. To keep the ragdoll standing, a raycast is casted, to calculate where the ground is, relative to the ragdoll.

* Lopoly Ninja by Tetra Arts (https://assetstore.unity.com/packages/3d/characters/humanoids/lowpoly-ninja-157942)

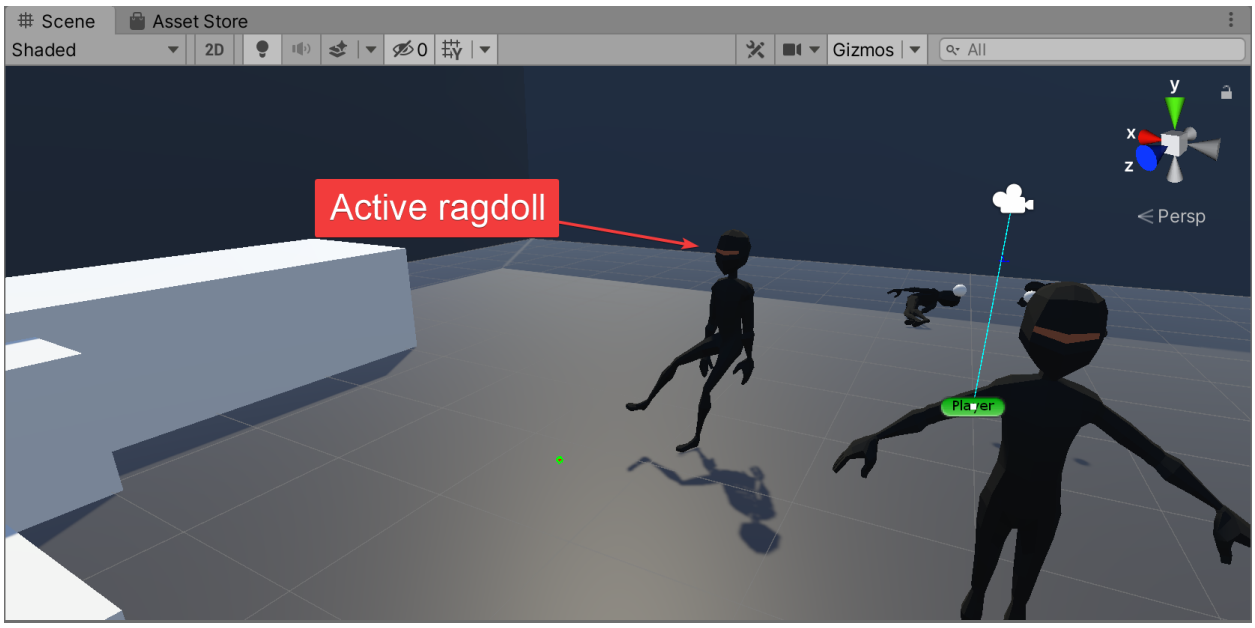



**Figure 5. Active ragdoll, trying to stand upright**

## Laboratory 1 defense task

Create a Microsoft logo using a plane for logo surface and area lights for colors.
Steps:

1. Created a 3D plane
2. Marked the plane as static, to enable baked lighting for it
3. Added four area lights (colors: red, green, blue, yellow)
4. In Window->Rendering->Lighting settings enabled “Auto generate”, to enable automatically baking lightmaps

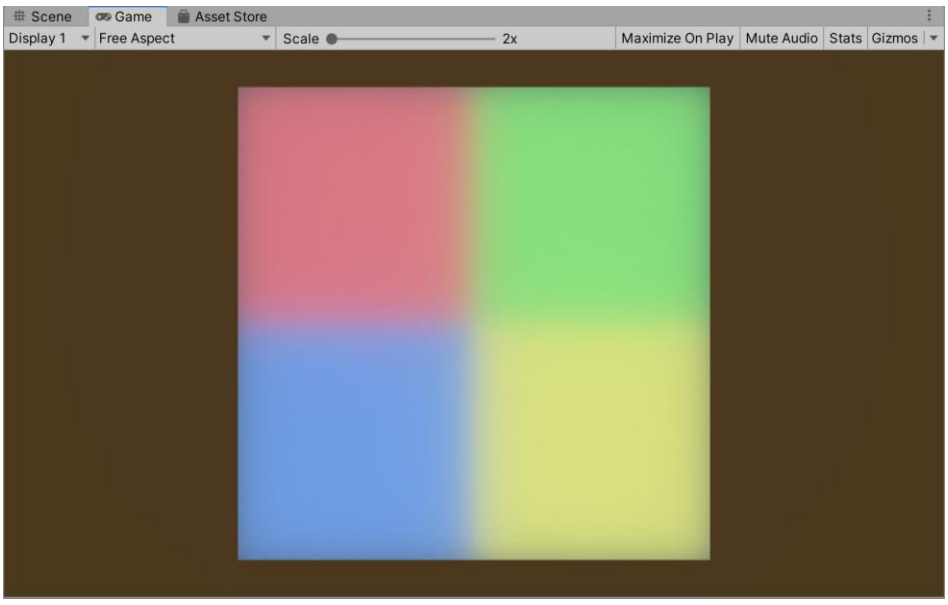



**Figure 6. Lab 1 defense result**

# Laboratory work #2

## List of tasks

1. Experiment with ProBuilder tools, start expanding, building your World environment;
2. Add animations to your game character (Assuming you have a working PlayerController. If you don't, create your own or get them from.
3. Create and/or animate 5 objects of your choice in your World (you can use Animator Component and/or Timeline);
4. Create at least 5 particle effects for your environment (dust, explosion, smoke gas, light sparks, etc.);
5. Add a custom skybox;
6. Create at least 3 different Physics Materials for various parts of the map (either it's a slippery platform/ice, bouncy wall, non-slippery ground, etc.);
7. Create 4 types of objects that use OnCollisionEnter / OnCollisionEnter2D / OnTriggerEnter / OnTriggerEnter2D.
8. Assign optimal colliders for the environment objects that are not moving and set their flags to static (test performance before and after);
9. Bake a lightmap and measure performance;
10. Optimize all textures depending on their parameters and measure graphical memory load;
11. Try hard vs soft shadows, different quality settings and measure the performance;

## Solution

## Task #1. *Experiment with ProBuilder tools, start expanding, building your World environment*

Created some custom fans with ProBuilder for an AC unit simple model.

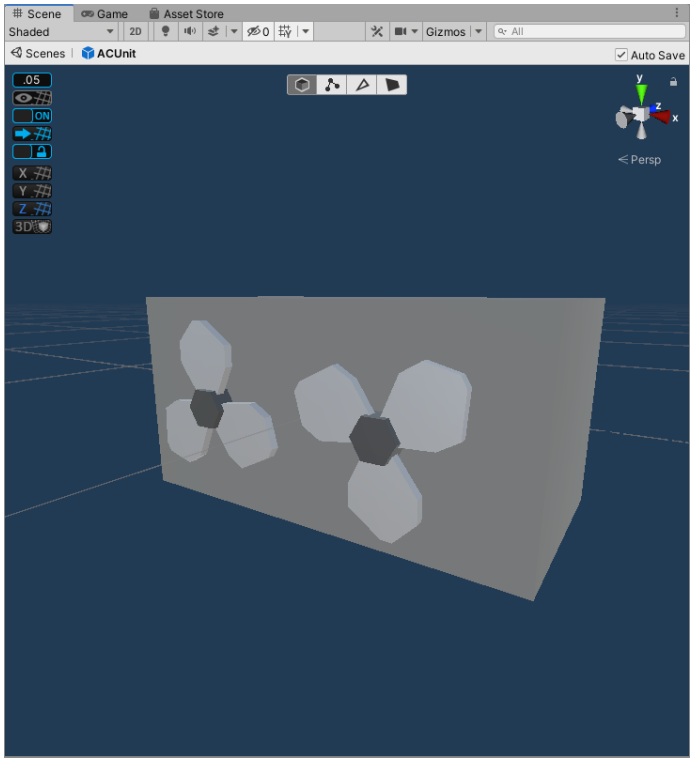



**Figure 7. Ac unit with custom fans made with ProBuilder**

## Task #3. *Create and/or animate 5 objects of your choice in your World*

Animated AC Unit propellers to spin: one spins faster than the other.

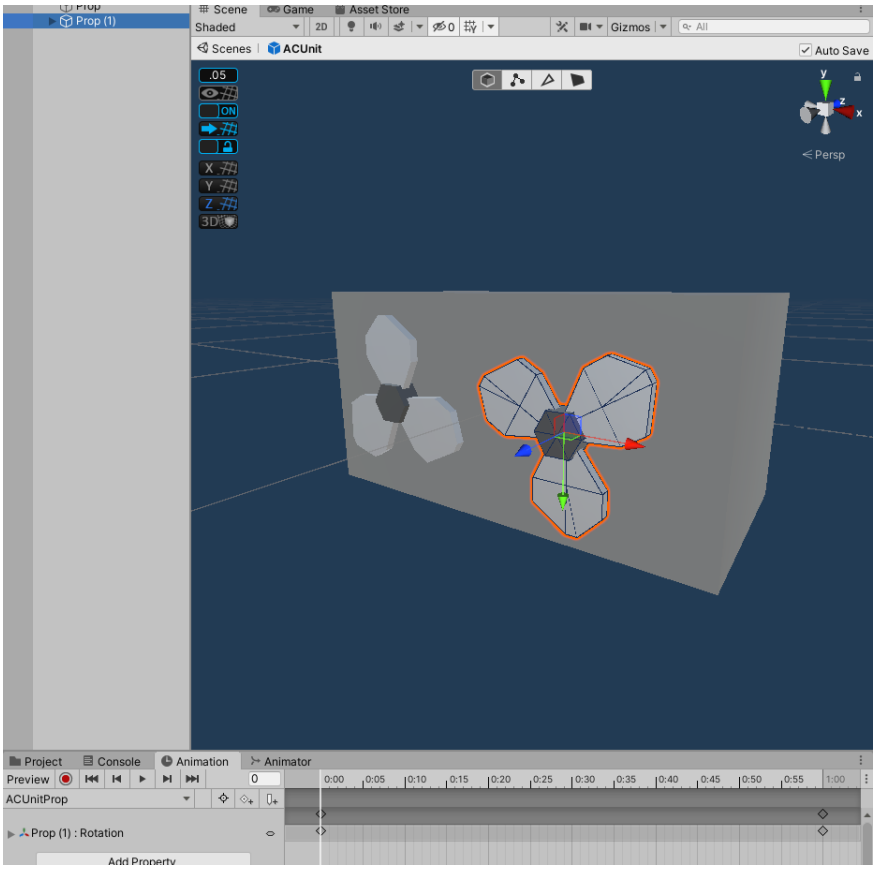



**Figure 8. AC Unit propeller animation**

Also animated a light inside the warehouse to flicker.


**Figure 9. Flickering light animation**

## Task #4. *Create at least 5 particle effects for your environment*

Added a mist particle effect using a custom-made texture.

