VR Lab #2: Virtual scene creation

1.1. Introduction and goals

When creating a virtual environment, we ask ourselves what should be on the scene. We usually know that our scene should have certain elements, with a certain appearance, and positioned in a certain manner. This appearance and placement may further change over time. There exist several ways to add content to a scene, manually or automatically (procedurally generating content using a script, e.g., when spawning enemies if we are setting up a videogame). When creating several scene elements, a common approach is to leverage a common model of an object, and then change a number of its properties, such as properties related to their appearance, placement, or behavior. As you have learned in VR Lab#1, *shaders* are the main tool to achieve these changes in appearance or behavior.

In this second lab assignment, you are going to learn some of the basics about synthetic content generation. This implies designing your scenes manually, by including, dragging, and scaling items; and also scripting to create dynamic content. Then, you are going to modify those items using shaders, both offline and in real time.

The main task of this assignment is to create a museum with some items of your preference, and to then modify their appearance using shaders. Specifically, you will learn how to:

- Manually set up a scene.
- Use the *Unity Asset Store*.
- Basics about procedurally generating content, and how to trigger scripts.
- Code shaders.
- Modify features, such as appearance properties, in real time.

As in the first lab, there will be different types of tasks. Most of them are small, self-explanatory tasks to give you a robust **basis** for the next assignments. When a new concept is introduced, you may have some **mandatory tasks** to reinforce the learning. Mandatory tasks are preceded by [MT-X]. Finally, **optional tasks** are presented at some points, and preceded by [OT-X]. To be eligible for continuous evaluation, you must complete at least all the mandatory tasks. To opt for the maximum grade (10/10), you need to complete both mandatory and optional tasks.

1.2. Synthetic content

In this first part of VR Lab #2, you are going to create a **minimalist museum**, where you can exhibit any item of your preference. The museum will also have different rooms, so that the user can freely interact within them.

1.2.1. Setting up the scene

We are going to start with a simple, one-room museum, consisting of the floor and four walls. We are not going to include a ceiling to facilitate the manipulation of the elements in the scene. First, create a **new 3D project** as in the previous assignment, where you learned how to use different *GameObjects*.

- To create the room, you should **create five different cubes** (one for the floor and four for the walls). We use cubes instead of planes so the walls have some thickness. You can change their thickness by scaling the object in order to make it similar to a real floor and walls. Once you have done it, you should have something like the room depicted in Figure 1.1. We recommend you to be organized and name all your objects with recognizable names, otherwise it will be hard to identify them once your scene is more populated.

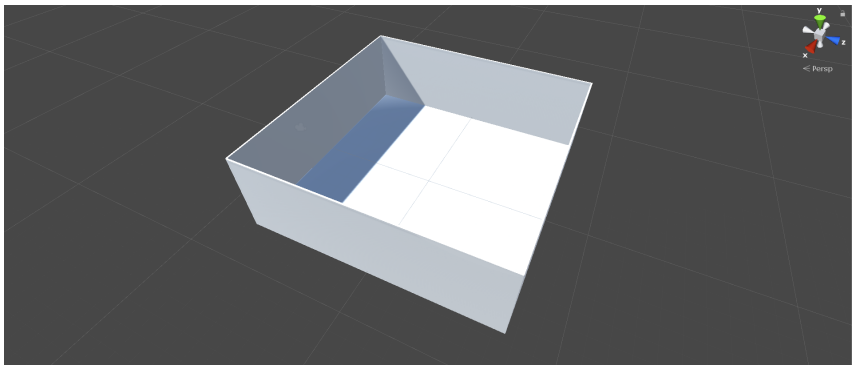

Fig. 1.1: Basic room.

Then, you are going to add four cylinders which will serve as pillars to exhibit the masterpieces.

- **Create four cylinders (which will serve as pillars) and modify them** until you have a room like the one in Figure 1.2.

Now you are going to select which items you would like to exhibit in your museum. We are going to use complex primitives and geometries, instead of the basic ones provided by Unity. The easiest way (but not the only one) to add sets of new models into your project is by using the *Unity Asset Store*. You can access the store by clicking in the *Asset Store* tab at the same level that your *Game Editor* tab. If you don't see it, click *Window*, and then Asset Store.

