virtualreality
LABORATORY

Inside Out

Documentation

Version 1.0

Author:

Andrii Khudoli, 807343

Change History

Date	Version	Description	Author
2023-02-24	1.0	Creating of the document	Andrii Khudolii

Table of contest

Change History 1
Table of contest 3
1 Introduction 4
1.1 Motivation 4
1.2 Goal 4
2 Gitlab integration 4
3 Overall project setup 6
4 Graphics setup 7
4.1 HDRP setup 7
4.2 HDRP pipeline profiles 9
5 VR interaction solution 10
5.1 VR interaction framework 10
5.2 Custom wrappers\bridges for VR input 12
6 Coding 13
6.1 Wand controllers 13
6.1.1 Beam-Wand 13
6.1.1.1 Heatmark filler 14
6.1.2 Grab-Wand 16
6.1.2.1 Selectable controller 19
6.2 Speech controller 20
6.3 Brackable controller 20
6.4 Outline controller 22
6.5 Spawner 23
6.5.1 Object distance 23
6.5.2 Spawner controller 24
6.6 Object holder 25
6.7 Global resources 27
7 Visual effects 28
7.1 Outline creation 29
7.2 Heat-beam creation 31
7.3 Grab-beam creation 32
8 Object destruction 34
9 Modelling and Texturing 36
10 Unity AssetStore release 40
11 Conclusion 41
12 References 41

1 Introduction

1.1 Motivation

Creating a project in a virtual reality laboratory was one of the most exciting and rewarding experiences I have ever had. Working with cutting-edge technology and exploring new and innovative applications allowed me to push the boundaries of what is possible and develop skills and expertise that are highly valued in many industries.

In the VR laboratory, I had access to the latest hardware, software, and tools, which allowed me to create immersive and engaging environments that transported users to new and exciting places. Whether I was developing a game, an educational tool, or a simulation, the possibilities were only limited by my imagination.

Not only was the experience creatively fulfilling, but it also had the potential to open up new career opportunities in a rapidly growing and exciting field. Virtual reality technology is already being used in fields such as healthcare, education, entertainment, and gaming, and is expected to become even more prevalent in the coming years. By gaining experience in developing VR projects, I feel well-positioned to take advantage of these opportunities.

Overall, I am incredibly proud of the project that I created in the VR laboratory, and I would highly recommend it to anyone who has a passion for innovation and a desire to explore new technology. The opportunities offered by virtual reality are vast and exciting, and I feel fortunate to have had the chance to be at the forefront of this rapidly evolving field.

1.2 Goal

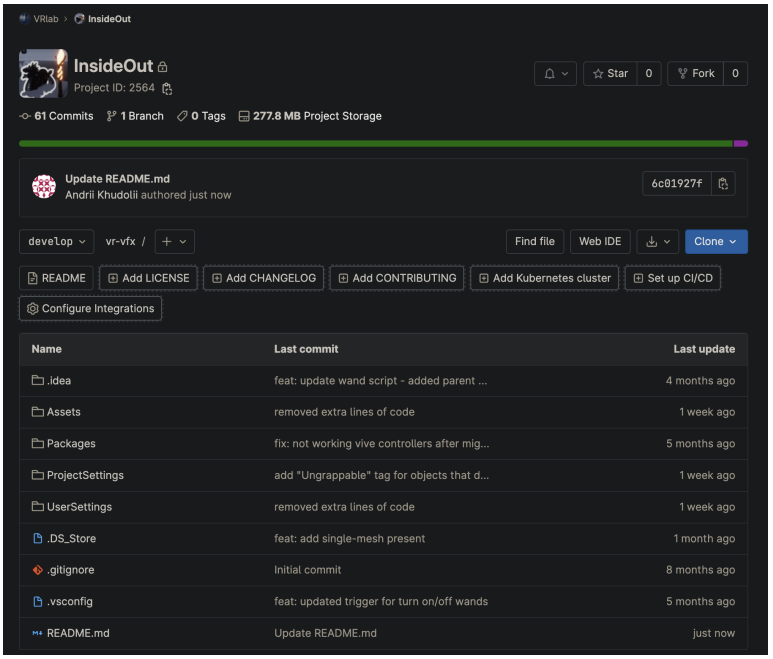
As a student working on this project, my main goal was to explore the potential of virtual reality (VR) technology for visual effects (VFX). By developing this project, I aimed to gain a deeper understanding of how VR could be used to create immersive and interactive VFX experiences. Additionally, I wanted to hone my skills in programming, 3D modeling, and animation, as well as improve my ability to work in a team and manage a complex project.

By creating a VR-based VFX experience, I hoped to advance the state of the art in VR technology and potentially inspire others to explore the possibilities of this emerging field. Finally, I may have been motivated by a desire to showcase my work to potential employers or collaborators, in order to establish myself as a skilled and knowledgeable developer in the field of VR and VFX.

2 Gitlab integration

To manage the project and keep track of the changes made to the code, I integrated Git into my workflow. I used my University's GitLab as the Git solution, which can be accessed using this link: https://gitlab.reutlingen-university.de/vrlab/vr-vfx/-/tree/develop. To interact with Git, I used

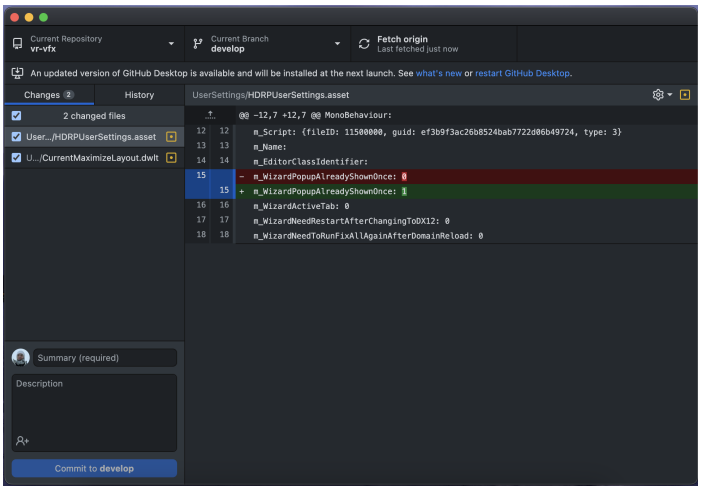
GitHub Desktop as the client solution.



To start working on the project, I first used the "git clone" command to create a local copy of the project on my computer. By running this command and specifying the GitLab URL, I was able to download the repository to my machine.

Whenever I made changes to the project, I used the "git add" command to stage the changes I had made for committing. This allowed me to select which files and changes should be committed to the repository.

After staging my changes, I used the "git commit" command to permanently save the changes to the repository. I often included a short message using the "-m" flag to describe the changes I had made.

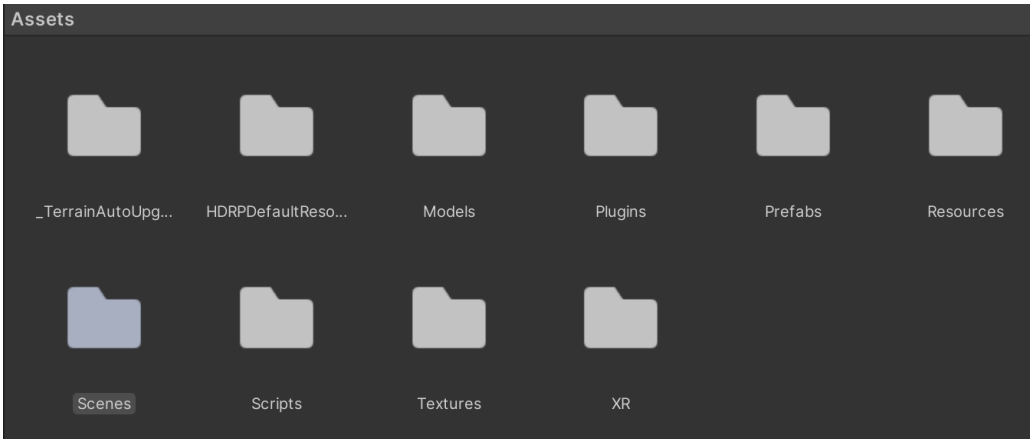


Whenever I needed to share my changes with other members of the team, I used the "git push" command to upload my changes to the GitLab repository. This allowed other members of the team to access and integrate my changes into their own local copies of the project.

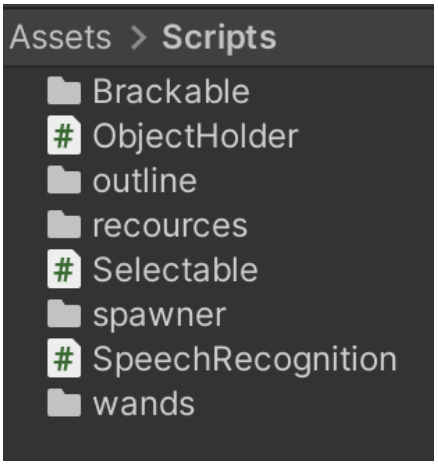
If updates were made on the other machine, I used the "git pull" command to download and integrate these changes into my local copy of the project.

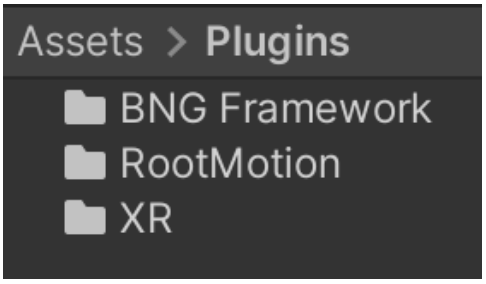
3 *Overall project setup*

I made sure to carefully set up the overall project structure and file management to keep everything organized and easy to access. All the project files were stored in related folders, with scripts located in the "scripts" folder, models in the "models" folder, and so on.



To further streamline the file management process, I used a consistent naming convention for all files, making it easier to quickly identify and locate the correct files when needed. I also made use of version control through Git to ensure that all team members were working with the most up-to-date versions of the files and scripts.



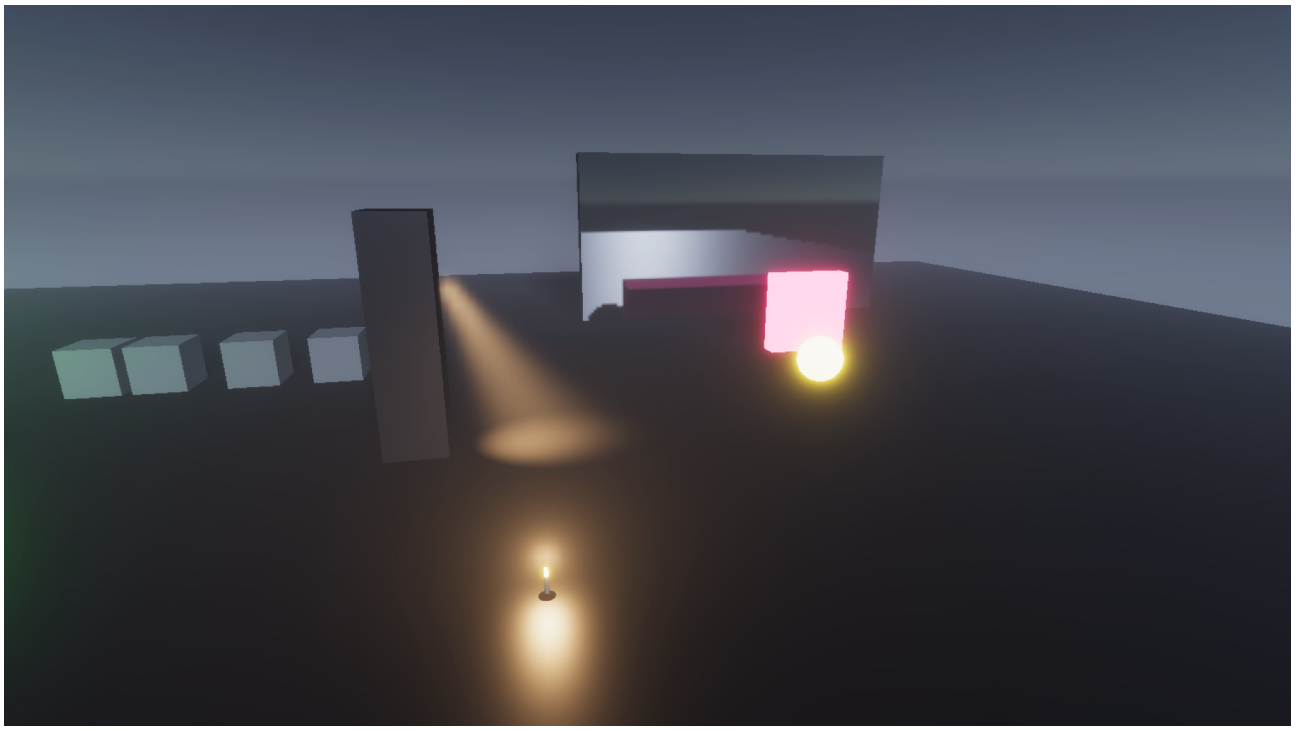


All plugins that were used in the project, are located in separate “Plugins” folder, to make it simple to use and don't litter the general assets structure.
Overall, I made sure that the project setup was well-organized, easy to navigate, and conducive to efficient development and collaboration. By taking the time to set up a robust file management system and development environment, I was able to work more efficiently and effectively, and

ultimately create a more successful project.