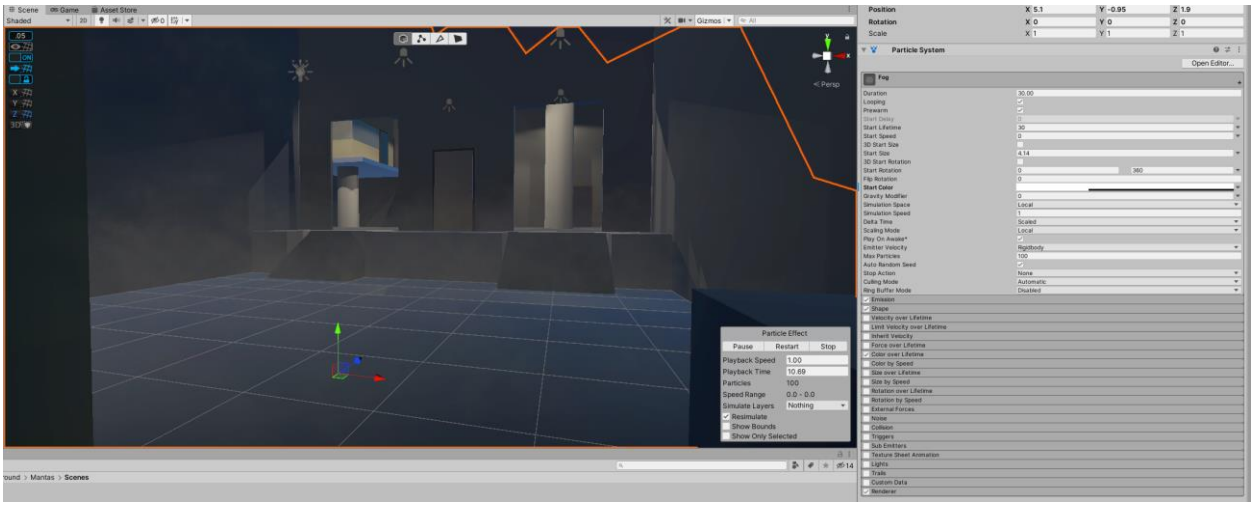


**Figure 10. Mist particle effect using a custom texture**

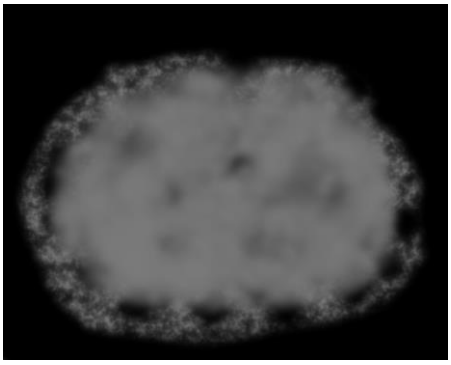


**Figure 11. Custom mist texture**

Also added a sparks particle effect to the flickering light in the warehouse with a custom spark texture.

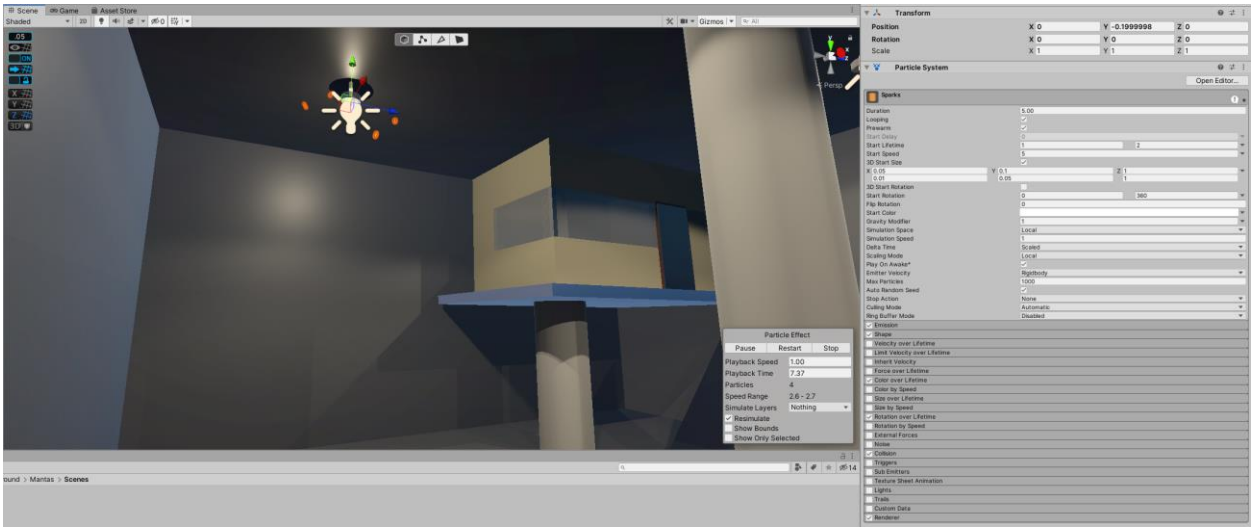


**Figure 12. Sparks particle effect**

**Figure 13. Spark custom texture**

Added some ambiguous particle effect for the AC Unit, falling from one of the propellers.

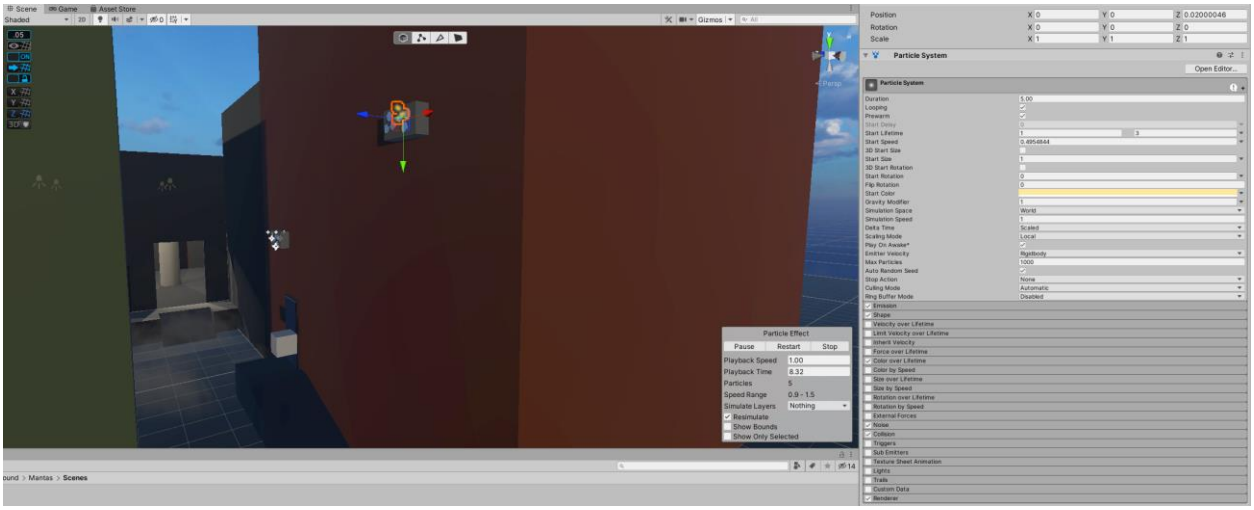


**Figure 14. Ambiguous particle effect for AC Unit propeller**

## Task #5. *Add a custom skybox*

Imported a free asset (AllSky Free) from the asset store for a custom skybox. Then created a custom material, applied it to the skybox settings in the lighting window and rotated the skybox according to the directional light of the scene.

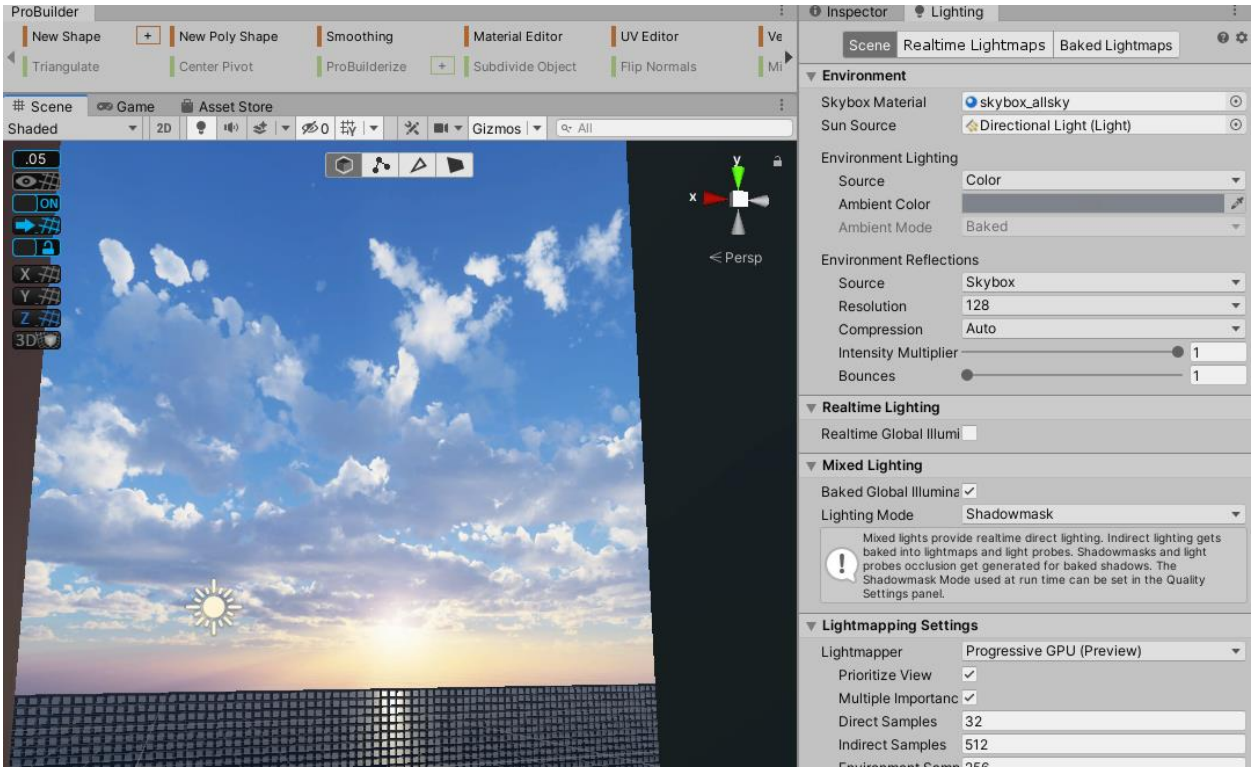



**Figure 15. Custom skybox**

## Task #6. *Create at least 3 different Physics Materials for various parts of the map*

Added a soccer ball (free model downloaded from Asset Store, Soccer Ball) and created a bouncy physic material.