- You will need a Unity account in order to be able to import assets.

- When you are logged in, you can type in the search bar to find whatever asset you would like

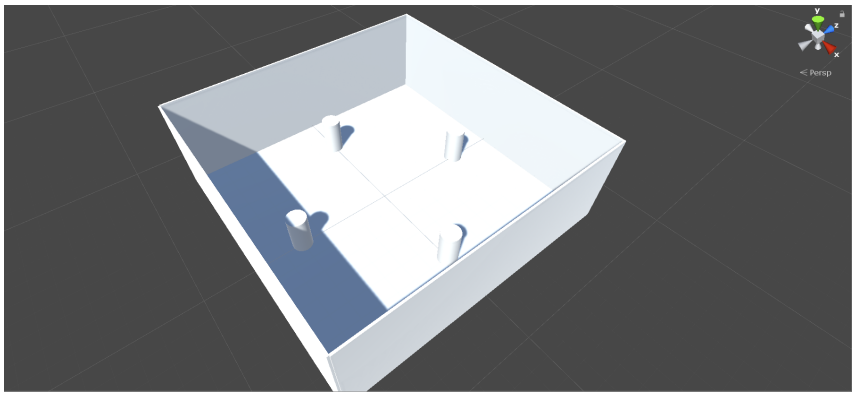

Fig. 1.2: Basic room with pillars.

to include in your museum. We encourage you to search for *low poly* items: This kind of models are formed with a very small number of triangles, which make them easy and computationally cheap to render (if your computer is not powerful, this type of models is well suited).

- Once you have identified a free pack of your preference, you can add it to your project by clicking the *Add to My Assets* button. Then, navigate to Unity's main menu and go to *Window* > *Package Manager*. In the *Packages* tab, select "My Assets", then select the ones you wish to download, and then import. Make sure that **every** checkbox is selected, and then click the *Import* button. In the present example, *Ultimate Low Poly Dungeon* has been downloaded and imported.

- The new models you have imported to your scene are in a new folder that has been created when importing. Locate some object that you would like to include in your museum, and drag it into your scene. You will need to drag the corresponding *prefab*. Move it and scale it properly, so that each of the included assets is placed on a pillar. In this example, we have distributed the room as depicted in Figure 1.3.

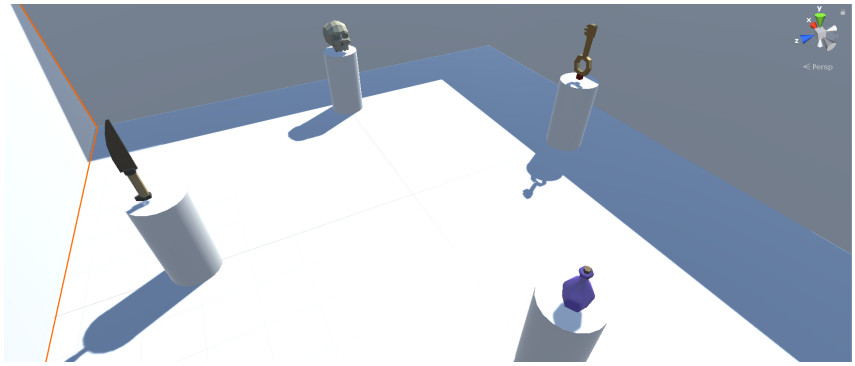

Fig. 1.3: Basic room with example items.

1.2.2. Moving around the scene

In the previous lab, you have learned how to rotate the camera with the WASD keys. In this lab, the camera will move (translate) with the WASD keys, while the mouse will be used to

rotate it.

- To add this functionality to your camera, you must add a script, and assign it to the main camera object, by dragging and dropping it into the corresponding game object (as you learned in the previous lab). Remember to create the script in a *Scripts* folder, to keep your project clean and legible.