4 Graphics setup

I chose to use the High Definition Render Pipeline (HDRP) in Unity to achieve a high level of visual fidelity and realism in the virtual environment. HDRP is a powerful rendering pipeline that provides advanced graphics features, such as real-time ray tracing, advanced lighting and shading, and high-quality post-processing effects.

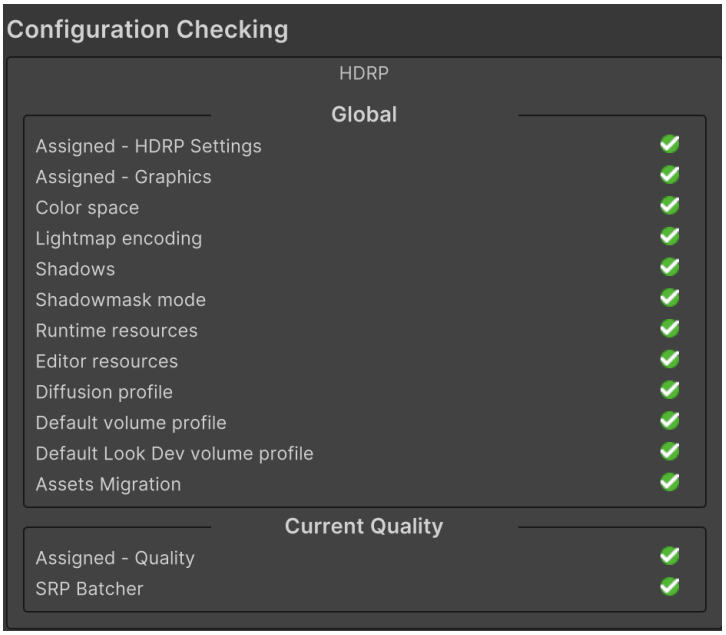


One of the main reasons why HDRP is an excellent choice for VR applications is that it is highly optimized for high-performance rendering, allowing for smooth and responsive VR experiences even on less powerful hardware. HDRP also provides a high degree of flexibility and customization, allowing developers to create unique and visually stunning virtual environments that are tailored to their specific needs.

In addition to its technical capabilities, HDRP also provides a range of tools and resources to help developers optimize and fine-tune their VR experiences for maximum impact. This includes support for a wide range of VR platforms and devices, as well as extensive documentation and resources to help developers get up and running quickly.

4.1 HDRP setup

Here are the steps I followed to set up a base HDRP project in Unity:



1. I opened Unity and created a new project, selecting "High Definition RP" as the template to create an HDRP project.
2. After the project was created, I navigated to the "Window" menu and selected "Package Manager". In the Package Manager, I searched for "HDRP" and made sure that the "High Definition RP" package was installed and up-to-date.
3. I then created a new Scene by going to "File" -> "New Scene", and added a new "Directional Light" to the Scene by going to "GameObject" -> "Light".
4. To set up the HDRP settings, I went to "Edit" -> "Project Settings" -> "Graphics", and set the "Scriptable Render Pipeline Settings" to "High Definition".
5. Next, I created a new folder named "Materials" in the "Assets" folder, which would contain all the Materials for my Scene.
6. I created a new Material by going to "Assets" -> "Create" -> "Material", named it "Default Material", and selected "HDRP/Lit" as the Shader.
7. Using the Inspector window, I adjusted the Material properties, such as color, metallic, and smoothness, to change the way it looked in the Scene.
8. Finally, I saved the Scene by going to "File" -> "Save Scene".

After following these steps, I was able to set up a basic HDRP project in Unity with a primary light source, HDRP settings, and a Material for rendering objects in the Scene. I could then start adding and customizing objects to create my own unique project.

4.2 HDRP pipeline profiles

In HDRP (High Definition Render Pipeline) graphics pipeline, there are several profiles that can be used to optimize rendering performance and adjust rendering settings for different use cases. Here are some of the most common profiles for HDRP:

Quality Settings: The Quality Settings profile in HDRP allows users to adjust the overall quality of the rendering. This profile includes settings such as texture quality, anti-aliasing, and shadow quality. The Quality Settings profile can be used to adjust the balance between visual quality and performance, depending on the specific use case.

Lighting Settings: The Lighting Settings profile in HDRP is used to adjust the lighting and shadows in a scene. This profile includes settings such as the intensity and color of the sun, the quality of the shadows, and the amount of global illumination. The Lighting Settings profile can be used to adjust the overall mood and atmosphere of a scene.

Volume Settings: The Volume Settings profile in HDRP is used to adjust the post-processing effects in a scene. This profile includes settings such as color grading, bloom, and ambient occlusion. The Volume Settings profile can be used to adjust the visual style of a scene and add additional depth and detail to the rendering.

Shader Settings: The Shader Settings profile in HDRP is used to adjust the behavior of individual shaders in a scene. This profile includes settings such as the level of detail for specific materials, the rendering mode for transparent objects, and the level of reflection for reflective surfaces. The Shader Settings profile can be used to adjust the visual quality and performance of specific objects in a scene.

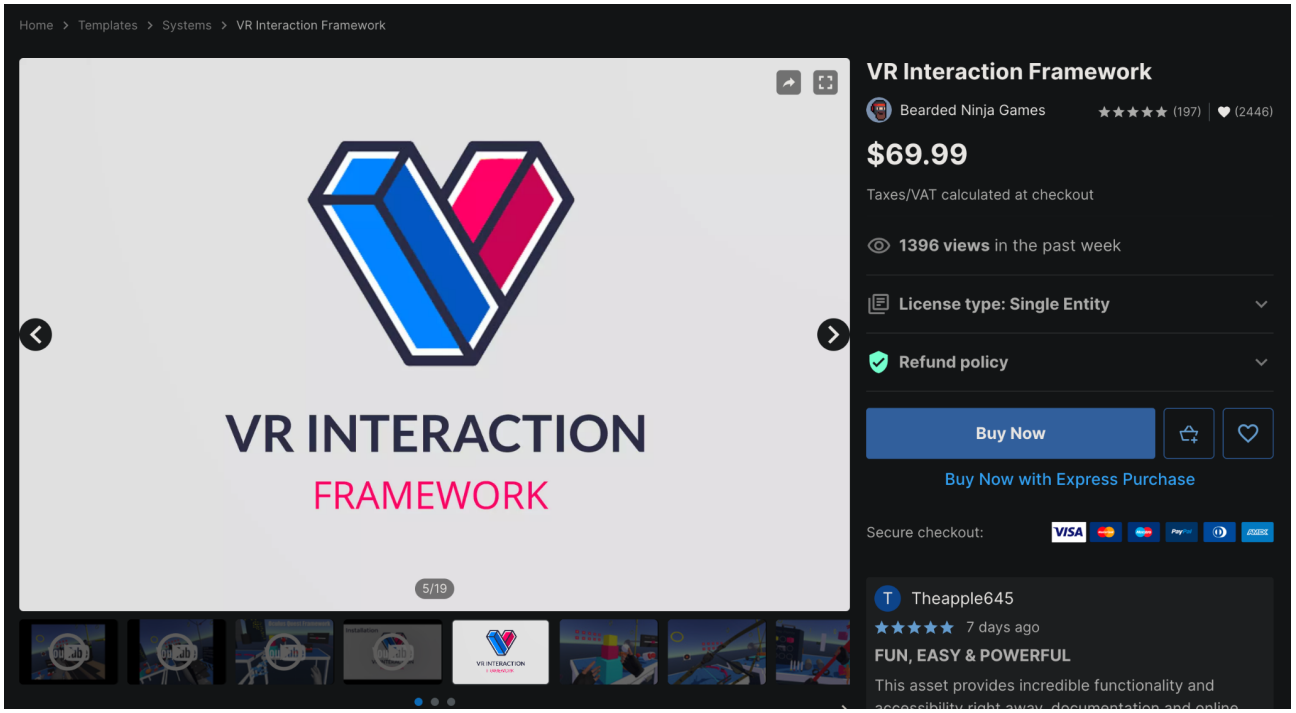
VFX Settings: The VFX Settings profile in HDRP is used to adjust the behavior of visual effects in a scene. This profile includes settings such as the particle system settings, the post-processing effects for particle systems, and the level of detail for specific VFX. The VFX Settings profile can be used to adjust the visual impact and performance of specific VFX in a scene.

Overall, the different profiles available in HDRP provide users with a powerful set of tools to optimize rendering performance and create customized visual styles for their projects. By adjusting the settings in these profiles, users can create stunning and immersive graphics that are tailored to their specific use cases.

5 VR interaction solution

5.1 VR interaction framework

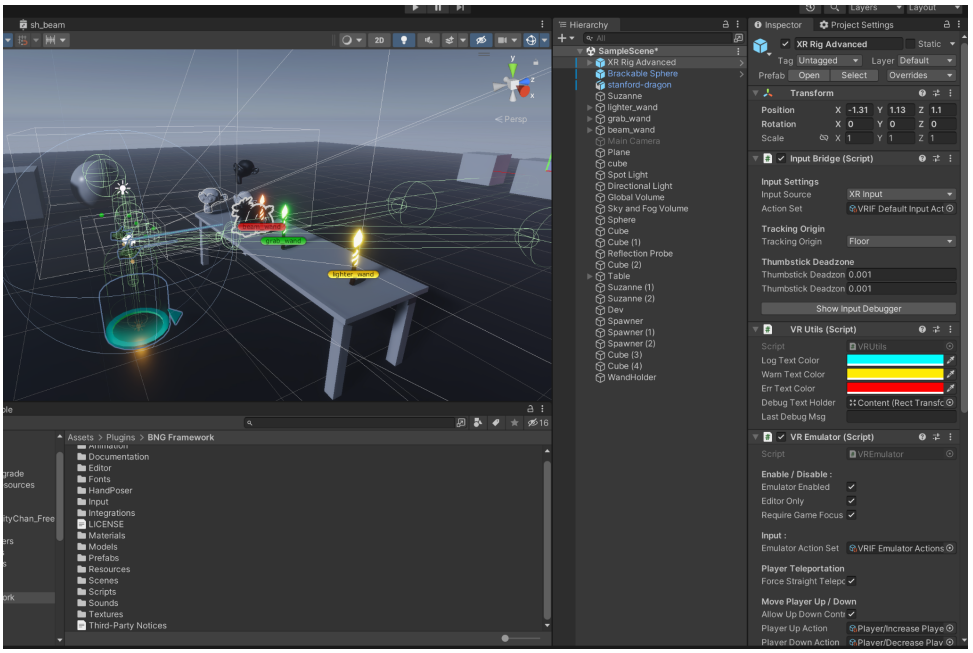
To implement the interactive features of my VR project, I made use of the VR Interaction Framework, which is a powerful and flexible tool for creating intuitive and engaging VR interactions. The package can be found at this link: https://assetstore.unity.com/packages/templates/systems/vr-interaction-framework-161066.



The VR Interaction Framework provides a range of pre-built components and tools that make it easy to implement a wide range of VR interactions, such as grabbing and manipulating objects, interacting with UI elements, and even navigating menus and other complex systems. The framework also provides a range of customizations and settings, allowing developers to fine-tune the behavior of these interactions to suit their specific needs.

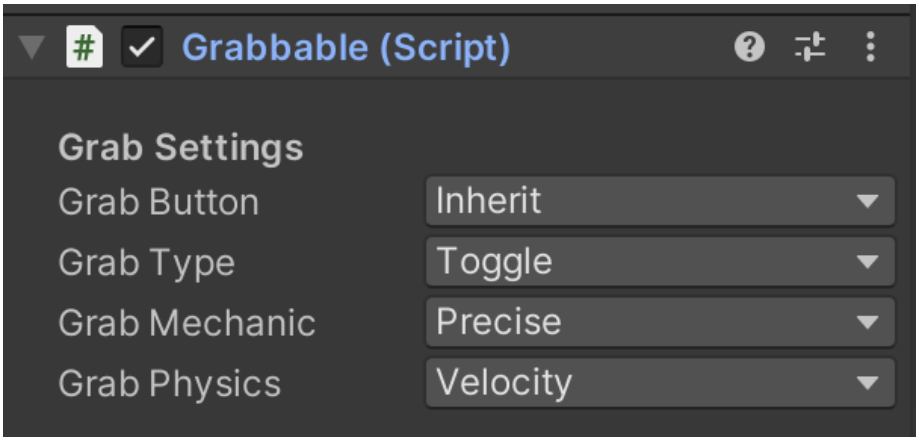
One of the main benefits of using the VR Interaction Framework is that it significantly reduces the amount of time and effort required to implement high-quality VR interactions. By providing pre-built components and tools, as well as extensive documentation and resources, the framework makes it easy for developers to quickly and efficiently implement engaging and immersive VR interactions.

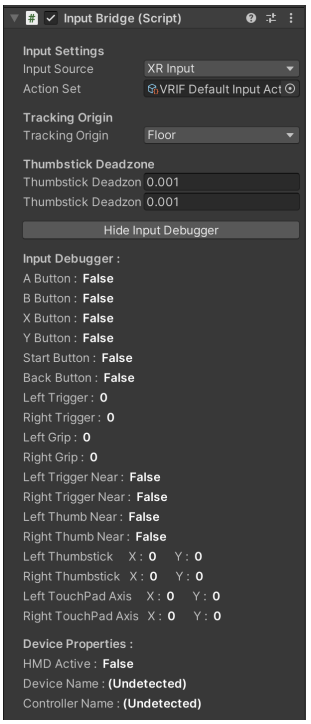
In addition to its ease of use, the VR Interaction Framework is also highly optimized for VR performance, providing a smooth and responsive user experience even on less powerful hardware. Overall, I found the VR Interaction Framework to be an excellent tool for implementing the interactive features of my VR project, and would highly recommend it to other developers working on VR applications.