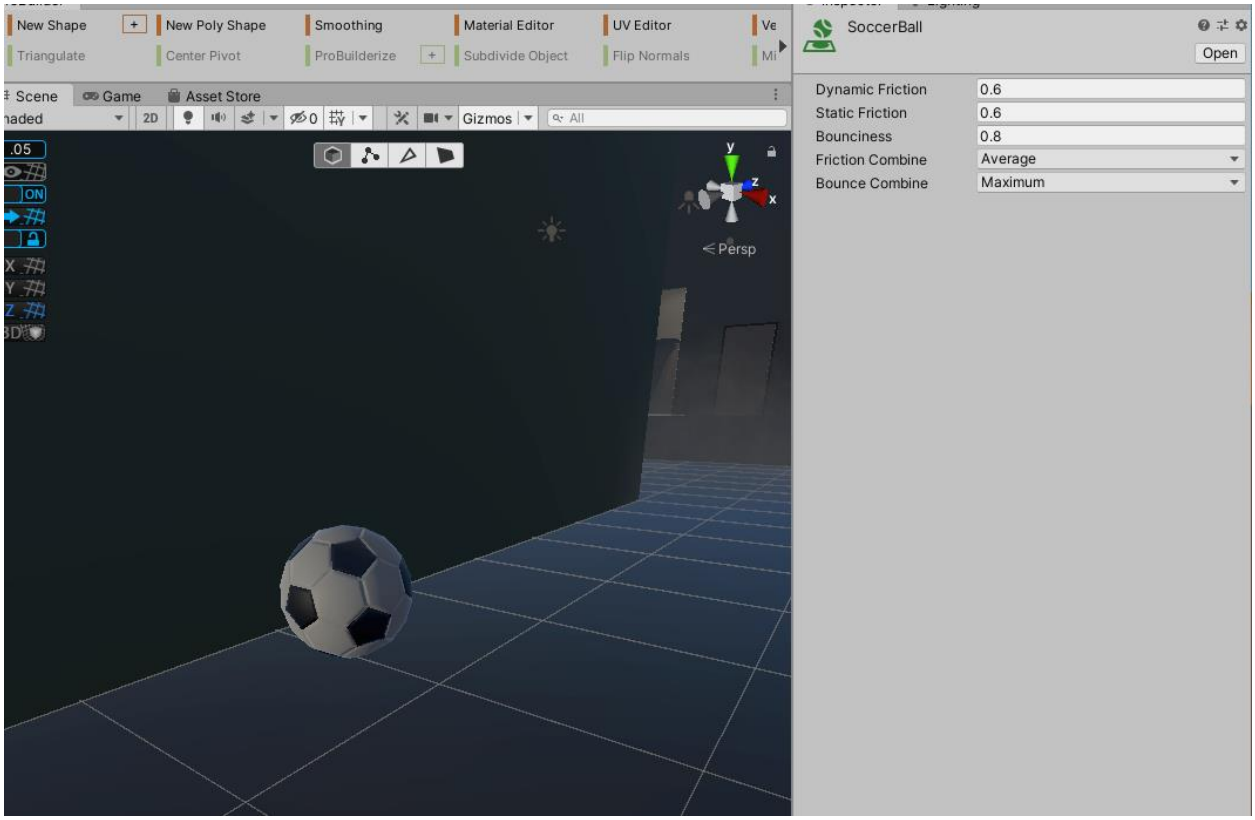



**Figure 16. Soccer ball with a bouncy physic material**

Also made a slippery ramp physic material.

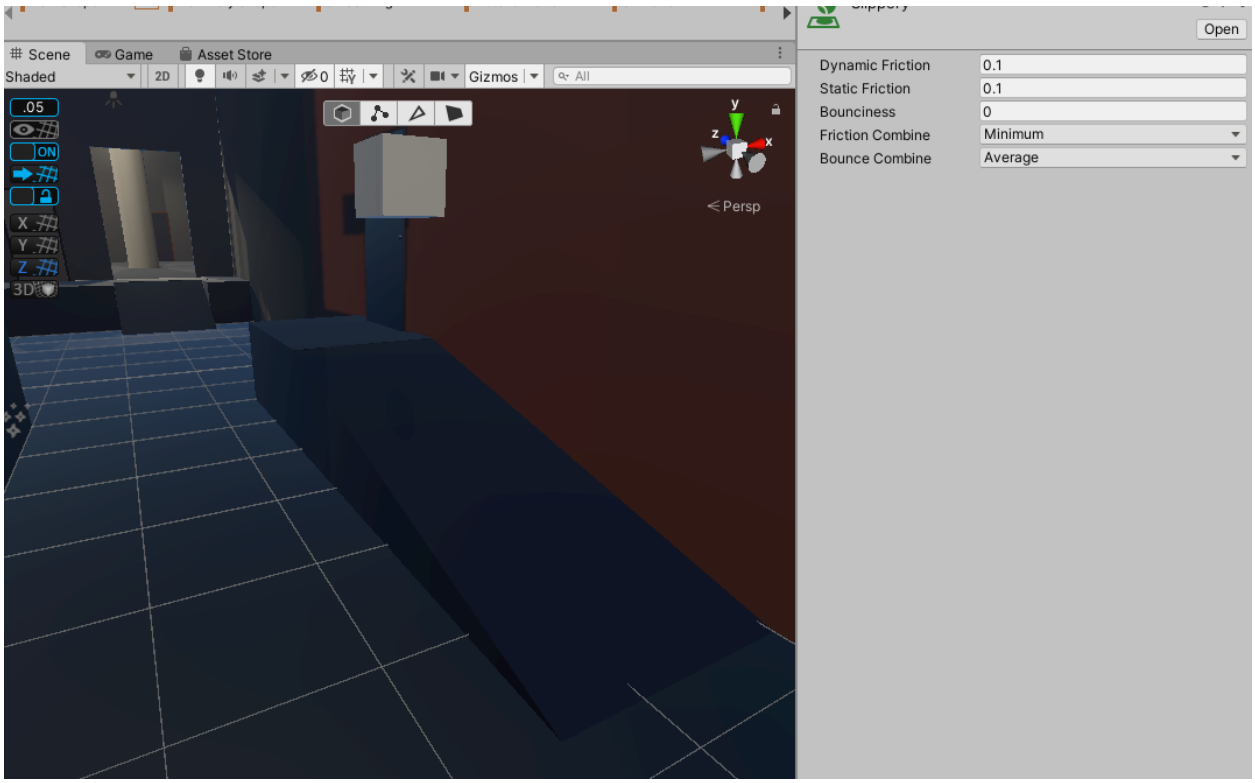



**Figure 17. Slippery ramp physic material**

## Task #7. *Create 4 types of objects that use OnCollisionEnter / OnCollisionEnter2D / OnTriggerEnter / OnTriggerEnter2D*

Active ragdolls are now a bit improved. Right now, it is possible to apply an animation to the ragdoll, and the ragdoll will try to mimic the animation, as closely as possible. The character was also redone with configurable joints instead of character joints to be able to be able to use the joint drives, for specifying the target rotations of the joints, which provides more physically accurate character movements.

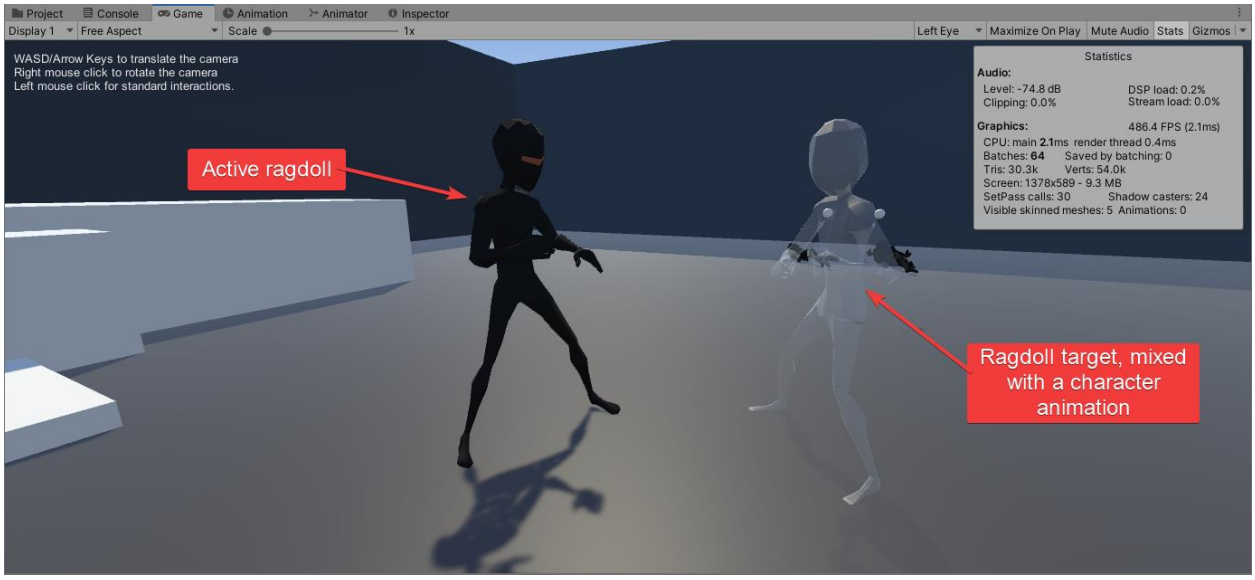



**Figure 18. Active ragdoll, following an animation**

## Task #8. *Assign optimal colliders for the environment objects that are not moving and set their flags to static (test performance before and after)*

To test physics performance using static objects, I duplicated a bunch of falling balls all around the scene and inspected the profile on how much CPU time is used on physics. Since there is not a lot of physics going on in the game world, there was no noticeable performance difference.

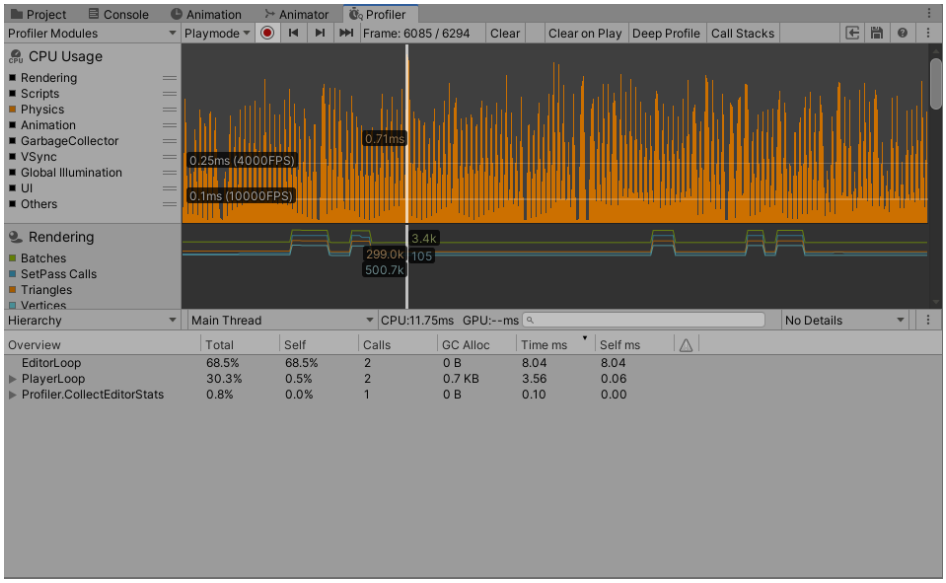



**Figure 19. Before marking objects as static**

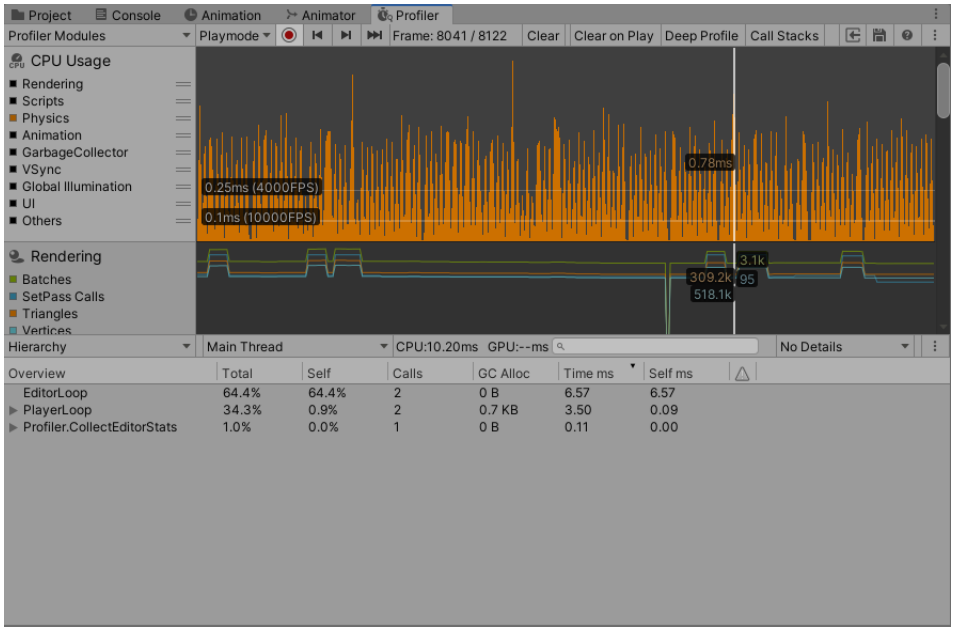



**Figure 20. After marking objects as static**

## Task #9. *Bake a lightmap and measure performance*

To test light performance, I set all non-moving objects to static, including lights, enabled mixed mode in all static lights and baked the lightmaps using the default lightmapper settings. The game uses a differed rendering path.