- This script can be based on the following code: `https://gist.github.com/gunderson/d7f096bd07874f31671306318019d996#file-flycamera-cs`, or any other of your preference whose functionality is similar.

1.2.3. Scripting *GameObjects*

As you probably know, many virtual (and even traditional) experiences procedurally generate content, and this generation can be conditioned by the user's position, orientation, proximity, etc. In this museum, there will be an empty room that behaves in that way.

- **Add an empty room** to your museum. You have to create a new room in your museum, as you did with the previous one. You should get something similar to Figure 1.4. Remember that you can **duplicate** items such as walls or floors, just by right-clicking them and selecting *duplicate item*.

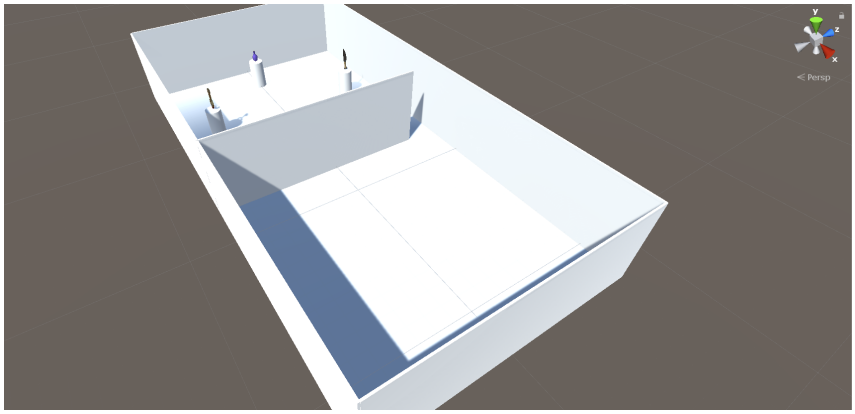

Fig. 1.4: New room incldued in the museum.

Our procedural generation of content will consist in randomly spawning objects. To do this, you will need to create a new *GameObject* that runs the script that will generate those objects. In Unity, **any script has to be attached to a game object in order to work**.

- Go to the tab *GameObject*, and then select *CreateEmpty*. You can rename it as *Spawner-GameObject*, so you can easily recognize it.

- In the *Scripts* folder of your project, create a new script by clicking the *New - C# Script*. Rename it as *ItemSpawner*. Finally, open it and add the following snippets of code.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class ItemSpawner : MonoBehaviour
{
    // Properties that can be changed from the Inspetor tab
    public GameObject ItemToSpawn; // Item to be spawned
    public Vector3 center; // Center of the cube to spawn
    public Vector3 size; // Size of the cube to spawn

    // Start is called before the first frame update
    void Start()
    {
      SpawnItem();
    }

    // Update is called once per frame
    void Update()
    {
      if (Input.GetKey(KeyCode.Q)) // When Q is pressed, an item is spawned
        SpawnItem();
    }

    public void SpawnItem()
    {
        . . .
    }

    // Debug funcion. In the Scene tab you can see a Red Box, which is the
    // volume where the object is going to be spawned.
    void OnDrawGizmosSelected()
    {
      Gizmos.color = new Color(1, 0, 0, 0.5f);
      Gizmos.DrawCube(center, size);
    }
}
```

When the game starts, the function is called, and starts running.

On the other hand, the `Update()` function is called once per frame. If the 'Q' key is pressed, it will therefore call the spawning function (`SpawnItem()`) too.

Finally, the `OnDrawGizmosSelected()` will allow you to debug your application in the *Scene* tab.