For character input was used “XR Rig advanced”, which provides base controls and the ability to switch different models of the hands\character.

The main script is “Input bridge”, with which I can debug the input device. And for object grabbing, I had to add a “Grappable” script to that object.





5.2 *Custom wrappers\bridges for VR input*

To create a clear and flexible control for Wands, I had to write an abstract Wand bridge script that overrides GrabbableEvents class from VRIF and controls the bool value “isActive”.

Also, in the same class was added a parser for touchpad input controls. The problem was that VRIF has only separate getters for touchpad inputs: left and right. And for my controllers, it depends on which arm the wand is located, so I had to check in which arm is this wand and then check the proper input from the joystick.

```
public override void OnTriggerDown()
{
    _isActivated = !_isActivated;

    if (_isActivated) OnWandActivated();
    else OnWandDisabled();

    OnWandHolding(_isActivated);

    base.OnTriggerDown();
}

public override void OnRelease()
{
    if (disableWhenDropped)
    {
        OnWandDisabled();
        OnWandHolding(false);
    }

    base.OnRelease();
}
```

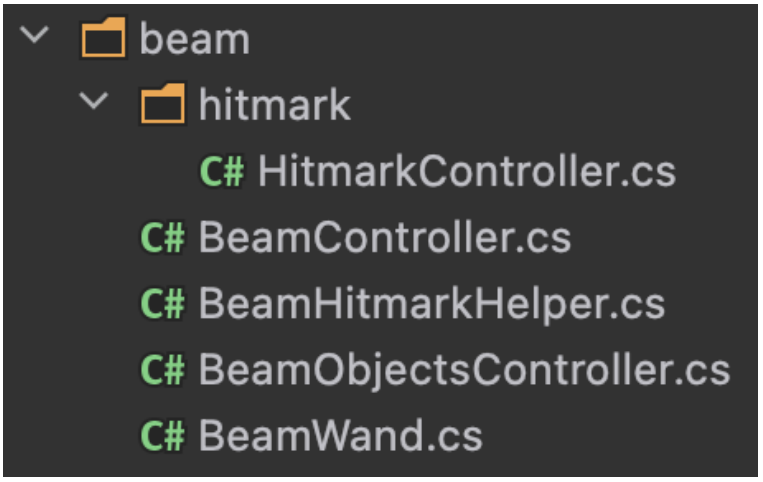
```
private void Update()
{
    TouchpadParser();
}

private void TouchpadParser()
{
    if(thisGrabber == null) return;

    Vector2 touchpadInput = Vector2.zero;

    if (thisGrabber.HandSide == ControllerHand.Left)
        touchpadInput = input.LeftThumbstickAxis;

    if (thisGrabber.HandSide == ControllerHand.Right)
        touchpadInput = input.RightThumbstickAxis;

    OnThumbstickAxis(touchpadInput);
}
```

Then, this Wand script is used as a Parent class. Generally, it’s being used for controlling the wand’s visuals, depending on its status. Also, in some cases, it could provide some methods or variables to another script, if necessary.

6 Coding

6.1 Wand controllers

6.1.1 Beam-Wand



Beam controller is split into multiple scripts, every each does that work. “BeamWand” controls the main visual on\off state of the wand. “BeamController” controls the vfx part of the beam, for the current state, it’s only changing the length of the beam. Then, “BeamObjectsController” - is the main physics controller.

```
public class BeamController : MonoBehaviour
{
    private BeamObjectsController _beamObjectsController;

    private bool _isObjectSelected;
    public VisualEffect vfx;


    private void Awake()
    {
        _beamObjectsController = gameObject.GetComponent<BeamObjectsController>();
    }

    private void Update()
    {
        vfx.SetFloat(name: "length", f: _beamObjectsController.targetObjectPointDistance * .5f);
    }
}
```

And here is the main part of the “BeamObjectsController” script. So what it does, is it creates and sends raycast beam forward from the head. If some object was hit, it saves length from the head to this point, so I can change the length of the beam to this value. Then, if the wand is activated, it sends a position to HitmarkController(later about its logic).
And if the object has a “Rigidbody” component in it, it adds a force for the object, that is Vetor from the wand’s head to the object. If velocity is higher than a limit, it doesn’t apply.

```
void Update()
{
    var ray = new Ray( origin: raycastPointer.transform.position,  direction: raycastPointer.transform.forward);

    if (Physics.Raycast(ray, out _rayOutput,  maxDistance: 1000))
    {
        targetObjectPointDistance = Vector3.Distance( a: _rayOutput.point,  b: gameObject.transform.position);

        if (!IsEnabled()) return;

        GameObject hittedObject = _rayOutput.transform.gameObject;
        Rigidbody objectRigidbody = hittedObject.GetComponent<Rigidbody>();

        ///
        _beamHitmarkHelper.CreateHitmarks( position: _rayOutput.point, _rayOutput.normal, hittedObject.transform);
        ///

        if (objectRigidbody == null) return;

        if(objectRigidbody.velocity.magnitude > MaxSpeed)
        {
            objectRigidbody.velocity = objectRigidbody.velocity.normalized * MaxSpeed;
        }

        objectRigidbody.AddForce((_rayOutput.point - gameObject.transform.position).normalized * .4f,
            ForceMode.Impulse);
    }
}
```

6.1.1.1 Heatmark filler

“BeamHitmarkHelper" was created for spawning and destroying hitmarks for a given position. It solves a problem when the beam is pointed at an object far away and the beam is moving. Logically, it should hit every few millimeters, but due to not precise inputs(controller), it creates a gap between hitmarks, and it should be filled.
So, actually, for this reason, this script has been created.

First thing first, a main method, that is called from “BeamObjectsController”. It creates a new hitmark, and if hitmarks were created before this one, it calls “FullFillHitmarks” which spawns additional marks between the two last. And if not - it just creates a new hitmark.

```
public void CreateHitmarks(Vector3 position, Vector3 normal, Transform hittedObject)
{
    SpawnHitmark(position, normal, hittedObject);

    if (hitmarks.Count > 1)
    {
        FullFillHitmarks(hitmarks[hitmarks.Count - 2], position, normal, hittedObject);
    }
    else
    {
        SpawnHitmark(position, normal, hittedObject);
    }
}
```

Here in the “FullFillHitmarks” method, it checks if the object, for which they should be created hitmark, is the same object as for the previous hitmark. It needs to exclude situations when

hitmarks are spawned in the air, between different objects.
Then, it calculates a count of the hitmarks, that should be spawned, depending on the distance to the last hitmark.
And last, it spawns hitmarks N times, which was calculated before.

```
private void FullFillHitmarks(GameObject previousHitmark, Vector3 endPosition, Vector3 normal, Transform hittedObject)
{
    if (previousHitmark.transform.parent != hittedObject) return;

    float count = Vector3.Distance(a: previousHitmark.transform.position, b: endPosition) * 10;

    for (float i = 0; i < 1; i += 1 / count)
    {
        SpawnHitmark(position: Vector3.Lerp(a: previousHitmark.transform.position, b: endPosition, i), normal, hittedObject);
    }
}
```

And finally, the “SpawnHitmark” method. It creates a new object of a given hitmark instance. Also, it gives random rotation, and random scale in given limits. And then it accinges it to the parent object.
In the end, it adds this hitmark to the list of all hitmarks.

```
private void SpawnHitmark(Vector3 position, Vector3 normal, Transform hittedObject)
{
    var hitMark:GameObject = Instantiate(Bullet_Mark, position, Quaternion.LookRotation(normal));
    hitMark.transform.Rotate(eulers: Vector3.up * 180);
    hitMark.transform.Rotate(eulers: Vector3.forward * Random.Range(.0f, 180.0f));

    var scaleValue:float = Random.Range(-.2f, .1f);
    hitMark.transform.localScale += new Vector3(x: scaleValue, y: scaleValue, z: 0);

    hitMark.GetComponent<HitmarkController>().beamHitmarkHelper = this;
    hitMark.transform.SetParent(hittedObject);

    hitmarks.Add(hitMark);
}
```

Each hitmark has a “HitmarkController” with a unique ID. This script is used for destroying hitmark after N time. And when the object is destroyed, it should be removed from the list of all hitmarks in the previous script.

```
public BeamHitmarkHelper beamHitmarkHelper;

[HideInInspector] public String id = Guid.NewGuid().ToString();

void Start()
{
      Destroy(this.gameObject, t: 1.5f);
}

private void OnDestroy()
{
    beamHitmarkHelper.RemoveHitmarkFromList(gameObject);
}
```

For that, it calls the “RemoveHitmarkFromList” method from the main “BeatHitmarkHelper” script. This method searches for a hitmark with the same id as given and then removes it from the list.

```
public void RemoveHitmarkFromList(GameObject hitmark)
{
    for (var i = 0; i < hitmarks.Count; i++)
    {
        if (hitmarks[i].GetComponent<HitmarkController>().id == hitmark.GetComponent<HitmarkController>().id)
        {
            hitmarks.RemoveAt(i);
            break;
        }
    }
}
```

6.1.2 Grab-Wand

Grab-wand contains 3 scripts for controlling different parts of it. The first one - is the “GrabWand” script, which is a child of the “Wand” script, and controls the visuals of the wand and also overrides thumbstick input to callback “OnThumbstickAxisCallback”.

Then, 2 other scripts are directly controlling the grabbing. The First one is “ArcController”, which controls visuals of the grabbing beam, and the second one controls directly the physics and logic of it.

First, let’s take a look to input variables. I can control a beam with 3 vectors: start position, end position, and 2 controls between them. Then, it’s pointerBeam, that controls the semi-transparent pointer to the object. It helps a player to aim at the objects. And “startTarfget” is a starting point, from which the beam starts.

```
[Header("Beam Parameters")]
public Transform startBeam;
public Transform beam2;
public Transform beam3;
public Transform endBeam;
public Transform pointerBeam;

public Transform startTarget;
```

Then, in the “Update” method I’m setting a starting position to the first point of the beam. Also, I’m updating the length of the “pointer beam” to reach the object to which it points, and if not any - set length to zero. Then it checks if some object is selected, and if so, it should activate the beam, set an endpoint(end middle points too) for it, and if the object is not selected - disable the beam.

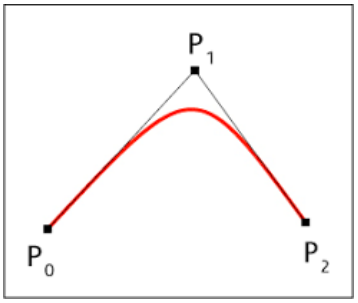
```
private void Update()
{
    startBeam.position = startTarget.position;

    var rayOutput:RaycastHit = _grabbableController.RayOutput;
    _isObjectSelected = _grabbableController.selectedObject != null;

    UpdateHoverBeam(Vector3.Distance(a: rayOutput.point, b: transform.position));

    if (_isObjectSelected)
    {
        UpdateHoverBeam(distance: .0f);
        SetEndPosition(_grabbableController.selectedObject.transform.position);
        gameObject.GetComponent<GrabWand>().beam.SetActive(true);
    }
    else
    {
        gameObject.GetComponent<GrabWand>().beam.SetActive(false);
    }
}
```

In the “SetEndPotision” method, it calculates the end position of the beam, by getting the position of the selected object. And also it calculates the position of the 2 control points in the middle, by using the bezier curve formula.

```
private void SetEndPosition(Vector3 position)
{
    endBeam.position = position;

    var wandPosition:Vector3 = transform.position;
    var targetPointerPosition:Vector3 = _grabbableController.targetPointer.transform.position;
    beam2.position = _CalculateQuadraticBezierPoint(t: .3f, p0: wandPosition, p1: targetPointerPosition, p2: position);
    beam3.position = _CalculateQuadraticBezierPoint(t: .6f, p0: wandPosition, p1: targetPointerPosition, p2: position);
}
```

For the bezier formula, I’m using a separate method called “_CalculateQuadraticBezierPoint”, in which I send 4 values:

1. t - how far this point should be from start. It is a float value in limits .0 - 1.0.
2. p0 - start position of the curve, I’m using the position of the wand for it.
3. p1 - position, in which wand points.
4. p2 - current position of the selected object.




```
private Vector3 _CalculateQuadraticBezierPoint(float t, Vector3 p0, Vector3 p1, Vector3 p2)
{
    float u = 1 - t;
    float tt = t * t;
    float uu = u * u;
    Vector3 p = uu * p0;
    p += 2 * u * t * p1;
    p += tt * p2;

    return p;
}
```