There was no noticeable performance improvement, although render thread CPU time decreased from 1.1ms to 0.9ms, batches decreased from 294 to 65 and saved by batching increased from 4 to 260. With a more complex map, this would definitely have a more noticeable difference.

The quality of lighting around the map also increased, because the lightmapper baked in ambient occlusion and global illumination.

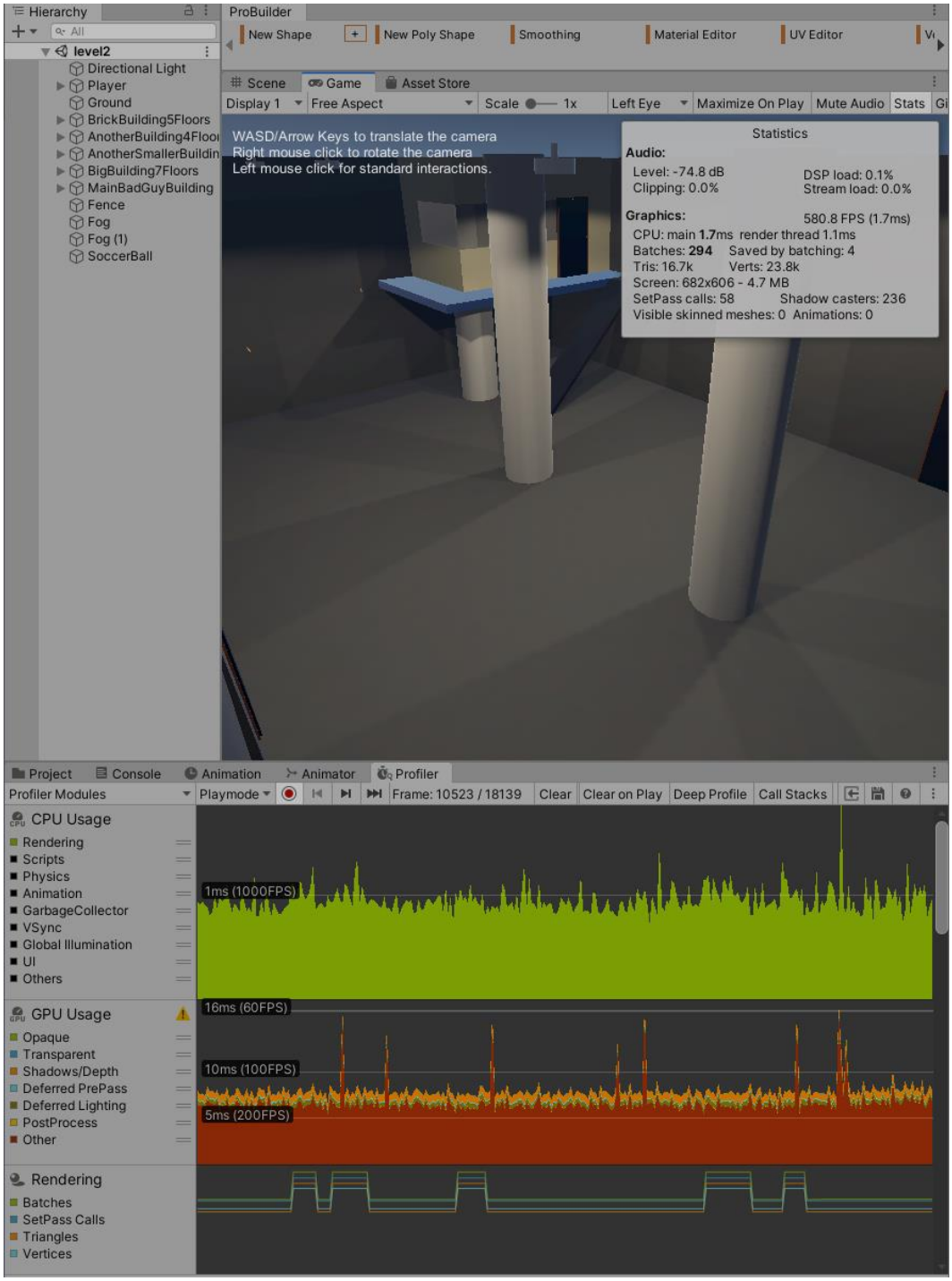



**Figure 21. Light performance before lightmapping**

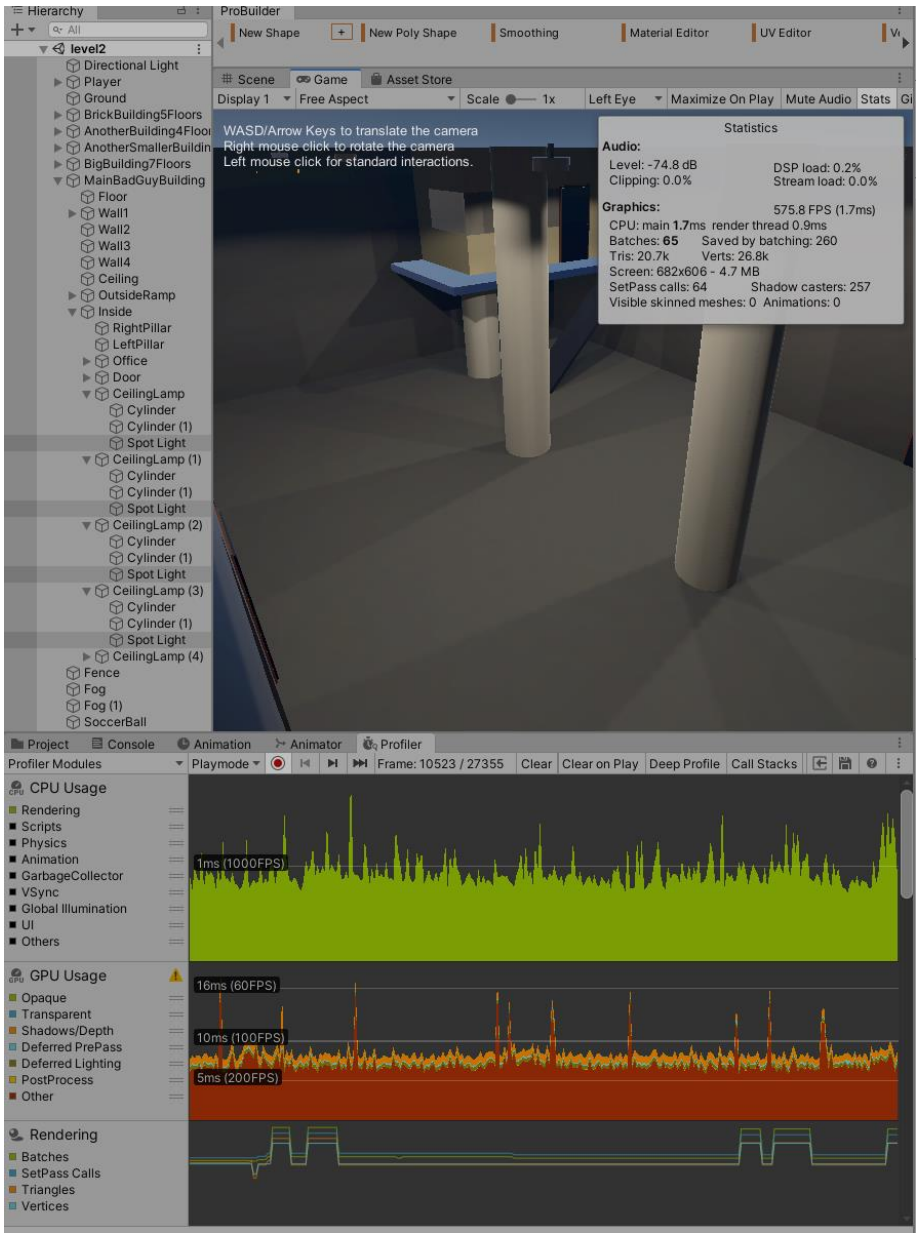



**Figure 22. Light performance after lightmapping**

## Task #10. *Optimize all textures depending on their parameters and measure graphical memory load*

There are almost no textures used in the game world, since the game uses a simple low-poly, no-textures (flat) design.

## Task #11. *Try hard vs soft shadows, different quality settings and measure the performance*

To test the performance difference of shadows, I tested with them on and off. Turning off the shadows reduced the render thread from 0.9ms to 0.8ms, which in turn increased FPS by a bit.