```
{
    . . .

    public void SpawnItem()
    {
      // Position to spawn
      Vector3 pos = center + new Vector3(Random.Range(-size.x/2, size.x/2),
                                         Random.Range(-size.y/2, size.y/2),
                                         Random.Range(-size.z/2, size.z/2));

      // Instantiate the object
      Instantiate(ItemToSpawn, pos, Quaternion.identity);
    }

    . . .
}
```

The spawning function instantiates an item in a pseudo-random position of the room.

- Now you can test the code by dragging the script into your recently created empty object. You may have noticed that there are three public variables in your code (namely `ItemToSpawn`, `center`, and `size`). You will have to modify and fine-tune them.

- Click on the object in the *Hierarchy* tab, and then **edit the `center` and `size` properties** associated to the script in the *Inspector* tab.

- Then, select an object from your *Asset* folder (e.g., one of the low-poly assets you just downloaded), and **drag it** into the property `ItemToSpawn`.

- Now, run the program and press 'Q' a few times. Take a look at your *Scene* tab. You should see how your item has spawned there multiple times, as depicted in Figure 1.5.

The previous code spawns a book if the 'Q' key is pressed. You can test that the script code works properly by pressing 'Q' and checking if it spawns the item you have selected.

[MT-0] Implement the required modifications to spawn the item only if the user (the camera) is inside the room, and only one item per second. To control elapsed time, check out the function `Time.deltaTime`.

[OT-0] Create another room with four cylinders as pillars. In this room, when the user is looking at the pillar, it has to be destroyed or made invisible. You can put items on top of the pillars and destroy the items instead of the pillars, if you prefer. Some things to consider:

- Have a look to the Unity function `RaycastHit`[1,2].
- For debugging, have a look at the function `Debug.DrawRay`[3] to draw a visible line and

[1] `https://docs.unity3d.com/ScriptReference/RaycastHit.html`
[2] `https://docs.unity3d.com/Manual/CameraRays.html`
[3] `https://docs.unity3d.com/ScriptReference/Debug.DrawRay.html`

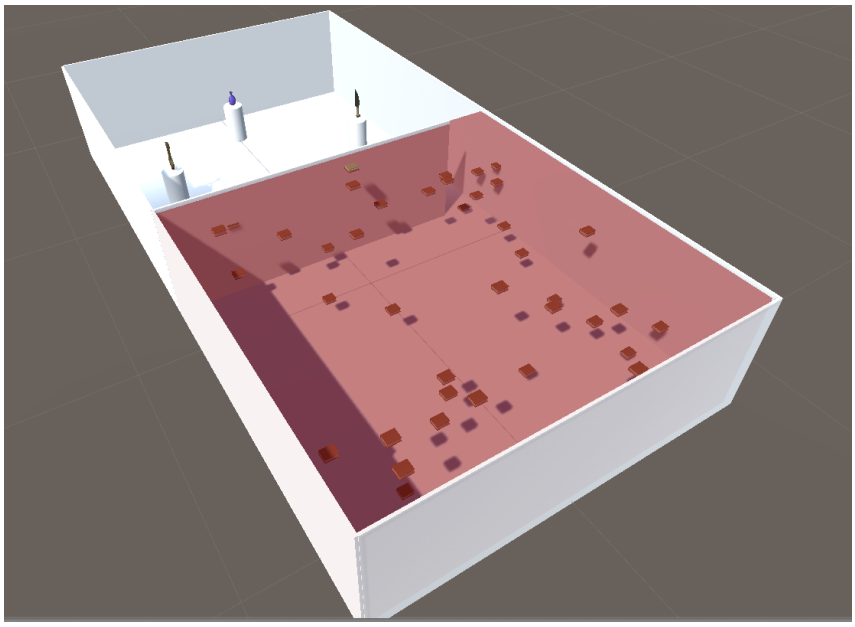

Fig. 1.5: Room with a book spawned multiple times.

know where the camera is looking at, and at the class *LineRenderer*[4] as well.

- To avoid destroying every item you look at, you can use **tags**[5], and therefore delete only certain tagged objects.

1.3. Shading

In the second part of VR Lab #2, you are going to learn how to program slightly more complex shaders. Shaders are useful to manipulate the appearance properties of an object.

1.3.1. Creating a simple shader

First, you are going to program a simple shader which indicates the **albedo** of the object, i.e., the diffuse reflection of total incident light or, in other words, the color of the object.

- Create a material in order to assign a shader to it. Shaders cannot be assigned directly to objects, they have to be *linked to a material* instead. Besides, different materials can make use of the same shader. Therefore, create a *Materials* folder, and create a new material.

- Create a new *Shaders* folder, and add a new standard shader.

- Then, drag the material to one of the objects that are over a pillar in your museum.

[4] `https://docs.unity3d.com/Manual/class-LineRenderer.html`
[5] `https://docs.unity3d.com/Manual/Tags.html`

- Assign the new shader to the material of the object. You have to drag your *shader* file to the *material* component of your object.

- Now, open your shader. You will notice that there is some default code included. Replace that code with the following code (be careful with the first line of code, since it is the shader name)[6]:

```
Shader "Custom/CoolShader" {
  Properties {
    _customColor ("Main color", Color) = (1, 1, 1, 1)
  }

  SubShader {
    CGPROGRAM
    #pragma surface surf Standard
    struct Input {
        float2 uv_MainTex;
    };

    fixed4 _customColor;

    void surf (Input IN, inout SurfaceOutputStandard o) {
        o.Albedo = _customColor.rgb;
    }
    ENDCG
  }
  Fallback "Diffuse"
}
```

In the code above, the first line indicates where the shader is going to appear in the hierarchy. Then, there is a *Properties* block, where you can write all the properties that the shader is going to have. These properties' values can be changed directly from the inspector window without modifying any code.

Inside the `SubShader` block, you can define all the structs and variables that are going to be used.

Finally, the `surf` function maps the variable values into the object. In this case, the *albedo* of the material is modified, hence modifying the main color of the object. You can see an example depicted in Figure 1.6. You can also check all the properties of the surface in the official documentation[7]. **Note that shaders are programmed in GLSL language, which is a bit more complex than other languages. You can find multiple tutorials through the internet**[8]

[6] `https://docs.unity3d.com/Manual/SL-SurfaceShaderExamples.html`
[7] `https://docs.unity3d.com/Manual/SL-SurfaceShaders.html`
[8] `https://learnopengl.com/Getting-started/Shaders`

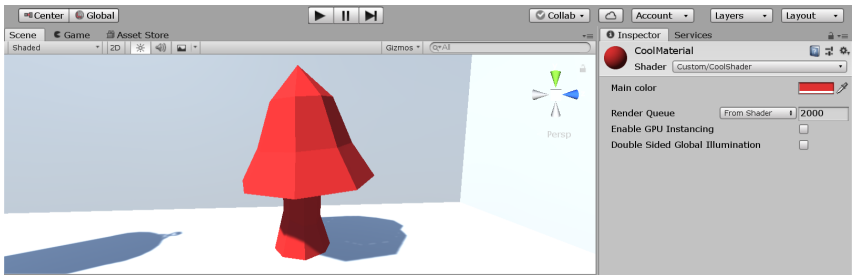

Fig. 1.6: Simple shader.

[MT-1] Create a new room in your museum that has some objects that glow, i.e., objects that have some light *emission*. *Tip*: You have to check which property should change in order to make an object *emit* light of a certain color. Remember that you can use the same shader in multiple materials, and change the properties of each material in isolation.

As you may have realized after doing this task, light emitted by objects does not affect (reflect on) the walls. Considering interreflections between objects (global illumination) is a non-trivial task when programming shaders, and is out of the scope of this lab session.

[MT-2] Now create a shader that allows you to include a texture[9]. To do that, you should create a shader with a property of type *Texture*, and map it to the albedo with the function `tex2D`. Check the Unity API documentation if you have any doubts. You should obtain something similar to what is depicted in Figure 1.7, but with your own texture.

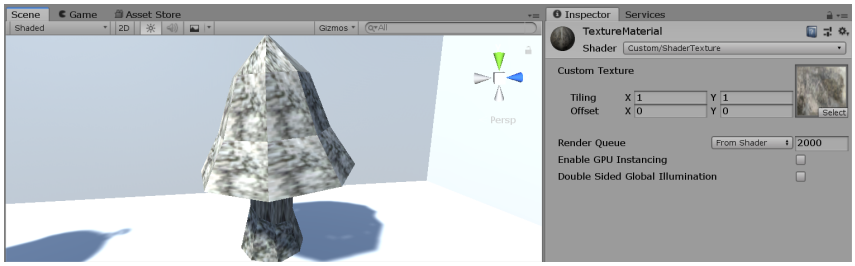

Fig. 1.7: Object with a texture.

1.3.2. Changing shader properties in real time

You can change some of a shader's properties in real time. You will now implement an effect that allows elements to disappear as you get close to them.

- First of all, create a new shader, and assign it to a new material.

- Add the following code to your new shader:

[9] In our context, a texture will be an image that, mapped to our object, allows us to easily modify some property or properties of the object's appearance in a spatially-varying manner.