And then it's the actual, main "GrabbableController", that controls all logic and physics, related to the object's moving.
So, first thing first, it needs to send a raycast forward to check if the wand is pointing to any object. If so, "OnObjectCollided" is called, and if not, and if a player is not holding the trigger, it releases the object.

```
var ray = new Ray( origin: raycastPointer.transform.position,  direction: raycastPointer.transform.forward);

if (Physics.Raycast(ray, out RayOutput,  maxDistance: 1000))
{
    OnObjectCollided(RayOutput.transform.gameObject);
}
else
{
    if (!IsGrabbing())
    {
        ReleaseLastObject();
        _lastHittedObject = null;
        selectedObject = null;
    }
}
```

In the "OnObjectCollided" method it checks some cases, that should be excluded before going to object selection. If it passed well, the "ControlSelect" method is called - it enables an outline effect on the object. I just need to pass there status of the outline: Selected, Hover, Unselected.
And the last step is checking if the player is holding the trigger or not. If so, the object's physics is disabled and the outline updated, and if not, physics should be turned back to the previous parameters, and the outline should be disabled.

```
// GameObject here is always != null
private void OnObjectCollided(GameObject hittedObject)
{
    if (!IsGrabbing() && selectedObject != null) selectedObject = null;

    if (selectedObject != null) return;


    // The wand is pointed to the new object
    ReleaseLastObject();

    if (hittedObject.GetComponent<Rigidbody>() == null || hittedObject.CompareTag("Ungrappable"))
    {
        _lastHittedObject = null;
        return;
    }

    ControlSelect(hittedObject, SelectableType.Hover);


    if (IsGrabbing()) OnWandReadyToGrab(hittedObject);
    else UpdateObjectPhysicsParameters(hittedObject,  isPhysics: true);

    _lastHittedObject = hittedObject;
}
```

Also, it's possible to move the selected object closer\further, by using the joystick input on controllers. to achieve it, an axis callback is assigned to the "OnThumbstickAxisCallback" method, which changes the object's position.

```
private void Awake()
{
    GetComponent<GrabWand>().OnThumbstickAxisCallback = OnThumbstickAxisCallback;
}
```

```
private void OnThumbstickAxisCallback(Vector2 input)
{
    SetTargetObjectPointDistance(position: _targetObjectPointDistance + input.y * .5f);
}
```

6.1.2.1 Selectable controller

To manipulate the object's outline, was created a "Selectable" script, that is located on the object with an outline shader on it.

Main controls are provided only by one method "UpdateStatus", which accepts an enum of statuses and then calls an appropriate method. But, before manipulating the object's outline settings, it should check if it's any outline on it, and if not - create one. For this "_checkOutlineScript" method is called.

```
public void UpdateStatus(SelectableType type)
{
    _checkOutlineScript();

    switch(type)
    {
        case SelectableType.Hover:
            _HoverObject();
            break;
        case SelectableType.Selected:
            _SelectObject();
            break;
        case SelectableType.Unselected:
            _UnselectObject();
            break;
        default:
            _UnselectObject();
            break;
    }
}
```

6.2 *Speech controller*

In the first stages of this project, it was an idea to control wands with micro input. For that was used default unity's "KeywordRecognizer" and "Dictionary" classes. First, I had to add keywords to the dictionary and assign a callback for them.
Then it's only enough to assign this dictionary to the keyword recognizer. And that's it, when a player says something in the microphone, it detects it and prints it to the console.

```
private KeywordRecognizer keywordRecognizer;
private Dictionary<string, Action>  actions = new Dictionary<string, Action>();

// Start is called before the first frame update
void Start()
{
    actions.Add("forward", OnAction);
    actions.Add("left", OnAction);
    actions.Add("right", OnAction);
    actions.Add("back", OnAction);
    actions.Add("pick up", OnAction);
    actions.Add("put down", OnAction);

    keywordRecognizer = new KeywordRecognizer(keywords: actions.Keys.ToArray());
    keywordRecognizer.OnPhraseRecognized += OnRecognizedSpeech;
    keywordRecognizer.Start();
}

private void OnRecognizedSpeech(PhraseRecognizedEventArgs speech)
{
    Debug.Log(speech.text);
    actions[speech.text].Invoke();
}
```

6.3 *Brackable controller*

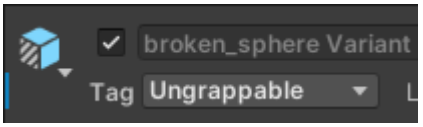
It's possible to destroy some objects in this demo game. Destruction was made in a very simple way - just replacing the actual object with his destructed variation. It worked well but was some problem when the object was moving and after that destroyed - velocity of the destroyed variation was zero, so it needs to be transferred from the original object to its destroyed variant. That's what the "AddVelocityToDestroyedObject" method is doing - adds a velocity for every child of destroyed object variation.

```
public void Broke()
{
    var spawnedObject = Instantiate(brokenObject, transform.position, transform.rotation);
    spawnedObject.AddComponent<BrockenController>();

    AddVelocityToDestroyedObject(spawnedObject);

    Destroy(gameObject);
}
```

Due to optimization, it's good to remove these destroyed parts of the object. I thought that it would be too boring just to destroy them, I wanted to scale them out to zero sizes and only then destroy them. For that purpose "BrockenController" was created. It assigns every part that should be scaled out and destroyed after some time.

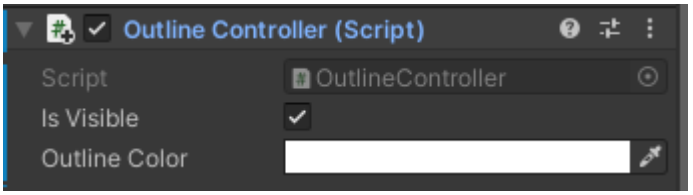
```
private float _timer = 5f;

private void Update()
{
    _timer -= Time.deltaTime;

    if (_timer <= 5f)
    {
        float scale = Mathf.Lerp(a: 0, b: 1, t: _timer / 2f);
        ScaleObject(transform, scale);
    }

    if (_timer <= 0f)
        Destroy(gameObject);

}

private void ScaleObject(Transform obj, float scale)
{
    foreach (Transform child in obj)
    {
        if (child.localScale.x < .01f || child.localScale.y < .01f || child.localScale.z < .01f)
            return;

        child.localScale *= scale;
    }
}
```

Also, for safety purposes, these pieces were tagged with the "Ungrappable" name to prevent them from being grabbed by Grab-wand.



6.4 Outline controller

For proper work of outline shaders it's not enough to have only materials with proper shaders. That's why "OutlineController" was created. It's used for controlling and animating the size and color of the outline.



By using this script it's possible to change the isVisisble parameter and also the color value. And this controller will change it with smooth animation.
This script is working in Edit mode too. Also, it's preventing of multiple instances of itself on one object. All these are possible by using attributes "DisallowMultipleComponent" and "ExecuteInEditMode".

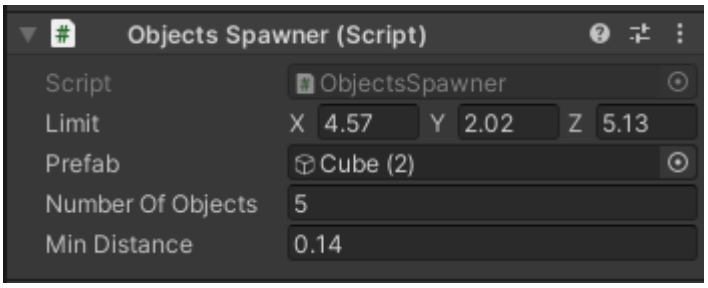
To achieve smooth animations I've used coroutines. For example. this coroutine is changing the width of the outline. It checks automatically, how much it should increase the width in one period of time. It gets the max outline width and speed of the animation from the global Dimens class, about which I'll tell later.

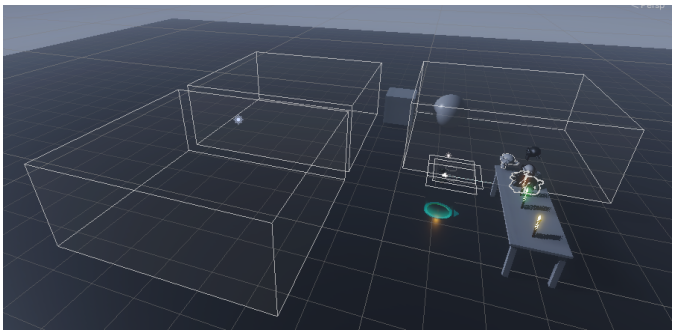
```
private IEnumerator UpdateState(bool isActive)
{
    const float duration = Dimens.FastDuration;
    var timer = 0f;

    while (timer < duration)
    {
        var startPos :float = GetOutlineValue();
        float step = Dimens.OutlineWidth * Time.deltaTime / duration;

        SetOutlineValue(isActive ? startPos + step : startPos - step);

        timer += Time.deltaTime;
        yield return null;
    }
}
```

To make it easier to use this outline, I made that is only enough to add the "OutlineController" script to the object. Then, this script will add "OutlineSettings", that then will already add outline materials. So, in general, it's only enough to add or remove only the "OutlineSettings" script and not every script\material.

6.5 Spawner

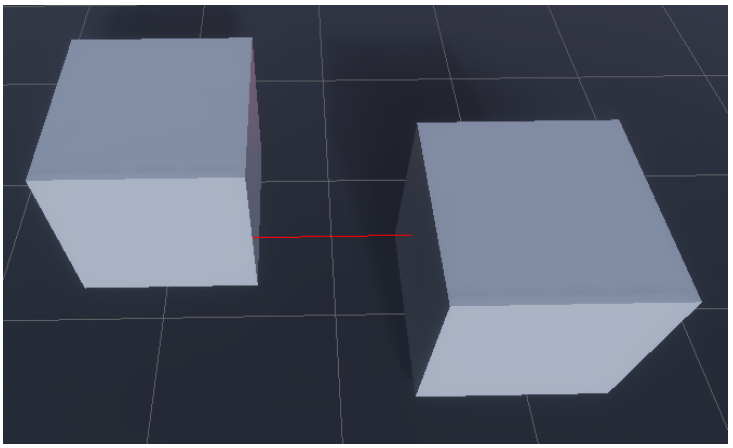
I needed to dynamically spawn objects for testing all mechanics. So, for this purpose was created Spawner script, which spawns objects in given constraints with a given number of objects and distance between them.
Also, it was visually displayed as dev gizmos.





6.5.1 Object distance

While creating a script for spawning objects, the hardest part was not to spawn objects inside each other. For that purpose was created a special script "ObjectDistance", which can be assigned to any object and he can draw the closest line between the given two objects and print the distance value to the console.
Also, it provides two static methods for external use in any script.



The first one is "GetClosestDistance" which accepts two transforms and provides the closest distance between them. To do that, it needed to get the Collider component of each object, and by using a collider. ClosestPoint method determines the closest point to another object.

```
public static float GetClosestDistance(Transform position1, Transform position2)
{
    Collider collider1 = position1.GetComponent<Collider>();
    Collider collider2 = position2.GetComponent<Collider>();

    Vector3 closestPoint1 = collider1.ClosestPoint(collider2.transform.position);
    Vector3 closestPoint2 = collider2.ClosestPoint(closestPoint1);
    // Calculate the distance between the closest points of the colliders
    float distance = Vector3.Distance(a: closestPoint1, b: closestPoint2);

    return distance;
}
```

The second method provides information about if these objects are inside each other or not, by using nearly the same logic as before.

```
public static bool IsInside(Transform position1, Transform position2)
{
    Collider collider1 = position1.GetComponent<Collider>();
    Collider collider2 = position2.GetComponent<Collider>();

    return collider1.bounds.Intersects(collider2.bounds);
}
```

6.5.2 Spawner controller

And finally, the “ObjectsSpawner” script. It has a method that spawns a given amount of objects. So, it creates a loop with a given number of iterations, in which, each cycle, spawns a new object, randomized its rotation(only the y and z axis), and then seeks a proper place to place it. If it is not such a place - it destroys it.

```
private void SpawnObjects()
{
    _usedTransforms = new List<Transform>();
    _usedPositions = new List<Vector3>();

    Vector3 prefabSize = prefab.GetComponent<Renderer>().bounds.size;

    for (int i = 0; i < numberOfObjects; i++)
    {
        Quaternion rotation = Quaternion.Euler(x: -90, y: Random.Range(0, 360), z: Random.Range(0, 360));

        var newObject = Instantiate(prefab, GetRandomPosition(prefabSize), rotation);

        if (!SetUniquePosition(newObject.transform, prefabSize))
            DestroyImmediate(newObject);

        _usedTransforms.Add(newObject.transform);
    }
}
```

So, what this method does is that it seeks a proper position when it can't find a good one. To check if it is a good position, the "IsTooClose" method was created. This method checks if an object is too close to any objects that were spawned before. If it is too close, it returns true; otherwise, it returns false. The method uses ObjecDistance.GetClosestDistance to check the distance between objects.

```
private bool IsTooClose(Transform position)
{
    float limitedDistance = Math.Clamp(minDistance, .01f, Single.PositiveInfinity);
    foreach (Transform usedPosition in _usedTransforms)
    {
        if (ObjectDistance.GetClosestDistance(usedPosition, position) < limitedDistance)
            return true;
    }

    return false;
}
```

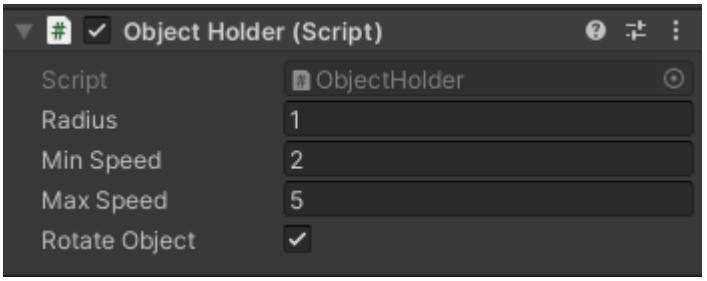
And random position searches like that, in “GetRandomPositionMethod”. I couldn’t just use given constraints, because I wanted to have objects exactly inside it and don’t have any extruded parts outside.

```
private Vector3 GetRandomPosition(Vector3 scale)
{
    Vector3 position;

    do
    {
        position = transform.position + new Vector3(
            x: Random.Range(-limit.x / 2 + scale.x / 2, limit.x / 2 - scale.x / 2),
            y: Random.Range(-limit.y / 2 + scale.y, limit.y / 2 - scale.y / 2),
            z: Random.Range(-limit.z / 2 + scale.z / 2, limit.z / 2 - scale.z / 2)
        );
    } while (_usedPositions.Contains(position));

    _usedPositions.Add(position);

    return position;
}
```

6.6 Object holder

I need some object which will hold any other rigidbody object, for that purpose were created the “ObjectHolder” script.