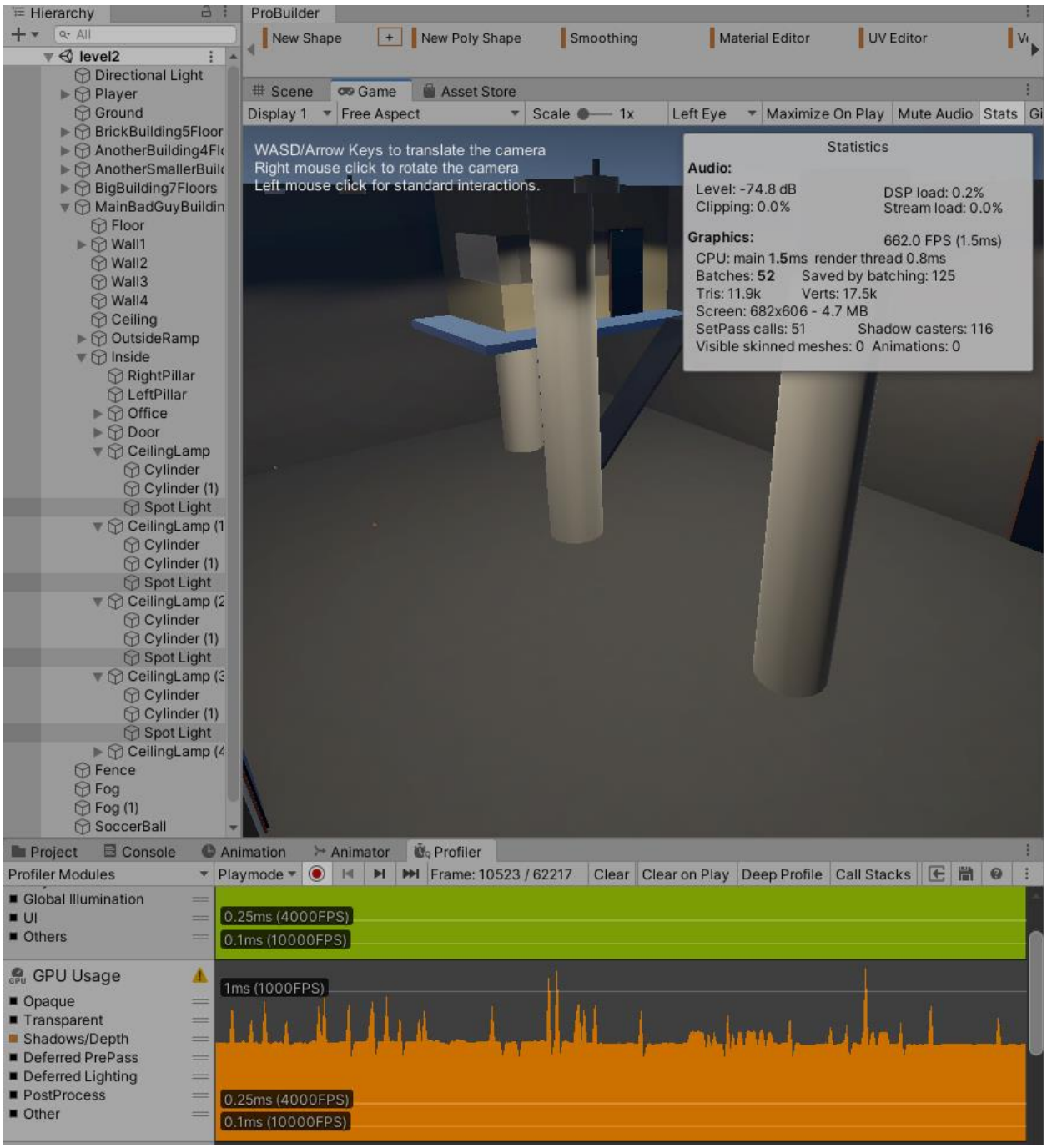



**Figure 23. Scene without shadows**

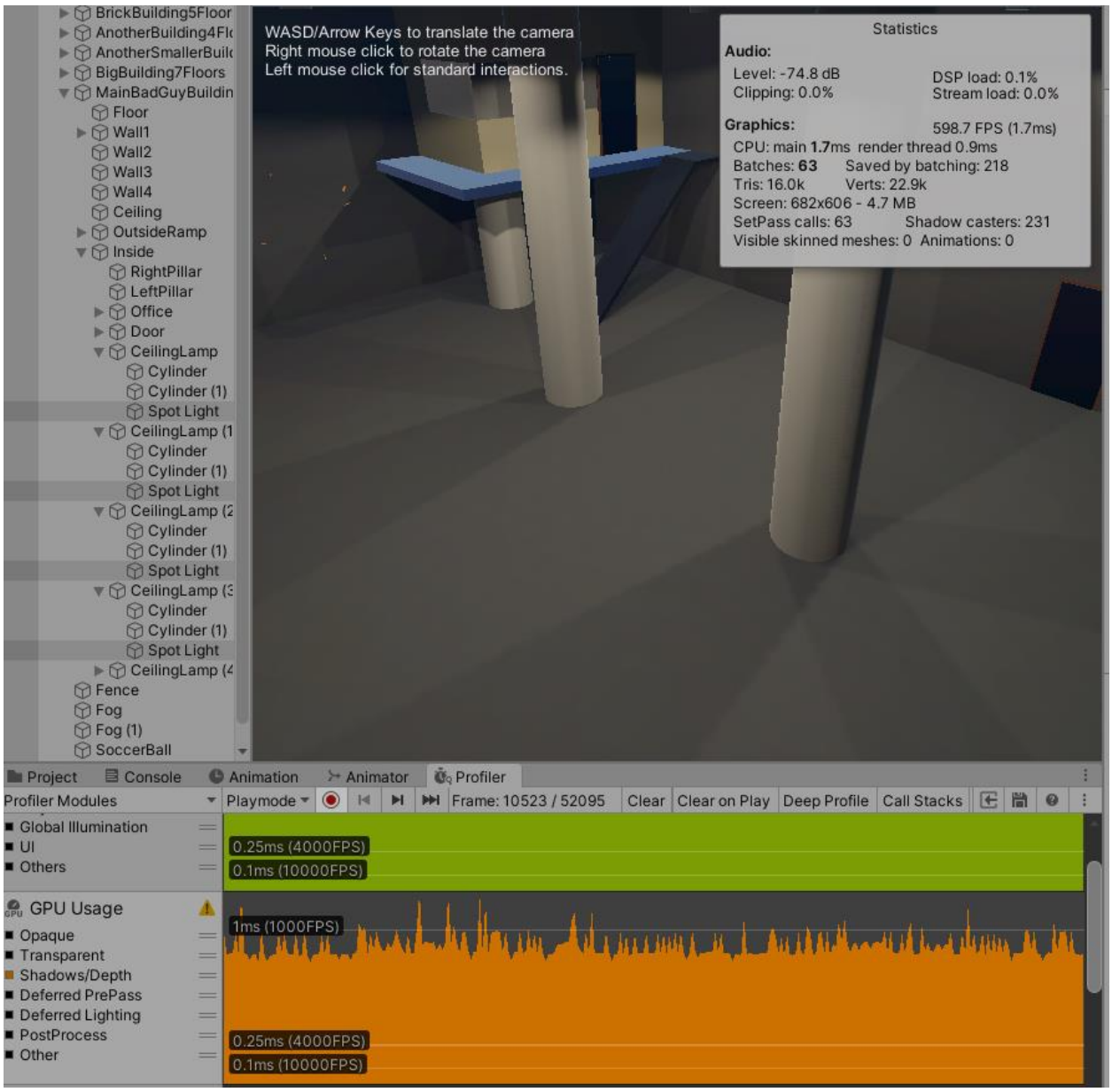



**Figure 24. Scene with soft shadows**

# *Defense task*

Add a plane and a ball (with a rigidbody) to the scene and make it bouncy infinitely on the plane using a script.

1. Added a plane and a ball
2. Attached a rigidbody on the ball
3. Created a script called “BouncyBall” on the ball
4. Added this code to the script:

```
public class BouncyBall : MonoBehaviour {
  private new Rigidbody rigidbody;

  private void Awake() {
    rigidbody = GetComponent<Rigidbody>();
  }

  private void OnCollisionEnter(Collision other) {
    rigidbody.AddForce(0, 5, 0, ForceMode.Impulse);
  }
}
```

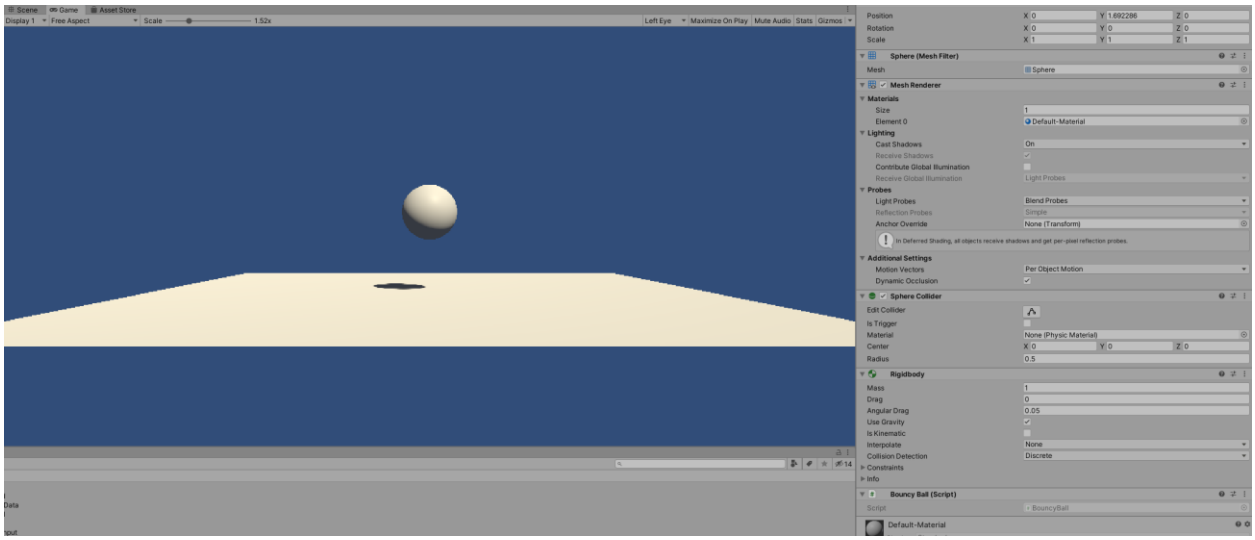


**Figure 25. Defense task: infinitely bouncing ball**

# Laboratory work #3

## List of tasks

1. Add a MENU system to your game
2. Add OPTIONS
3. Game must have a GUI
4. Add attack mechanics
5. Implement Opponents
6. Add health / powerup mechanics and indication in the GUI
7. Add scoring system
8. Add "Game over" condition
9. Add "Post Processing"
10. Add Interactive sounds

## Task #1. *Add a MENU system to your game*

Added a pause menu.

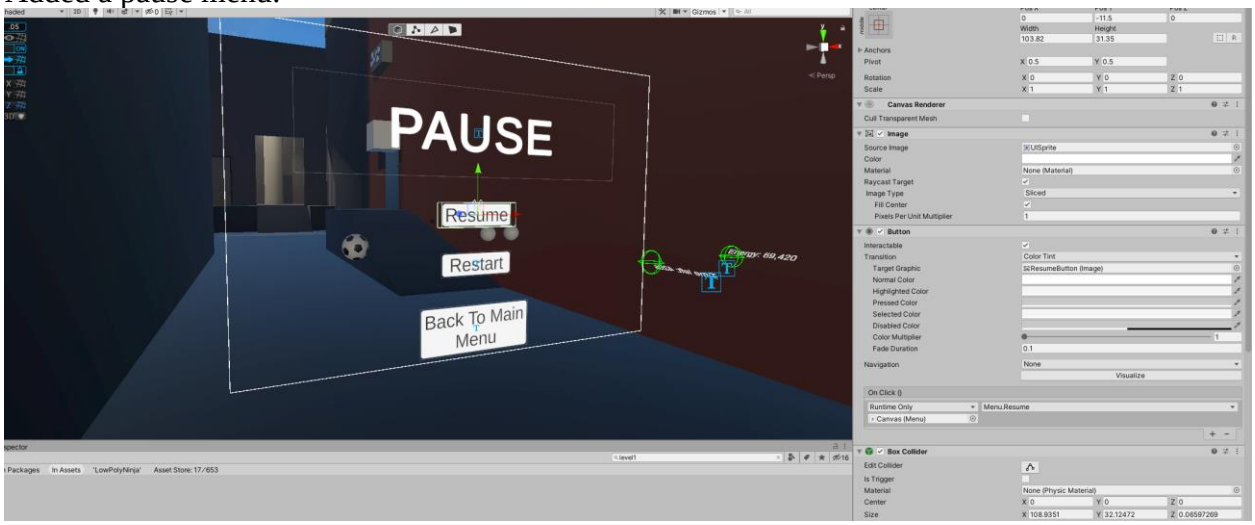



**Figure 26. In-game pause menu**

Added a game over menu.