```
Shader "Custom/TransparentShader"
{
    Properties
    {
        _Color ("Color", Color) = (1,1,1,1)
        _Transparency ("Transparency", Range(0, 1)) = 1
    }
    SubShader
    {
        Tags {"Queue" = "Transparent" "RenderType" = "Transparent" }

        CGPROGRAM

        #pragma surface surf Standard fullforwardshadows alpha:fade

        sampler2D _MainTex;

        struct Input {
            float2 uv_MainTex;
        };

        fixed4 _Color;
        fixed _Transparency;

        void surf (Input IN, inout SurfaceOutputStandard o)
        {
            o.Albedo = _Color.rgb;
            o.Alpha = _Transparency;
        }
        ENDCG
    }
    FallBack "Standard"
}
```

- Then create a script and assign it to the game object we want to hide from the user.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Disappearing : MonoBehaviour
{
    private MeshRenderer meshRenderer;
    void Start(){
      meshRenderer = GetComponent<MeshRenderer>();
    }
```

- As done in one of the previous tasks, we will calculate the user's position and distance to the corresponding object. You can base your script in the following snippet:

```
    // Update is called once per frame
    void Update()
    {

        // Calculate transparency depending on distance

        // You have to select the property name, not the display name
        meshRenderer.material.SetFloat("_Transparency", transparency);
    }
}
```

Now run the application and check that the object disappears as the user gets closer.

1.3.3. A final, optional task

- **[OT-1]** As a small homage to Indiana Jones, we are going to dedicate one of the museum rooms to recreate an adaptation of the treasure scene (see Figure 1.8). Put a treasure in a pillar in the center of the scene, give it a gold-ish appearance with a shader **coded by you**, and make the object rotate faster as you get close to it. Once you are close enough (you decide what "close enough" means), teleport the object to another part of the room. You could obtain something similar to Figure 1.9.

Fig. 1.8: Indiana Jones treasure scene.

Fig. 1.9: Example of the adaptation of the Indiana Jones scene.

1.4. Reporting your results

In this lab assignment you should submit a short **video**, showing that your museum is well designed. Make sure that the video correctly shows all the required aspects (main tasks, and any other additional feature you included). The video does not need any further edition (do not include texts, canvas, or audio; just record your screen while running your application). You can use free, open-source software such as OBS[10] for the recording. If the video is too large for the Moodle task (above 50MB), you can submit a .TXT file with a link to Google Drive (using your Unizar account).

You also have to submit your **shaders** with representative names, as well as all the **scripts** that you coded. All of them have to be submitted in a .ZIP file. That zip should also include a brief .PDF file indicating what each shader/script does, in a concise, yet descriptive sentence. Some guidelines that you need to follow:

- The video file should be named `labXX-video-YYYYYY.[mp4|avi]`. The .ZIP file should be named `labXX-supp-YYYYYY.zip`. In both cases, `XX` is the number of this lab, and `YYYYYY` your NIP.
- You can include any other relevant material in the .ZIP file.
- We will not look at submitted items that have not been referenced in the .PDF file (remember that a single, descriptive sentence explaining the utility of each element is enough).

In terms of evaluation, mandatory tasks can get you up to 8 points out of 10, and the 2 remaining points can be obtained through the optional tasks, and depending on the final result of your assignment.

[10] `https://obsproject.com/es`

You should submit the assignment via Moodle, uploading them in the corresponding Moodle task. **Deadline: March 19th, 2024 - 23:59**.