It’s possible to set the radius in which it will seek objects, the speed of moving towards the center, and if an object should rotate when reaches the center.



What it does is seeks for the first rigdibody in a given radius and if this exists - moves to the center and then rotates it.

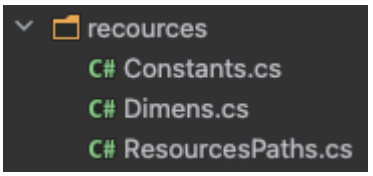
```
private void HandleHeldObject()
{
    float distance = Vector3.Distance(a: transform.position, b: heldObject.transform.position);

    if (distance > radius || heldObject.transform.parent != null)
    {
        DropHeldObject();
    }
    else
    {
        MoveHeldObject(distance);
        RotateHeldObject(distance);
    }
}
```

The most complicated part was to check all cases when object should or should not be held by this script. That's why it's so many if statements here and in other methods.

```
private void FindClosestRigidbody(Collider[] colliders)
{
    Rigidbody closestRigidbody = null;
    float minDistance = float.MaxValue;

    foreach (Collider collider in colliders)
    {
        Rigidbody rb = collider.GetComponent<Rigidbody>();

        if (rb != null && rb.transform.parent == null)
        {
            float distance = Vector3.Distance(a: transform.position, b: rb.transform.position);

            if (distance < minDistance)
            {
                minDistance = distance;
                closestRigidbody = rb;
            }
        }
    }

    if (closestRigidbody != null)
    {
        HoldObject(closestRigidbody, minDistance);
    }
}
```

6.7 Global resources

At the latest for the coding part of this documentation are global resources for scripts. It is good practice to extract some constant values to a special file to have easy and fast access to them.



Just for some constant values, such as strings and names of variables, the “Constants” script was created.

```
public class Constants
{
    public const String OutlineMaxWidthName = "_2DOutlineWidthMax";
    public const String OutlineWidthName = "_2DOutlineWidth";
    public const String OutlineColorName = "_OutlineColor";

    public const String ShaderZTestName = "_ZTest";

    public const String PrefabOutlineMaskName = "Outline Mask";
    public const String PrefabOutlineFillName = "Outline Fill";
}
```

Dimens script is more for variables that can be adjusted in the future and should be adjusted with additional logic in the code. “Shader.SetGlobalFloat” just sets given changed max outline width to all outline shader instances that exist in the project.

```
[DisallowMultipleComponent]
[ExecuteInEditMode]
public class Dimens : MonoBehaviour
{
    [Range(0f, 10f)] public const float OutlineWidth = 5f;

    [Range(0f, 3f)] public const float FastDuration = 0.25f;

    public static readonly Color OutlineSelectedColor = Color.white;
    public static readonly Color OutlineHoverColor = Color.gray;

    void Awake()
    {
        UpdateShadersProperties();
    }
    void OnEnable()
    {
        UpdateShadersProperties();
    }


    void Update()
    {
        UpdateShadersProperties();

    }

    private void UpdateShadersProperties()
    {
        Shader.SetGlobalFloat(Constants.OutlineMaxWidthName, value: OutlineWidth);
    }
}
```

And “ResourcePaths” is for constant strings that contain paths for maybe some files, materials, etc…

```
public class ResourcesPaths
{
    public const String OutlineMaskShader = @"Shaders/outline/variant1/2DOutlineMask";
    public const String OutlineShader = @"Shaders/outline/variant1/2DOutline";
}
```

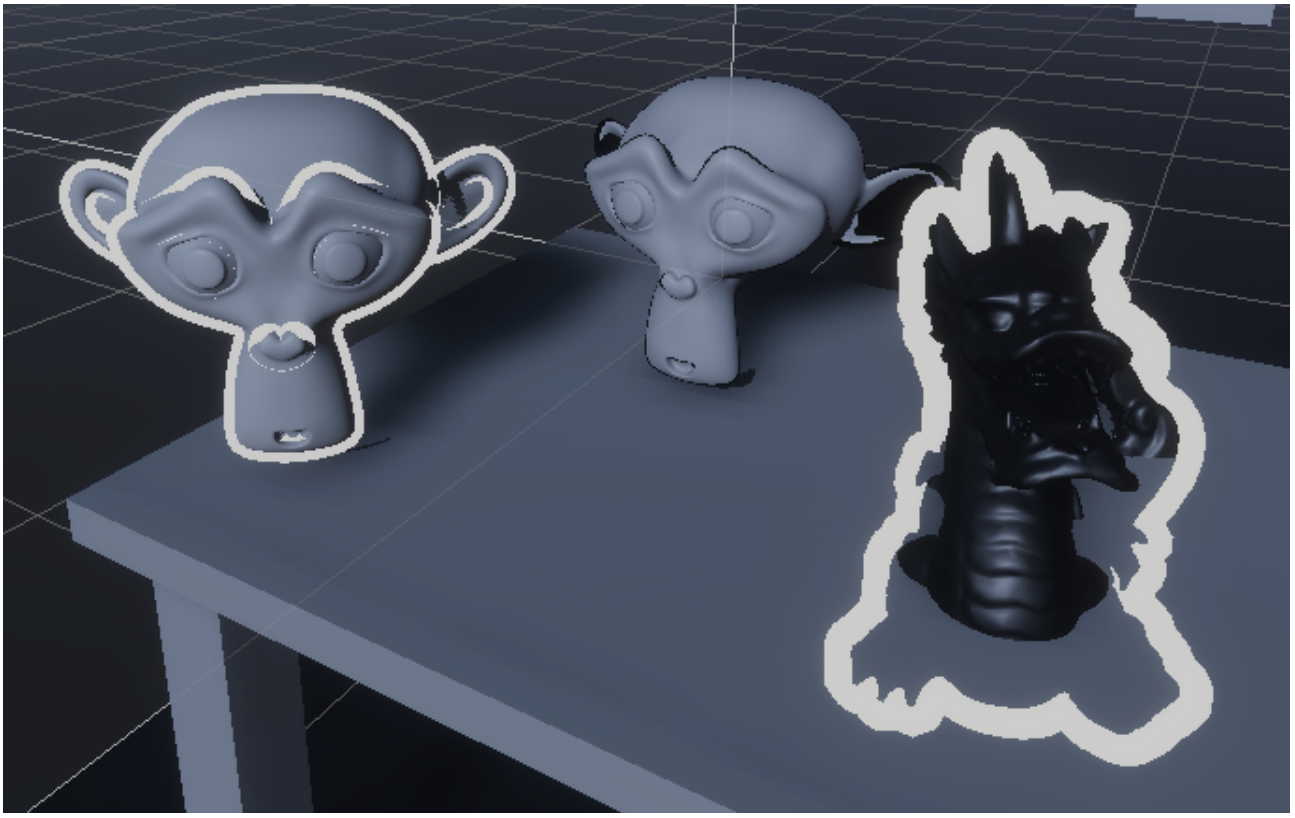
7 *Visual effects*

The other important part that I wanted to explore while working on this project, was creating beautiful visual effects. As with Visual Graph and also with code-written shaders. It worked out pretty well, I gained a lot in writing code-based shaders, but I had no experience, so it was a long journey.

While creating shaders, I’ve noticed that shaders don’t work well with VR rendering. The most goal-breaking problem was that shaders were rendered only in one headset’s screen(left or right), but they should do that in both. For that, I had to enable additional settings in shaders, which is shown in the next image.

```
/// calculates and sets the built-in unity_StereoEyeIndex and unity_InstanceID shader variables
/// to the correct values based on which eye the GPU is currently rendering.
UNITY_SETUP_INSTANCE_ID(v);

/// initializes all v2f values to 0.
UNITY_INITIALIZE_OUTPUT(v2f, o);

/// tells the GPU which eye in the texture array it should render to, based on the value of unity_StereoEyeIndex.
/// This macro also transfers the value of unity_StereoEyeIndex from the vertex shader
/// so that it will be accessible in the fragment shader only if UNITY_SETUP_STEREO_EYE_INDEX_POST_VERTEX
/// is called in the fragment shader frag method.
UNITY_INITIALIZE_VERTEX_OUTPUT_STEREO(o);

/// making shader works even on the layers where unity's fog is enabled
// UNITY_TRANSFER_FOG(o, o.vertex);
```

7.1 Outline creation



While learning how to program outline shader, were created 3 different variants of this shader. But each of them was using one method - Vertex Extrusion.
The Vertex Extrusion method for creating an outline shader involves duplicating a mesh, scaling it slightly, and rendering it behind the original mesh to create the illusion of an outline.

The main outline(on the right), was made using two shaders - one is creating the same white mech behind the actual object and another shader masks out all white vertices that intersect with the original model.

```
const float3 world_scale = float3(
    // scale x axis
    length(float3(unity_ObjectToWorld[0].x, unity_ObjectToWorld[1].x, unity_ObjectToWorld[2].x)),
    // scale y axis
    length(float3(unity_ObjectToWorld[0].y, unity_ObjectToWorld[1].y, unity_ObjectToWorld[2].y)),
    // scale z axis
    length(float3(unity_ObjectToWorld[0].z, unity_ObjectToWorld[1].z, unity_ObjectToWorld[2].z))
);

/// dividing by object's scale to make outline width the same on every object
v.vertex.xyz += v.normal * clamp(_2DOutlineWidth, 0, _2DOutlineWidthMax) * .005f / world_scale;

o.pos = UnityObjectToClipPos(v.vertex);
```

Another challenging part was to automatically create ± the same width of outline for any scale of an object. So for that reason, the extrusion formula needs to be added object's scale parameter.
And then the mask shader just needs to cull off extra parts of the outline.

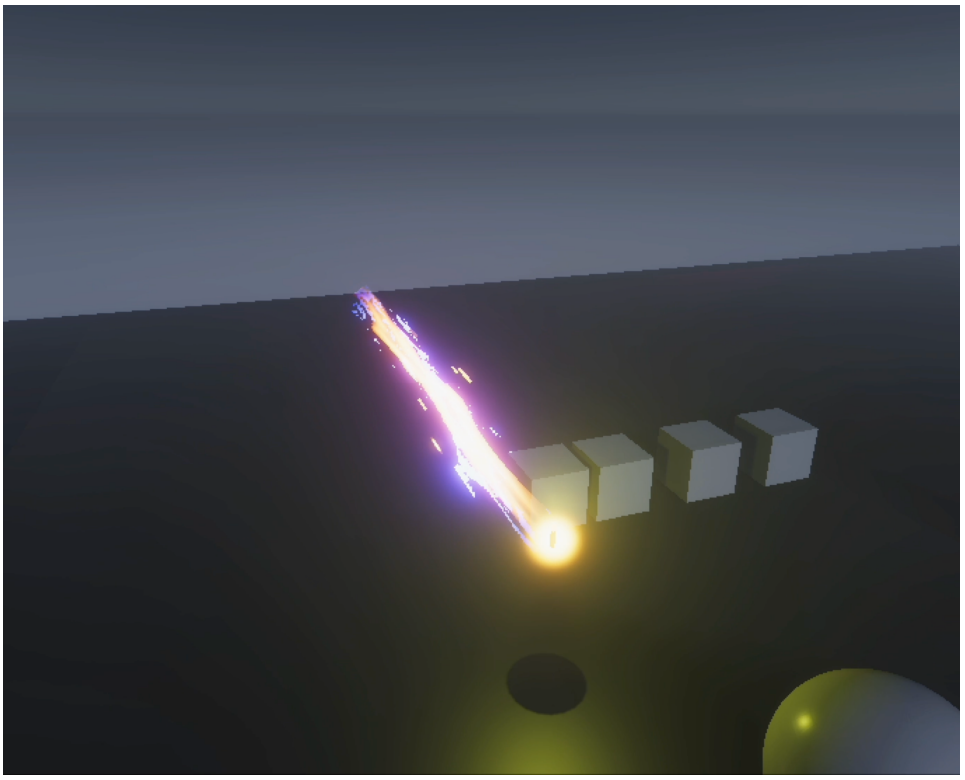
```
Properties
{
    [Enum(UnityEngine.Rendering.CompareFunction)] _ZTest("ZTest", Float) = 0
}

SubShader
{
    Tags
    {
        "RenderType" = "Transparent"
        "Queue" = "Transparent+1"
    }
    LOD 100

    Pass
    {
        Name "Mask"
        Cull Off
        ZTest [_ZTest]
        ZWrite Off
        ColorMask 0

        Stencil
        {
            Ref 1
            Pass Replace
        }

    }
}
```

The outline of the object on the left is actually the same outline but without a mask layer. So it's possible to see the outline of the object too.