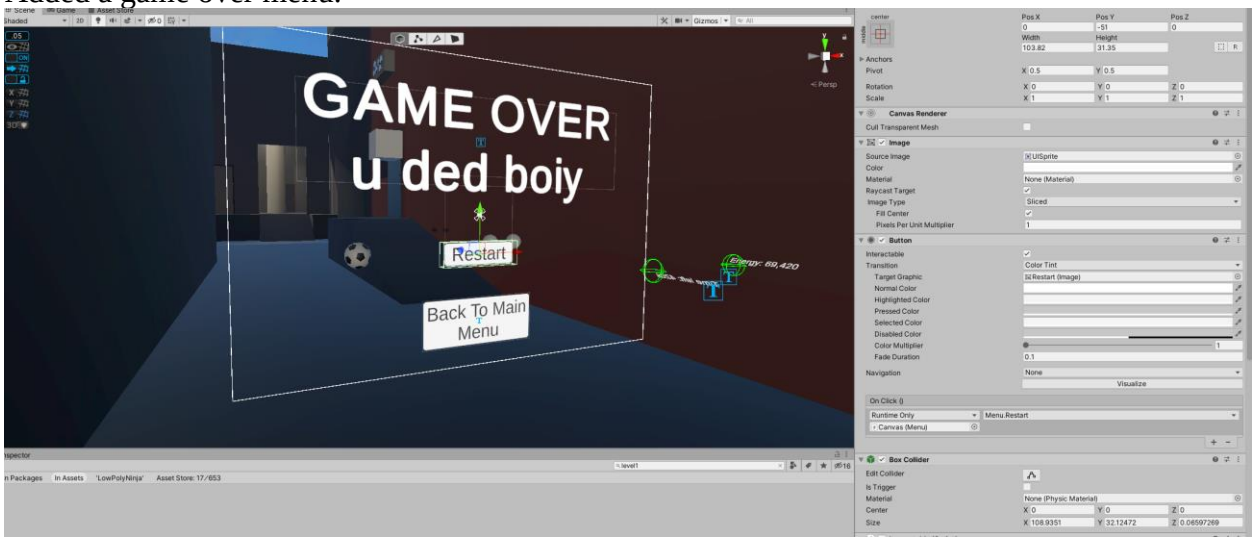



**Figure 27. Game-over menu**

Added a game completed menu.

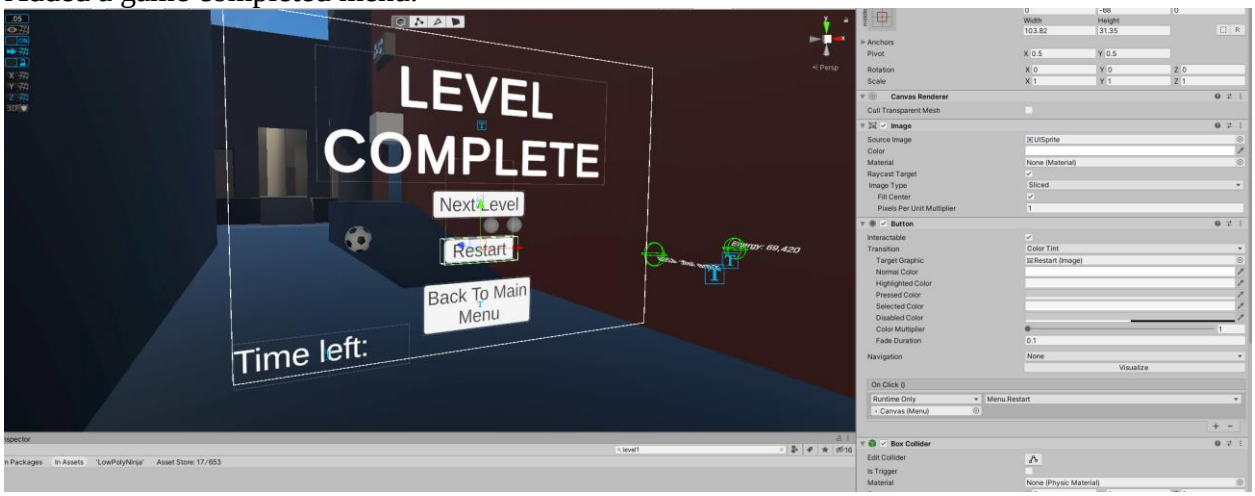



**Figure 28. Level complete menu**

## Task #2. *Add OPTIONS*

Added buttons for music and sound sliders in the options menu.

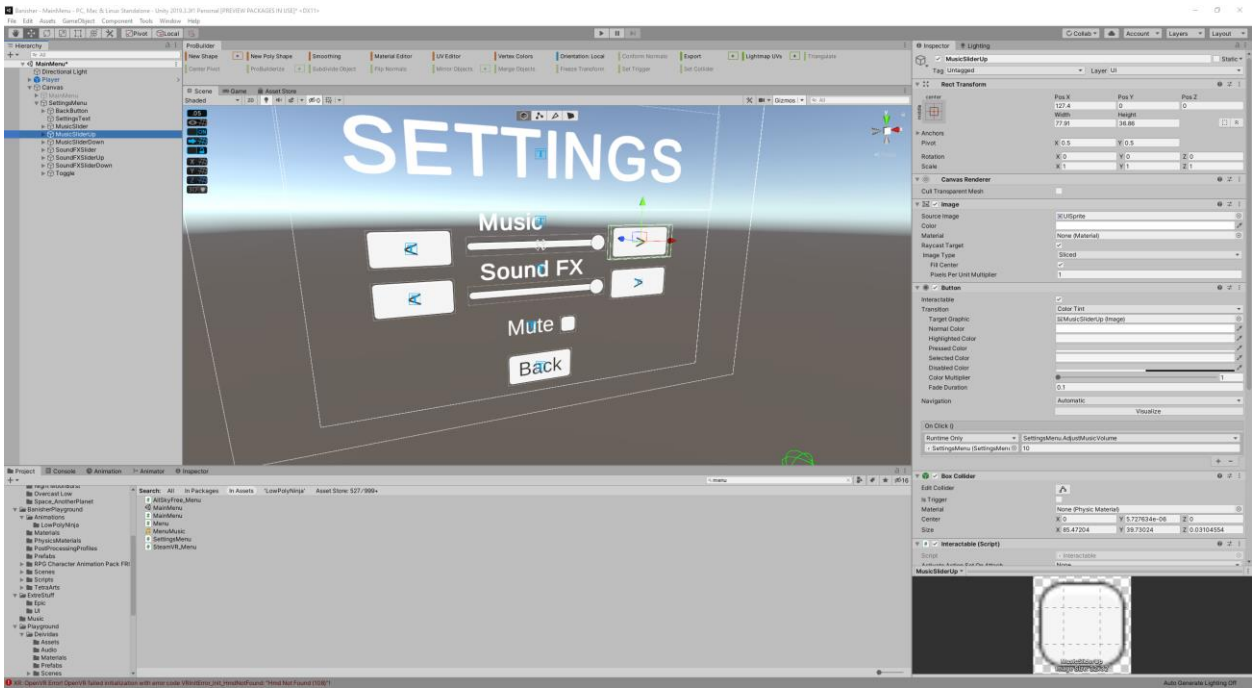



**Figure 29. Buttons for sliders in options**

```
public void AdjustMusicVolume(float amount)
{
    SetMusicVolume(musicSlider.value + amount);
}

2 asset usages
public void AdjustSoundFxVolume(float amount)
{
    SetSoundFxVolume(musicSlider.value + amount);
}
```

**Code 1. Functions called when slider adjusting buttons are clicked**

## Task #3. *Game must have a GUI*

**Vis energy UI.**

The player can see their current Vis energy level by looking at their right arm.

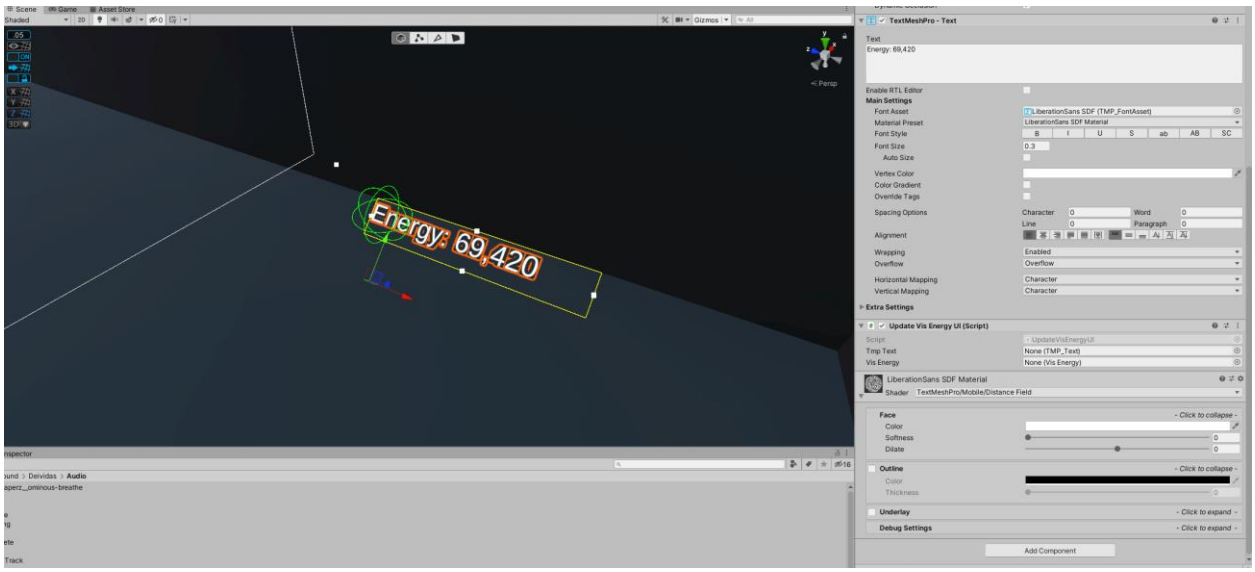



**Figure 30. Energy UI on the right hand of the player**

```
public class UpdateVisEnergyUI : MonoBehaviour  Mantas Klimašauskas
{
    [SerializeField] private TMP_Text tmpText;  Unchanged

    [SerializeField] private VisEnergy visEnergy;  Unchanged

    private void Awake()  Event function  Mantas Klimašauskas
    {
        if (!tmpText) tmpText = GetComponent<TMP_Text>();
        if (!visEnergy) visEnergy = GetComponentInParent<VisEnergy>();
    }

    private void Start()  Event function  Mantas Klimašauskas
    {
        visEnergy.OnChange.AddListener(OnChangeVisEnergy);
        UpdateUI((int) visEnergy.Energy);
    }

    private void OnChangeVisEnergy(VisEnergy visEnergy, float previousEnergy)
    {
        UpdateUI((int) visEnergy.Energy);
    }

    // ReSharper disable once InconsistentNaming
    private void UpdateUI(int energy)  Mantas Klimašauskas
    {
        tmpText.text = $"Vis: {energy}";
    }
}
```

**Code 2. Vis energy UI update script**

## Task #4. *Add attack mechanics*

The enemies and the player have scripts to remove their energy on collisions.

```
[RequireComponent(typeof(Collider))]
8 asset usages
public class RemoveVisEnergyOnCollision : MonoBehaviour {  Mantas Klimašauskas
    [SerializeField] private VisEnergy visEnergy;  Unchanged

    [SerializeField] private AnimationCurve energyDamageCurve = AnimationCurve.Linear(1, 0, 100, 100);  Serializable

    [SerializeField] private bool useColliderWhitelist = false;  Unchanged

    [SerializeField] private List<Collider> colliderWhitelist = new List<Collider>();  Serializable
    [SerializeField] private List<Collider> colliderBlacklist = new List<Collider>();  Serializable

    private void Awake() {  Event function  Mantas Klimašauskas
        if (!visEnergy) visEnergy = GetComponentInParent<VisEnergy>();
    }

    private void OnCollisionEnter(Collision other) {  Event function  Mantas Klimašauskas
        if (useColliderWhitelist && !colliderWhitelist.Contains(other.collider)) {
            return;
        }

        if (!useColliderWhitelist && colliderBlacklist.Contains(other.collider)) {
            return;
        }

        float modifier = 1;
        VisEnergyCollisionModifier collisionModifier = other.collider.GetComponent<VisEnergyCollisionModifier>();

        if (collisionModifier) {
            modifier = collisionModifier.ReducedVisEnergyModifier;
        }

        float impulse = other.impulse.magnitude * modifier;
        visEnergy.Energy -= energyDamageCurve.Evaluate(impulse);
    }
}
```

**Code 3. RemoveVisEnergyOnCollision component**

## Task #5. *Implement Opponents*

Every enemy and the player have a VisEnergy component, which manages their current energy level.

```
public class VisEnergy : MonoBehaviour {  Mantas Klimašauskas
    [Serializable]
    1 exposing API
    public class VisEnergyChangeEvent : UnityEvent<VisEnergy, float> { }

    [SerializeField] private float energy = 100;  "50"
    [SerializeField] private float minEnergy = 0;  Unchanged
    [SerializeField] private float maxEnergy = 100;  "50"

    [SerializeField] private VisEnergyChangeEvent onChange;  Serializable

    public float Energy {  Mantas Klimašauskas
        get => energy;  Frequently called
        set {  Frequently called
            value = Mathf.Clamp(value, minEnergy, maxEnergy);

            // ReSharper disable once CompareOfFloatsByEqualityOperator
            if (energy != value) {
                float previousEnergy = energy;

                energy = value;

                if (enabled) {
                    onChange.Invoke(this, previousEnergy);
                }
            }
        }
    }

    public float MinEnergy {  Mantas Klimašauskas
        get => minEnergy;
        set => minEnergy = value;
    }

    public float MaxEnergy {  Mantas Klimašauskas
        get => maxEnergy;  Frequently called
        set => maxEnergy = value;
    }

    public VisEnergyChangeEvent OnChange => onChange;  Mantas Klimašauskas
}
```

**Code 4. VisEnergy component**

The enemies are active ragdolls (fully physics controlled). Each enemy consists of several inner character objects:

- One acts as the ragdoll, has all the joints, colliders, rigidbodies.
- One acts as the animated character.

- One maps the rotations from the animated character to itself. The ragdoll joints then try to adjust according to this character's rotations using joint drives.

The active ragdoll then stands by using a raycast, to check where the ground is. It only tries to stand when it is touching anything and there's a ground nearby.

For enemies to see the player, they are constantly casting rays towards the player. The ray cast has a distance limit and an FOV limit, so the player must be close and in front of an enemy to be detected. If the enemy loses sight of the player, they move towards the last seen position of the player and stop there.

The enemy is moved by applying a small amount of force to the body of the enemy, in the direction of the destination. The enemy is also rotated (using a torque) towards its destination. Also, a running animation is played when the enemy is moving.

## Task #6. Add health / powerup mechanics and indication in the GUI

**Banishment.**
The player can banish themselves (enter the slow-motion mode), by pressing a key on the controller.

```
public class PlayerBanish : MonoBehaviour {  Mantas Klimašauskas
    [SerializeField] private SteamVR_Action_Boolean banishButton;  Serializab
    [SerializeField] private KeyCode banishKey = KeyCode.Space;  Unchanged

    [SerializeField] private BanishManager banishManager;  Unchanged

    private void Awake() {  Event function  Mantas Klimašauskas
        if (!banishManager) banishManager = BanishManager.GetInstance();
    }

    private void Update() {  Event function  Mantas Klimašauskas
        if (Input.GetKeyDown(banishKey) || banishButton.stateDown) {
            banishManager.ToggleExsilium();
        }
    }
}
```

**Code 5. Player banish script**

When banished, the slow motion is activated and camera post-process effects are also activated.

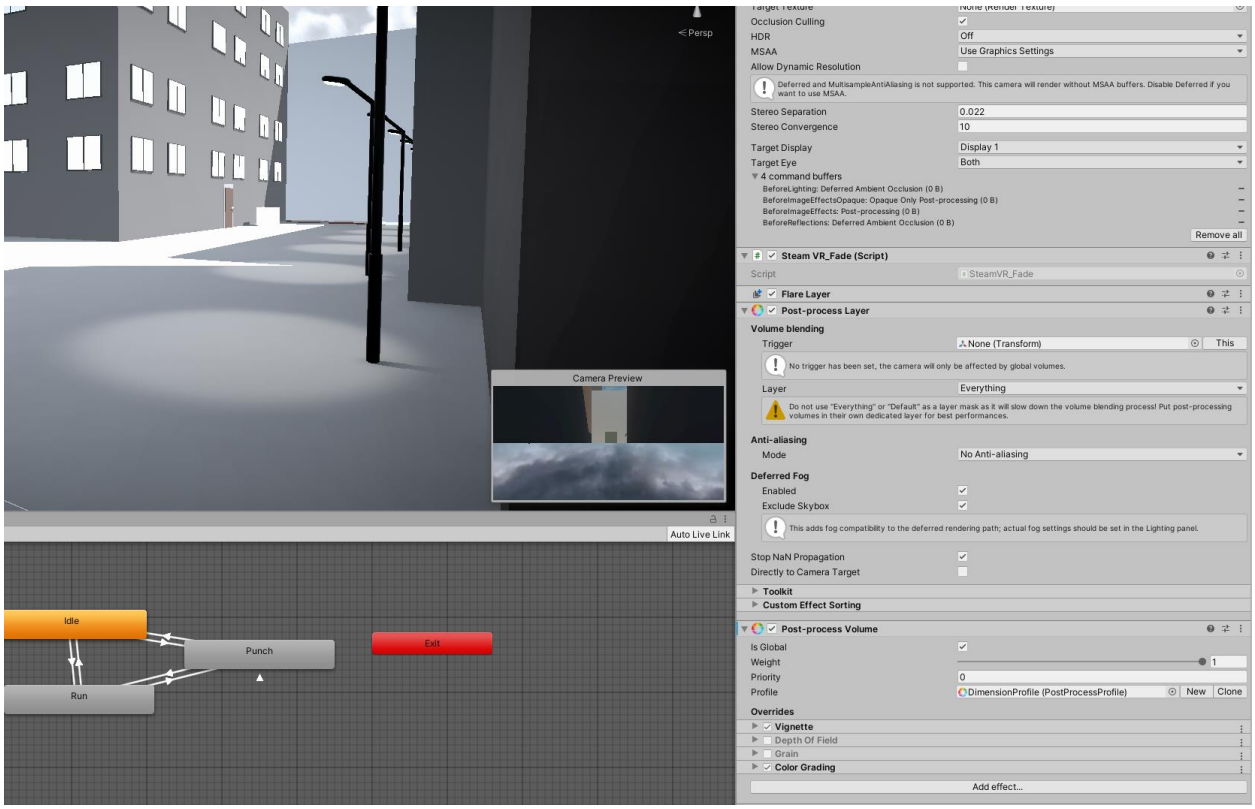



**Figure 31. Camera post-process effects**

## Task #8. *Add "Game over" condition*

The currently alive ninjas are being tracked by the NinjaManager.

```
public class NinjaManager : MonoBehaviour  Mantas Klimašauskas
{
    public const string NinjaManagerTag = "NinjaManager";

    [SerializeField] private List<ActiveRagdoll> ninjas = new List<ActiveRagdoll>();

    [SerializeField] private UnityEvent onNinjasChange;  Serializable

    public int CurrentNinjasAlive => ninjas.Count(ninja :ActiveRagdoll  => ninja.IsAlive);

    public UnityEvent OnNinjasChange => onNinjasChange;  Mantas Klimašauskas

    private void Start()  Event function  Mantas Klimašauskas
    {
        foreach (ActiveRagdoll ninja in ninjas)
        {
            ninja.VisEnergy.OnChange.AddListener(OnNinjaVisEnergyChange);
        }
    }

    private void OnNinjaVisEnergyChange(VisEnergy visEnergy, float previousEnergy)
    {
        onNinjasChange.Invoke();
    }

    public static NinjaManager GetInstance()  Mantas Klimašauskas
    {
        return GameObject
            .FindWithTag(NinjaManagerTag) // GameObject
            .GetComponent<NinjaManager>(); // NinjaManager
    }
}
```