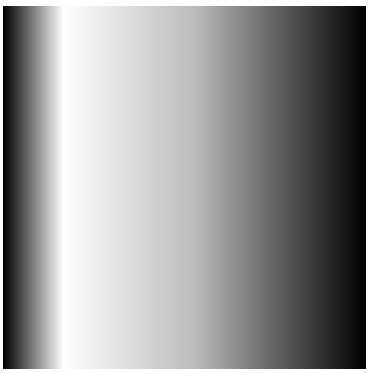
And the last variation of this shader is one, that creates a plane, and renders it only on the object. Basically, it just renders it in a projection coordinate system, so it depends on how close the camera is to the object.

```
v2f vert(appdata v)
{
    v2f o;
    o.vertex = UnityObjectToClipPos(v.vertex);

    float3 norm = normalize(mul((float3x3)UNITY_MATRIX_IT_MV, v.normal));
    float2 offset = TransformViewToProjection(norm.xy);

    o.vertex.xy += offset * _Outline * v.color.r;
    o.vertex.z += v.color.g * v.color.r * 0.00015;

    //o.uv = TRANSFORM_TEX(v.uv, _MainTex);
    //UNITY_TRANSFER_FOG(o,o.vertex);
    return o;
}
```

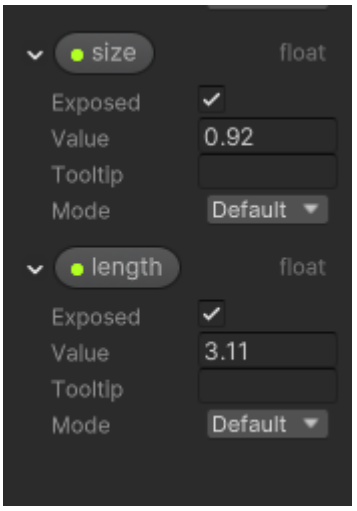
7.2 Heat-beam creation



This visual effect was created by using Unity's Shader Graph. Basically, it has 3 different layers in it. That was created to make this effect deep enough and visually stunning. These layers are: the base, the yellow-colored layer, the same but a little bigger and darker to add more depth to it, and last one is the small particles layer.
The main layer was created with custom textures, for that purpose, the graphic app was drawn a white layered shape of the beam(on the left) and the mask layer(on the right), which shows where the beam should be most visible and where it should fade out.

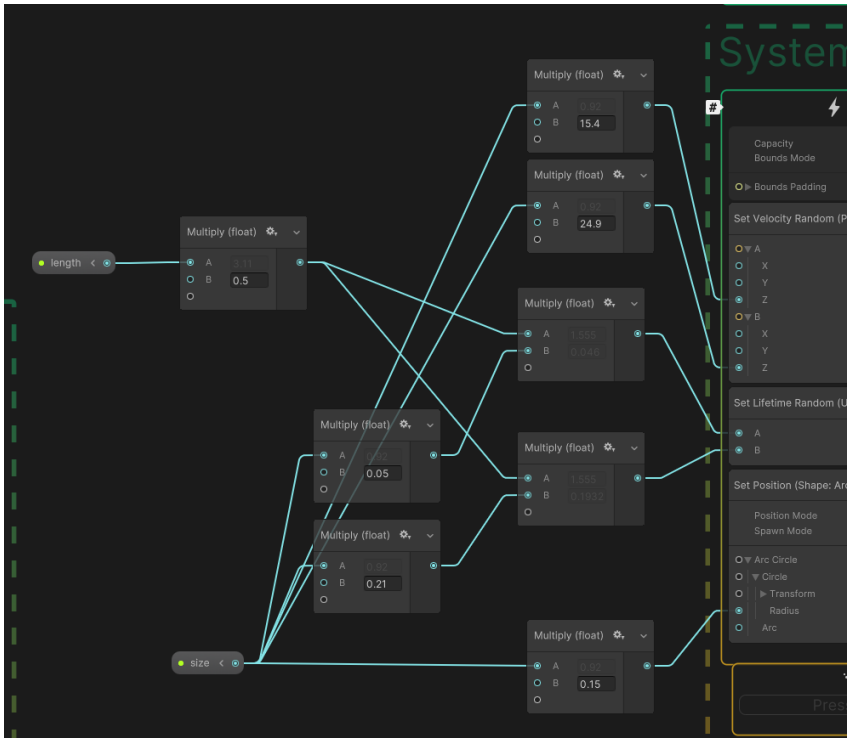


Crete shader wasn't enough. The next part was to make it interact with an environment. At least, it should point to the object and for that its scale should be dynamically changed. For that purpose, created a public variable "length", which tells how long this beam should be.

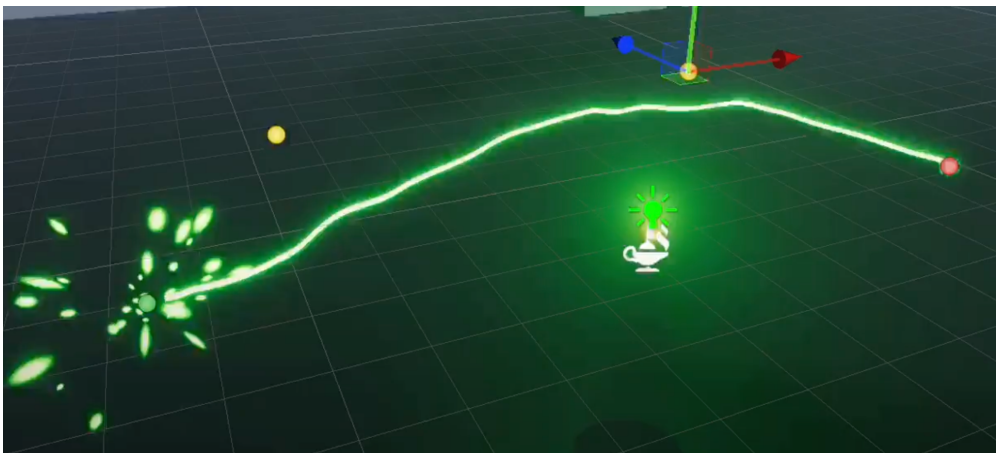


And then, by using this variable and the "size" variable, I can adapt the beam to the proper sizes and proportions.

For example. I need to change a lot of parameters, depending on input length, and all of them are very different, so a different multiplier applies to every output parameter.



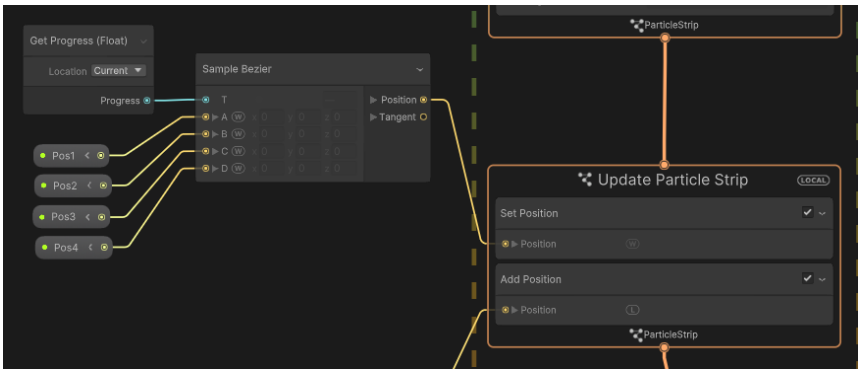
7.3 Grab-beam creation



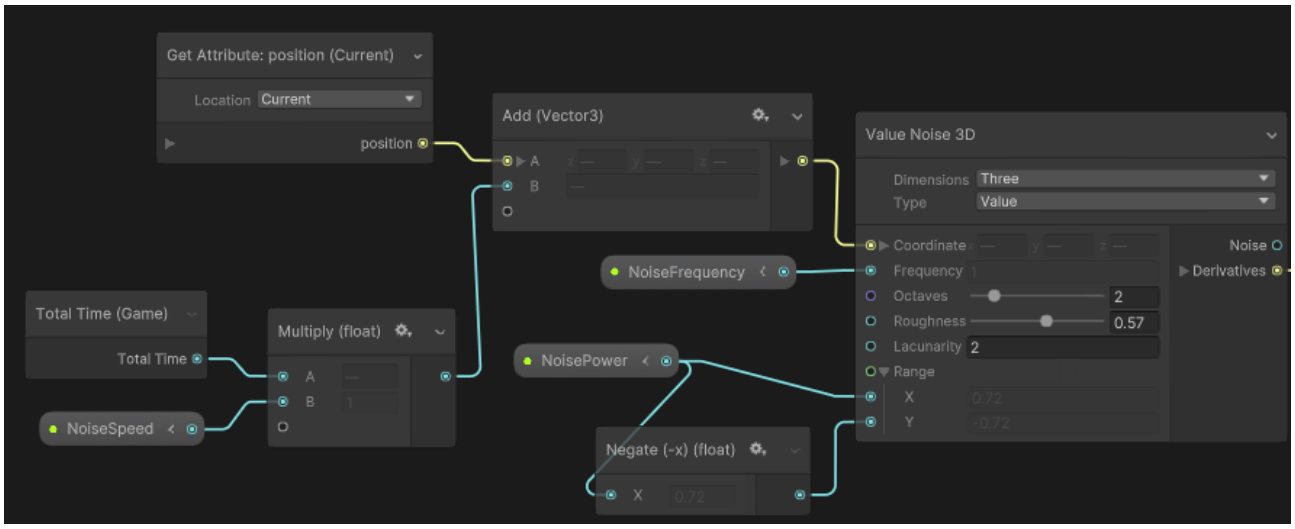
For Grab-wand also needs to be created a vfx beam. But it has a difference from the previous one,

because it should have control points at the start, end, and somewhere in the middle. For that, used the “Particle Strip” variant in Shader Graph.

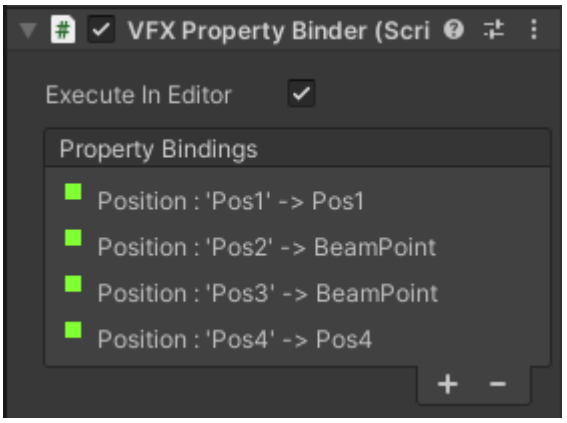
In the position, I’m providing real Transforms from the scene. It has 4 positions: start, end, and two somewhere between them. All these parameters go to the “Sample Bezier” curve.



And then, add some noise effects to it needs to be used noise map and the total time of the game. From the inspector, I can control such parameters as “NoiseFrequency”, “Noise Power”, and “Noise Power”, which helps me to make more chaos or calm. And then, after all these computations, it goes to the “AddPosition” node.



And then, to bind real Transform objects to transform parameters in the shader, I need to use the “VFX Property Binder” component on the object with this shader.



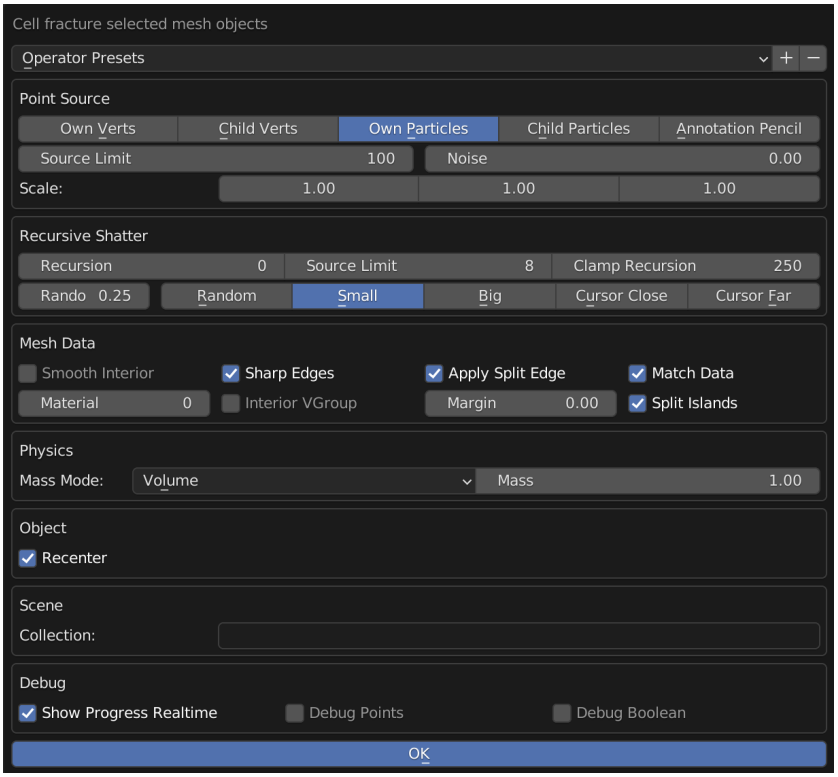
8 *Object destruction*

For objects destruction was used Blender with the Cell Fracture modifier on objects.
Cell fracture is an object destruction method in Blender that allows you to break a mesh into smaller pieces or "cells". It is a useful tool for creating realistic destruction effects in game development.