**Code 6. NinjaManager**

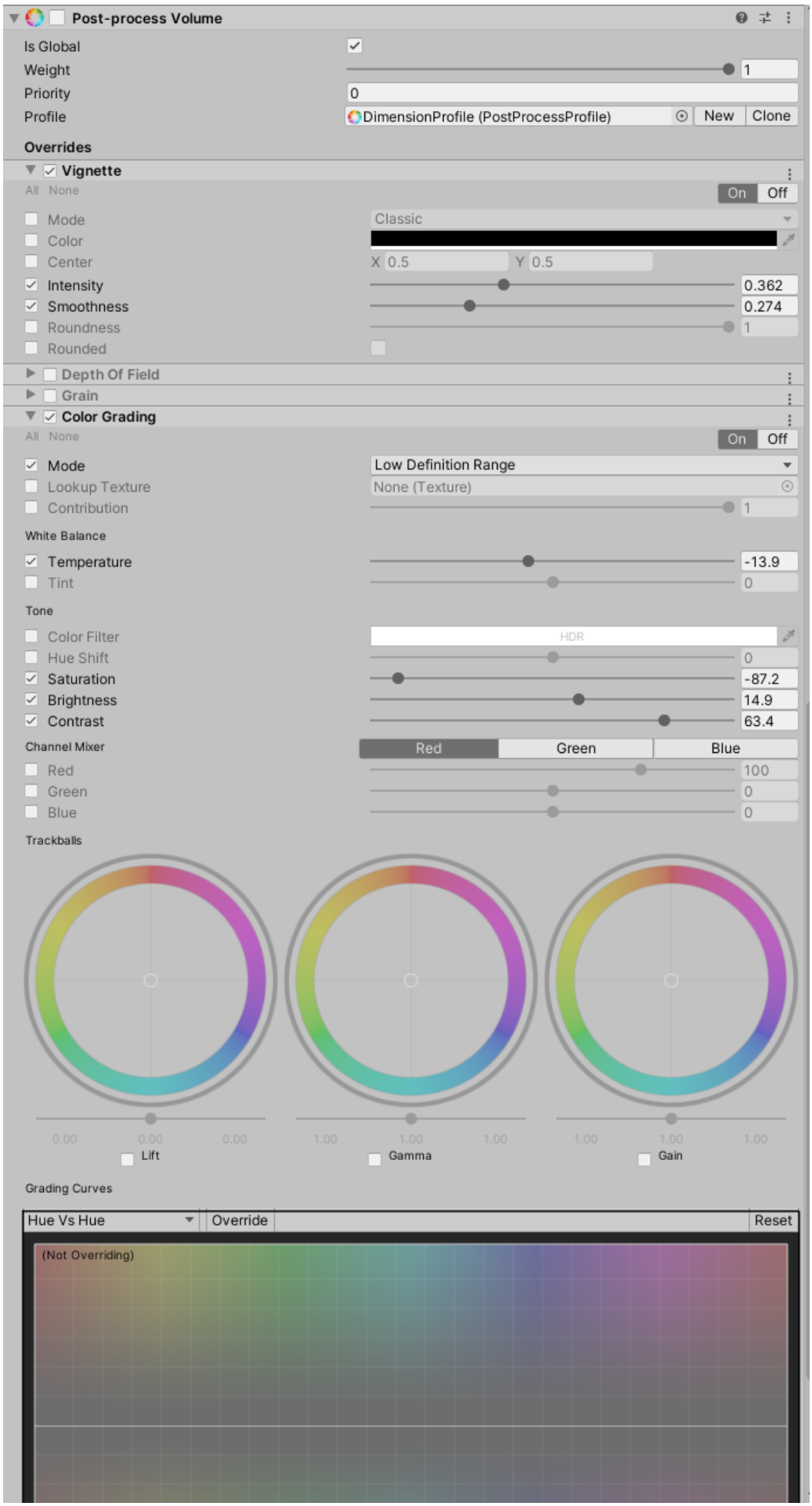
Whenever a ninja dies, the ninja manager is notified, and the ninja manager notifies any other listeners. When the game detects that no ninjas are left alive, the game completed screen is activated.

```
void Start()  Event function  Mantas Klimašauskas +1
{
    Time.timeScale = 1f;
    pauseGame = false;
    gameEnded = false;

    DisableComponents();

    visEnergy.OnChange.AddListener(OnChangeVisEnergy);
    ninjaManager.OnNinjasChange.AddListener(OnNinjasChange);
}

private void OnNinjasChange()  Mantas Klimašauskas +1
{
    if (ninjaManager.CurrentNinjasAlive <= 0)
    {
        ShowLevelCompletedMenu();
    }
}
```

**Code 7. Listening to ninja count change**

## Task #9. *Add "Post Processing"*

When banished two pos-processing effects are activated:


Post-process Volume
Is Global
Weight
1
Priority
0
Profile
DimensionProfile (PostProcessProfile)
New
Clone
Overrides
Vignette
All None
On Off
Mode
Classic
Color
Center
X 0.5
Y 0.5
Intensity
0.362
Smoothness
0.274
Roundness
1
Rounded
Depth Of Field
Grain
Color Grading
All None
On Off
Mode
Low Definition Range
Lookup Texture
None (Texture)
Contribution
1
White Balance
Temperature
-13.9
Tint
0
Tone
Color Filter
HDR
Hue Shift
0
Saturation
-87.2
Brightness
14.9
Contrast
63.4
Channel Mixer
Red
Green
Blue
Red
100
Green
0
Blue
0
Trackballs
0.00 0.00 0.00
Lift
1.00 1.00 1.00
Gamma
1.00 1.00 1.00
Gain
Grading Curves
Hue Vs Hue
Override
Reset
(Not Overriding)

## Task #10. *Add Interactive sounds*

The ball in the map instantiates an object with an AudioSource attached to it, whenever it detects a hard-enough collision.

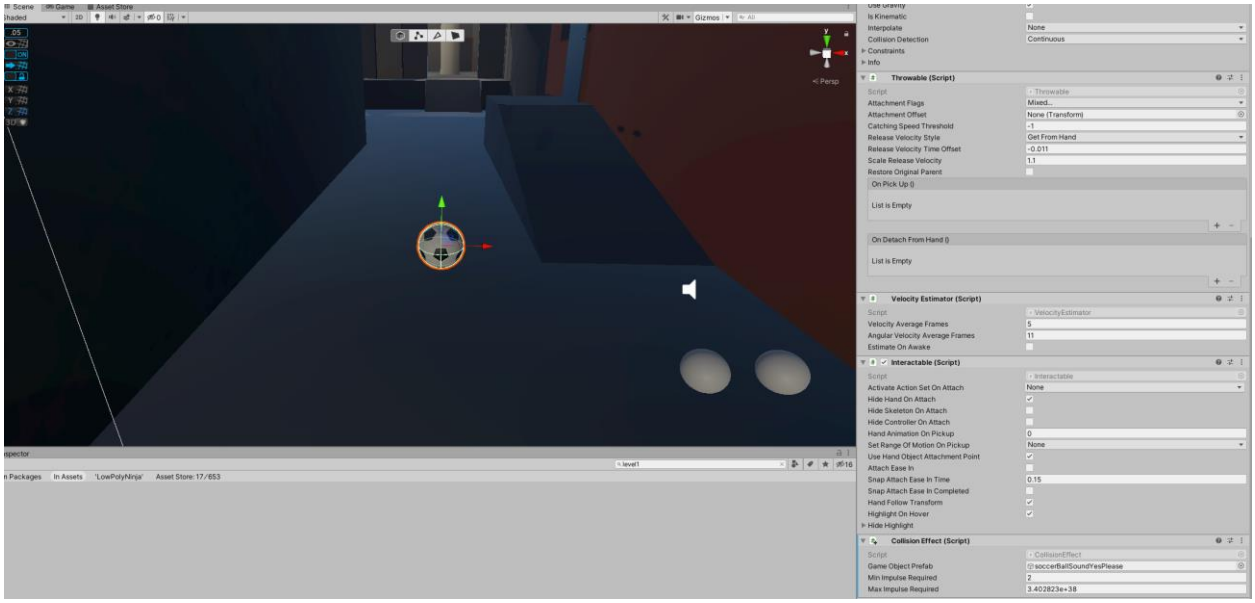


**Figure 32. Ball collision sound**

Enemies also instantiate an object with an AudioSource when their hands hit something.

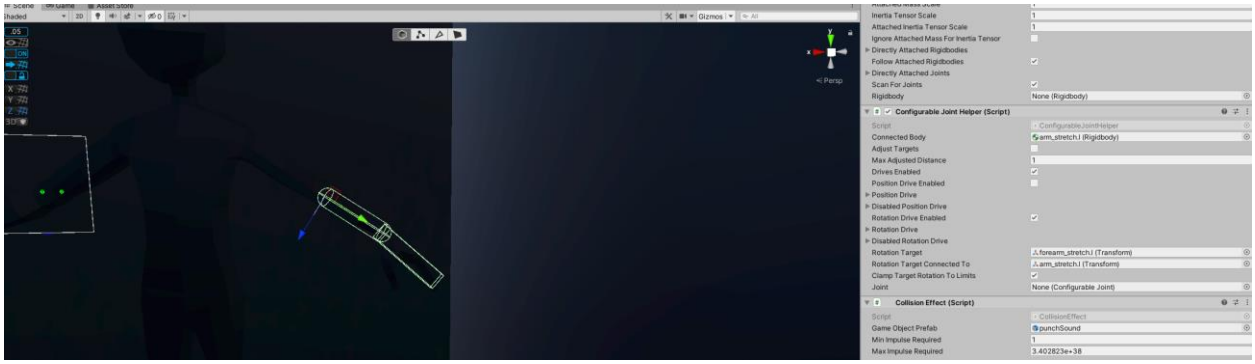


**Figure 33. Enemy hand collision sound**

The player hands also have collision sounds.

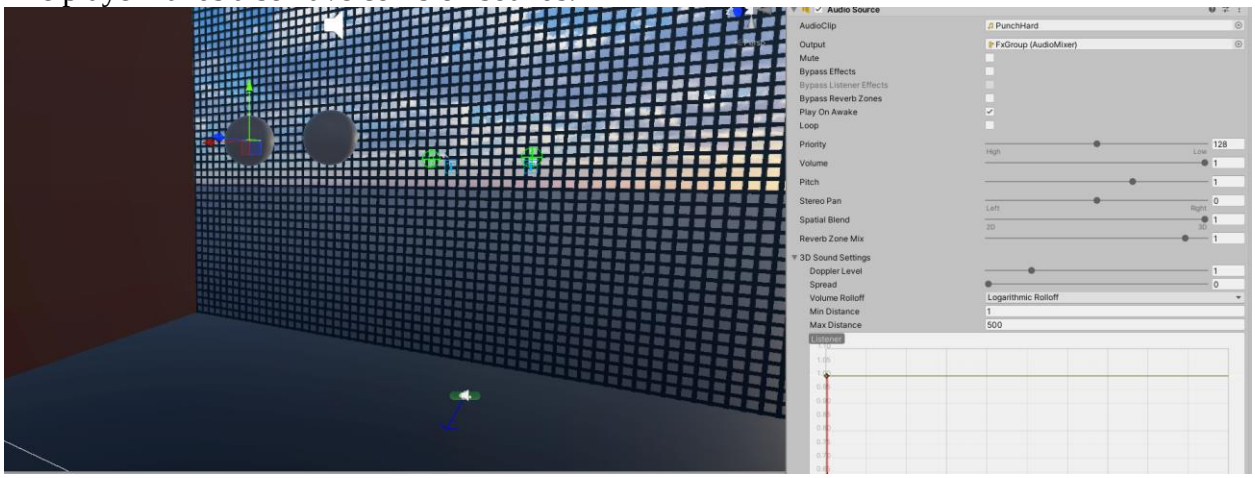


**Figure 34. Player hand collision sound**

# User's manual

**How to play?** The players objective is to kill all enemies before a timer runs out. The player must traverse the levels, find all enemies they can and apply damage to kill them. The player can also use their special ability called “Banish”, which slows down time for the player, so they could react faster.

**Descriptions of the rules of the game**.

- Each enemy and the player have Vis energy associated with them. When Vis reaches 0, the enemy/player dies.
- The energy can be “taken” from the enemies or the player by simply applying collisions on their bodies. The damage taken then depends on the total impulse of the collision.
- Enemy hands and the player hands have a damage multiplier, which means when a hit is registered with those hands, a lot more damage is applied. For example: hitting an enemy with hands will kill it faster than hitting them with a random object.
- The player starts regaining Vis energy automatically if the player didn’t take damage in the past 10 seconds.
- When the player enters the Exsilium (the slow-motion mode), they start losing their energy. To stay in the Exsilium the player must have at least 20 Vis, otherwise the player is automatically unbanished.
- The enemies have a limited field of view and sight range. They only see in front of them (where the head is facing).

**Descriptions of the controls / keys.** The player moves using the thumbstick on the left controller. To toggle banishment, the trackpad must be clicked. Picking up objects can be achieved by holding the trigger or the grip button on any of the controllers, holding them near the item to be picked up.

To enter the pause menu, the menu button must be pressed on any controller. To select menu items, the trigger is used on any controller.

# ANNEX

Source code of the project will be attached with the report.