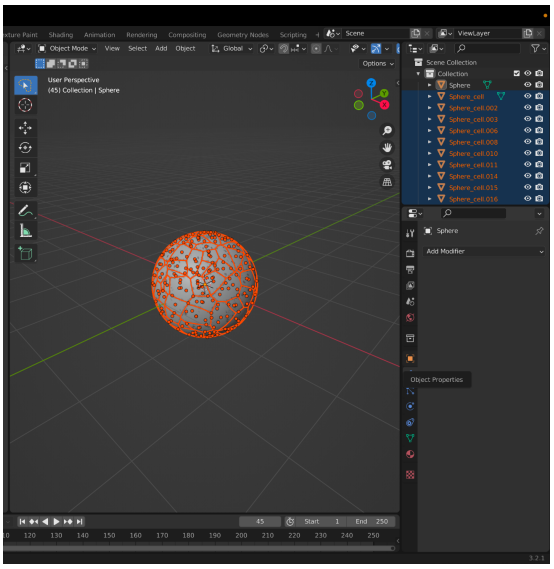
To use cell fracture in Blender, I followed these steps:

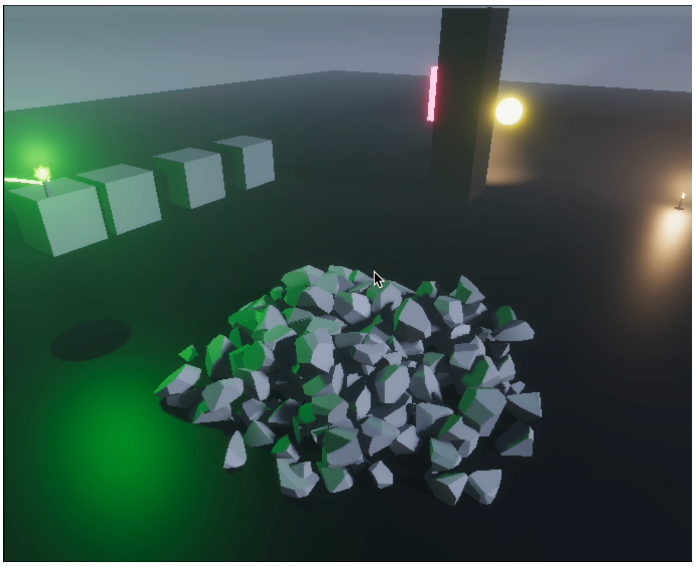
1. Select the object you want to fracture and go to the "Modifiers" tab in the Properties panel.
2. Click "Add Modifier" and choose "Cell Fracture" from the drop-down menu.
3. Adjust the settings for the cell fracture modifier. You can adjust the number of fragments, the shape of the fragments, and other parameters to customize the effect.
4. Click "Fracture" to break the object into smaller pieces.
5. After you have fractured the object, you can animate the pieces to create a destruction effect.

You can use Blender's physics engine to simulate the movement and collisions of the fragments, or you can manually animate them.

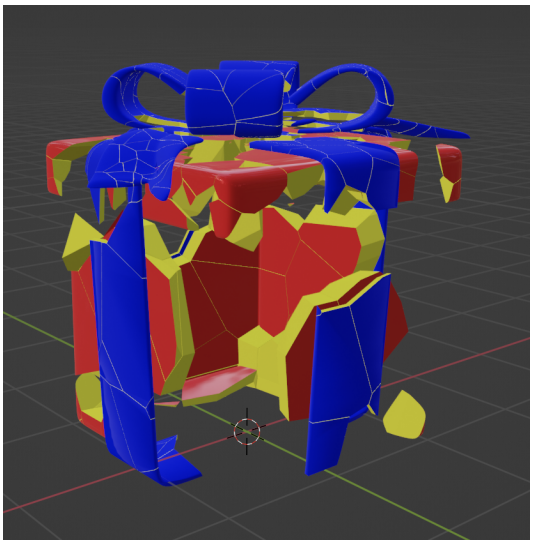
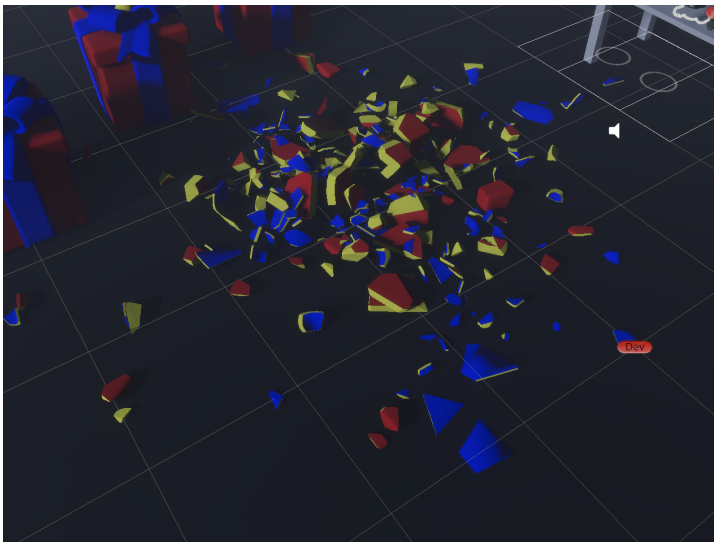


After using this component, I got an easily destroyed variant of the object.
Here is the default sphere and its destroyed variant, already in unity. Works well, but if will be a lot of cells, it will not be good for the optimization process.



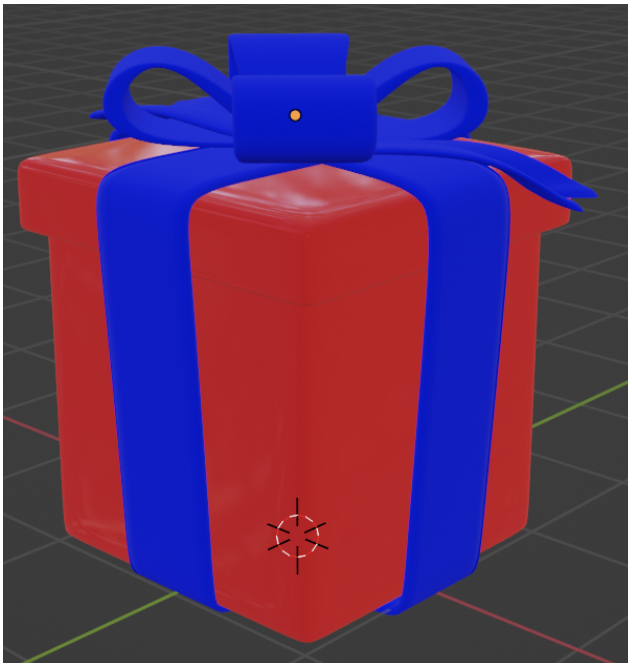


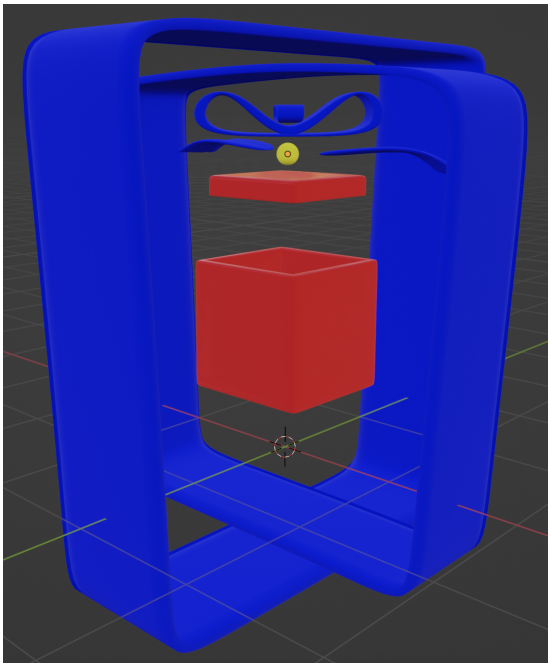
And here is a destroyed variant of the self-modeled present. It doesn't have all parts after the cell fracturing process, I don't know why, but it doesn't matter, because the player won't see it in the game.

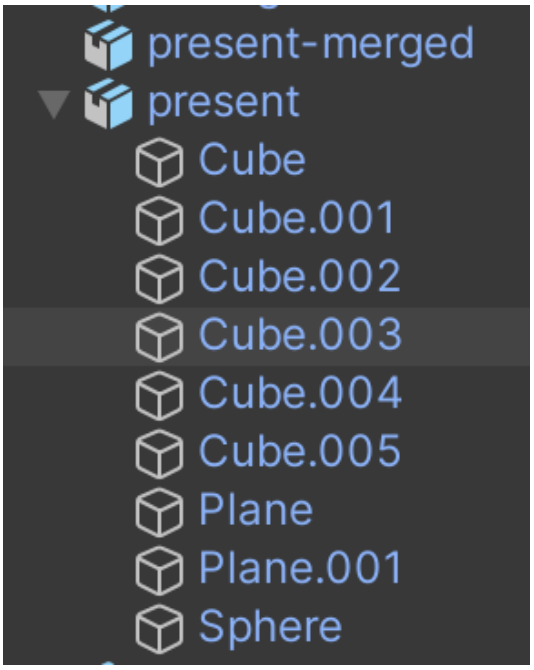
9 *Modelling and Texturing*

In the late steps of the project, I started to model a custom model instead of the default cube. By following this tutorial on youtube, and adding some custom elements, was created and textured the present model. As it's visible from the right image, this model contains several layers.

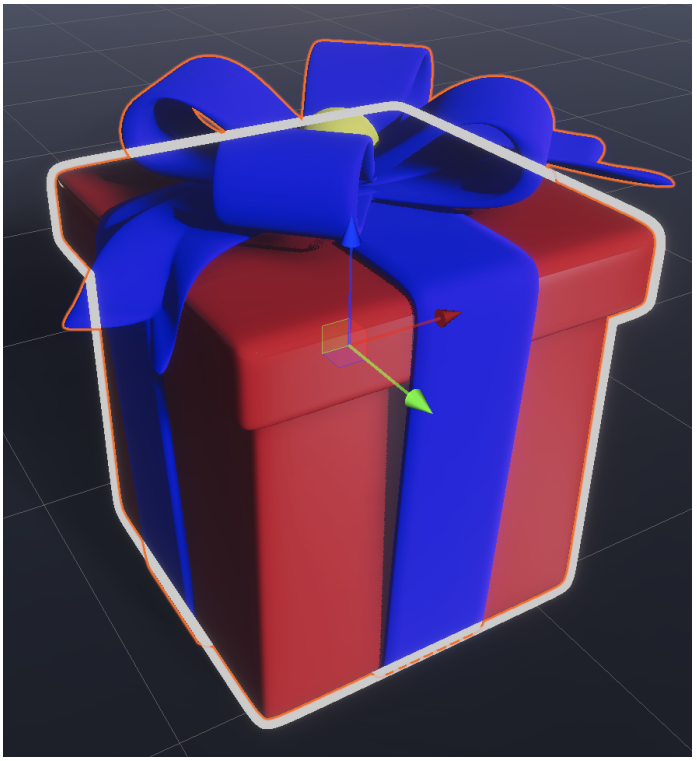


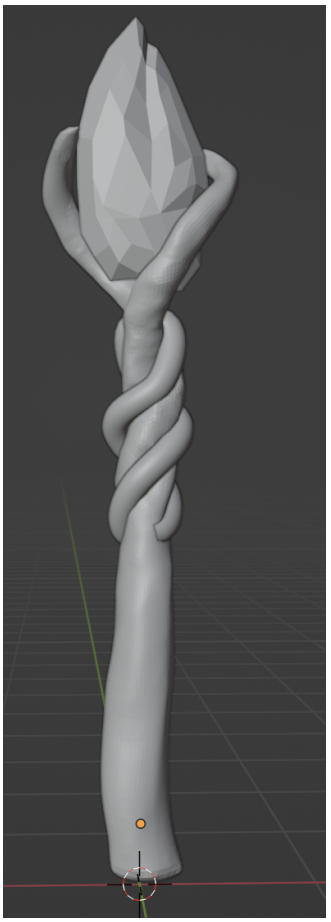


That created problems later with my outline shader, because to Unity it was imported as an object with all these parts as children. And, as was explained above, my shader creates an outline for each individual object.



So, to fix it, I had to merge all these parts into one object. After that was done, the present looked like that in unity. Still not perfect, but already better. I have no idea why the outline doesn't apply to the blue strip, but for now, it's not critical.



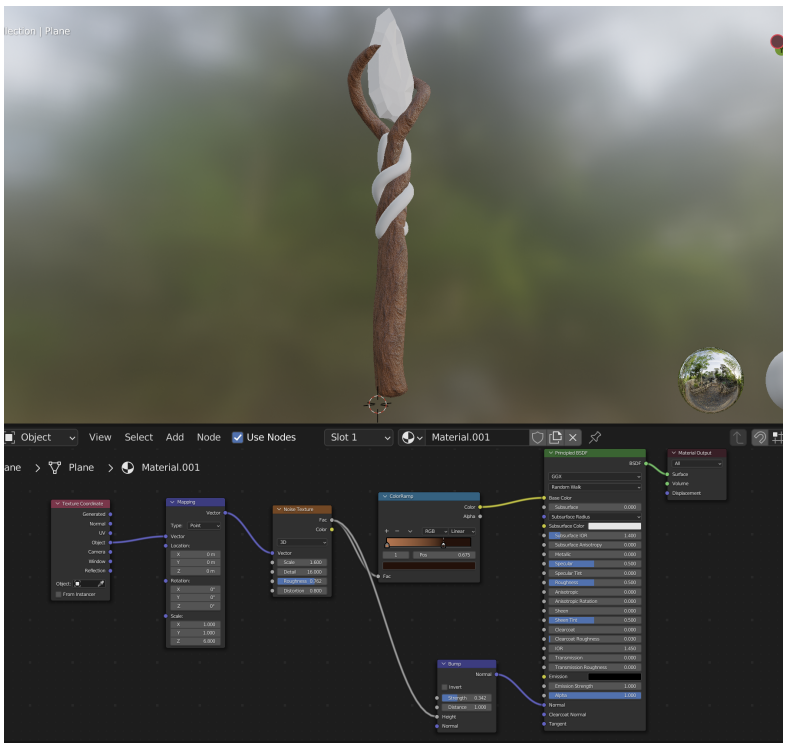


After the presentation was finished, I started modeling, I would say, a central part of this project - wand. I've found some image references of magical wands from pop culture and started modeling my own wand from scratch.

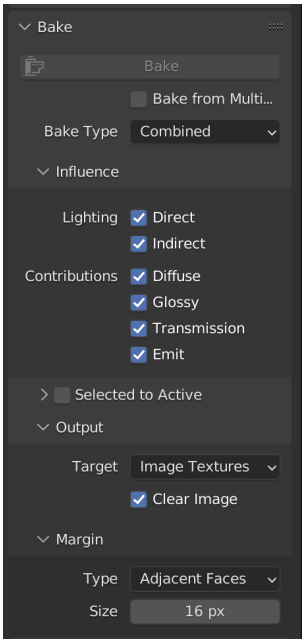
During modeling, I had a problem with meshing, because I had no experience in it and didn't do everything as I should have. But anyway, everything was fixed and the final result looked like that.

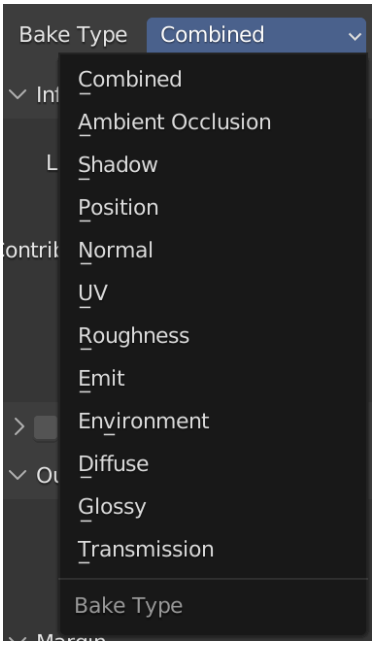
For this model created custom "Heart" part - was a stone in the middle. Also was used Curve component for creating a curved "thing" under this stone.

The next part was texturing. For the wand was created custom wood shader, with a bump map, glossy and etc. And in the end, it looked like this, the image on the right.



But, I didn't expect those materials, created in Blender, are not possible to export to Unity in an easy way. The solution was found by using texture baking.
I had to export the default texture, normal map, and glossy map. By this simple combination, the texture in Unity looks nearly the same as in Blender.





Baking textures in Blender is the process of creating a texture map that captures the details of a 3D model's surface. This can include information such as color, reflection, and surface roughness.

1. Diffuse: This option captures the base color of a 3D model's surface, without any specular highlights or reflections.

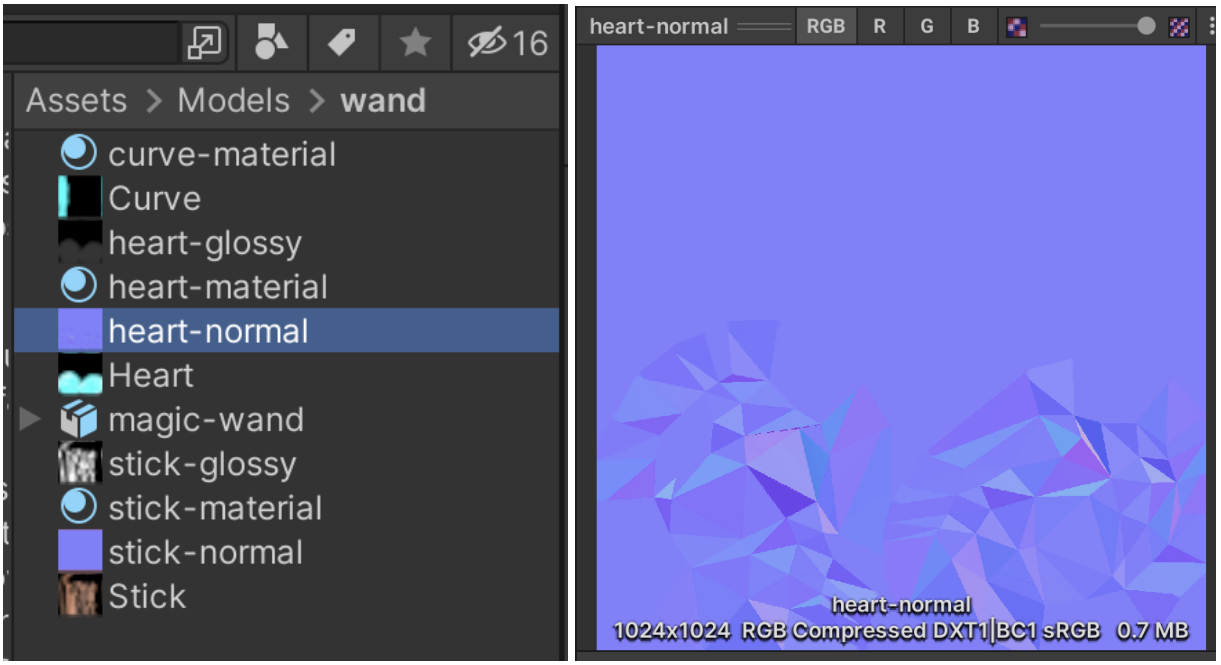
2. Glossy: This option captures the reflection and highlight of a 3D model's surface, which is useful for creating a shiny or reflective surface.

3. Normal: This option captures the orientation of the surface's normal vectors, which can be used to create a detailed 3D surface.

4. Combined: This option captures all the details of a 3D model's surface, including diffuse color, reflection, and surface roughness.

5. Emit: This option captures the amount of light emitted by the surface of a 3D model, which can be used to create a glowing or emissive surface.

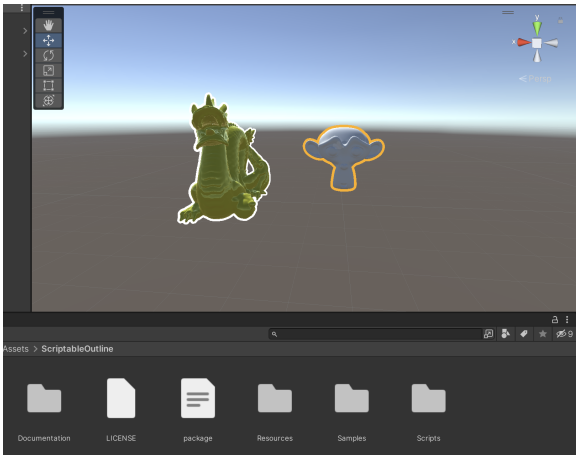
In summary, there are different options for baking textures in Blender using the Cycles render engine. Each option captures different details of a 3D model's surface and can be used to create realistic and detailed textures. Baking textures are a useful technique for improving the performance of a 3D model and making it look more realistic.

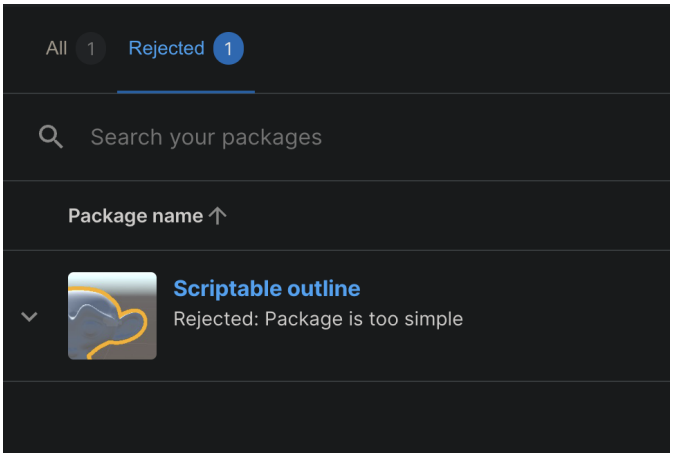


Then, after baking all required textures, that's how they are looking in unity.

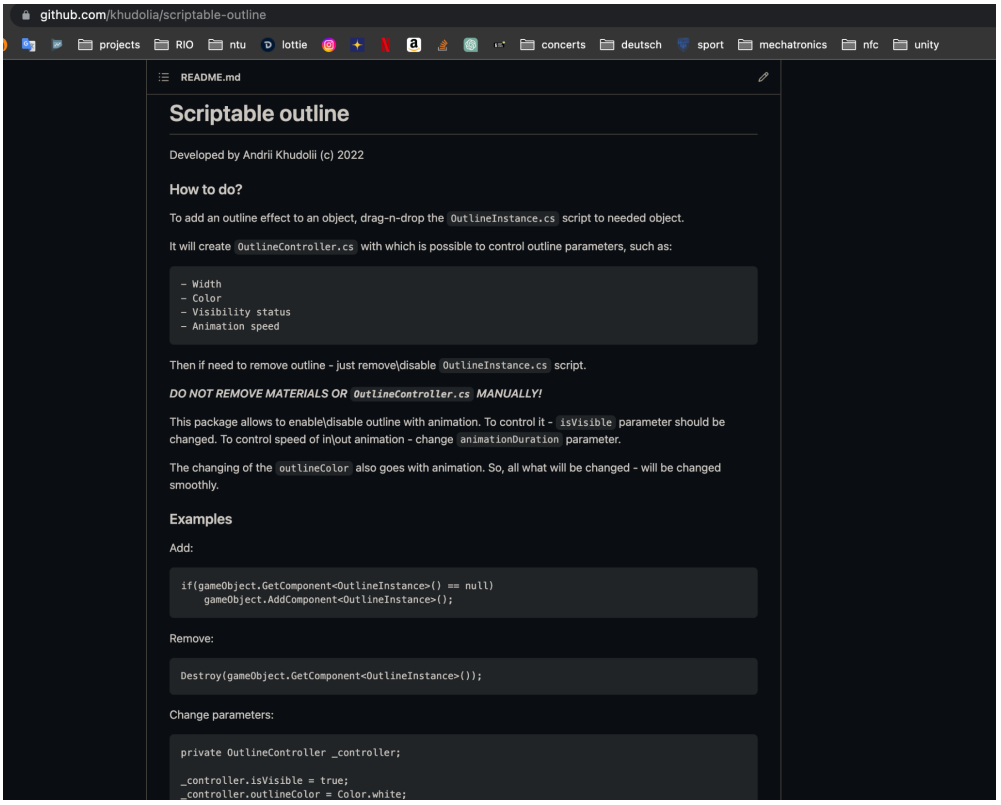
10 Unity AssetStore release

From outline shader was created external package, so anyone in the lab and outside of the university could use it for self-purposes.
But, unfortunately, this package was rejected from the asset store, because it's too simple. Truly, I agree with them, because it's really too simple. So I will add more features to it and will send it again.





But, anyway, it's available on [GitHub] and everyone could download it from there. And also it's available in the lab's teams via the Files->Assets folder.



11 Conclusion

In conclusion, the project involved creating a simple game prototype in Unity using various components and techniques such as object destruction using Blender's Cell Fracture modifier, custom modeling, and texturing, and texture baking in Blender. The project also included the creation of an outline shader that was turned into an external package for public use, although it was rejected from the Unity Asset Store for being too simple. Despite this setback, the project provided valuable learning experiences in game development and asset creation, as well as the opportunity to share these assets with others through GitHub and the university's Teams platform.

12 References

1 Gitlab repository: https://gitlab.reutlingen-university.de/vrlab/vr-vfx/-/tree/develop
2 Github repository: https://github.com/khudolia/vr-vfx
3 VR interaction framework: https://assetstore.unity.com/packages/templates/systems/vr-interaction-framework-161066
4 VRIF documentation: https://wiki.beardedninjagames.com/
5 Present model tutorial: https://www.youtube.com/watch?v=4T5Gg7kb7r8
6 Scriptable outline: https://github.com/khudolia/scriptable-outline