VR Lab #1: Introduction to Unity and 360º content visualization

1.1. Introduction and goals

Virtual Reality is experiencing an unprecedented growth. On the one hand, technology is rapidly evolving, which is leading to a decrease in the price of many consumer-level VR devices (e.g., Oculus Quest, HTC Vive, Valve Index...), as well as an increase in their availability for many users. On the other hand, professional creators and practitioners are exploring new ways of presenting a wide variety of content: cinematographic content, videogames, narrative experiences, virtual tourism, or even urban design.

While for some of those applications 3D, computer-generated content is enough, for others one may want to display real, captured footage in a VR headset. How is this real footage captured? How can we visualize it in a VR headset? In this lab assignment we will learn a very common way of representing and visualizing real, captured footage (which can also be used for synthetic content). It will also serve us as an introduction to Unity, and functionalities that we will be using throughout the course.

Specifically, throughout this assignment you will learn the following:

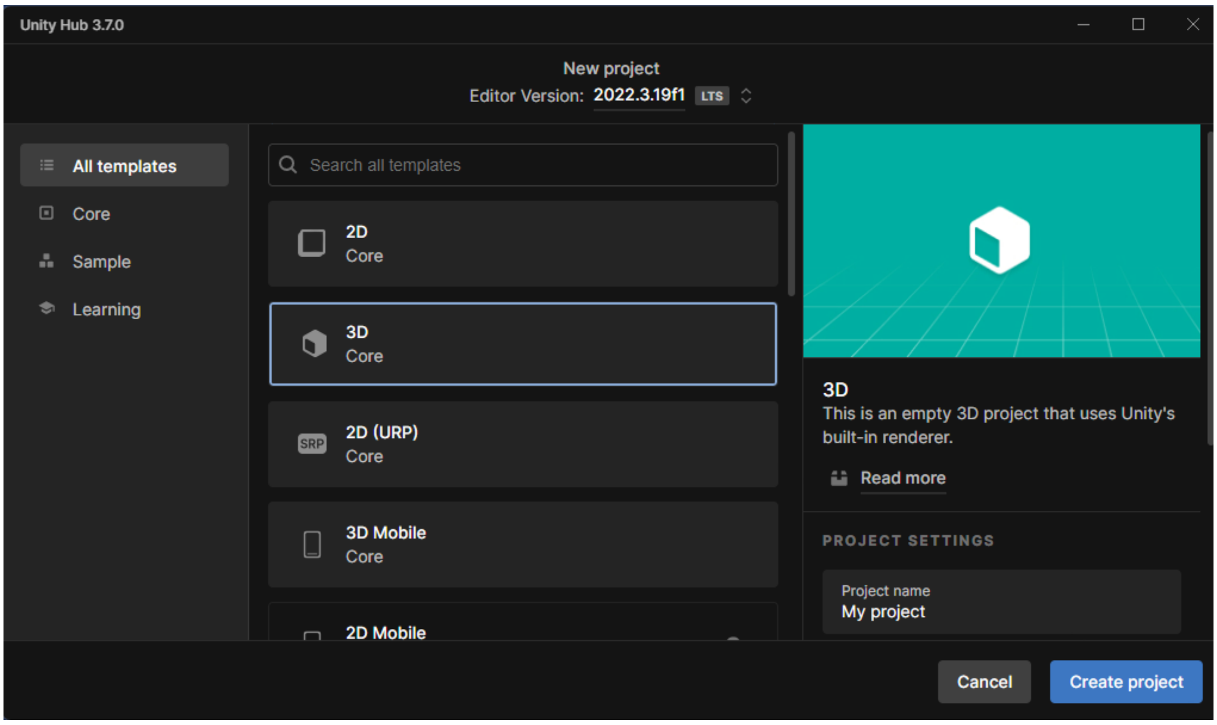
- A short introduction to Unity and its use for developing applications for VR.
- A short introduction to 360º content acquisition and storage.
- Visualization of 360º content in a VR headset.
- Basic use of the Unity engine, and its VR functionalities.
- Shader basics, including what is a shader and why are they useful.

There will be different types of tasks in this lab sessions. 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 [0T-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. Introduction to Unity

Unity is a powerful, publicly available cross-platform game engine, which allows to design, build, compile and run a wide variety of games and experiences in both 2D and 3D. Working in most of the current available development platforms, Unity has a basic programming API in C#, and its engine runs over OpenGL (in Windows, Mac and Linux), Direct3D (in Windows) and OpenGL ES (in Android y iOS). It provides an huge support in shading techniques, and facilitates the design thanks to its drag-and-drop system.

To complete this assignment (and also for the next ones in this course), a complete installation of Unity is needed. This means you will need to **install Unity**. Unity offers a free, non-commercial purposes license that is available for any user. Note that Unity recently introduced the *Unity Hub*, which will allow you to manage multiple versions of the editor, add modules, create new projects, etc. For this course, we recommend **Unity 2022 LTS**.



3D
This is an empty 3D project that uses Unity's built-in renderer.

Project name
My project

Fig. 1.1: Unity hub with Unity 2022 LTS installed. When you create a new project, you can select different base scenes or templates depending on your needs.

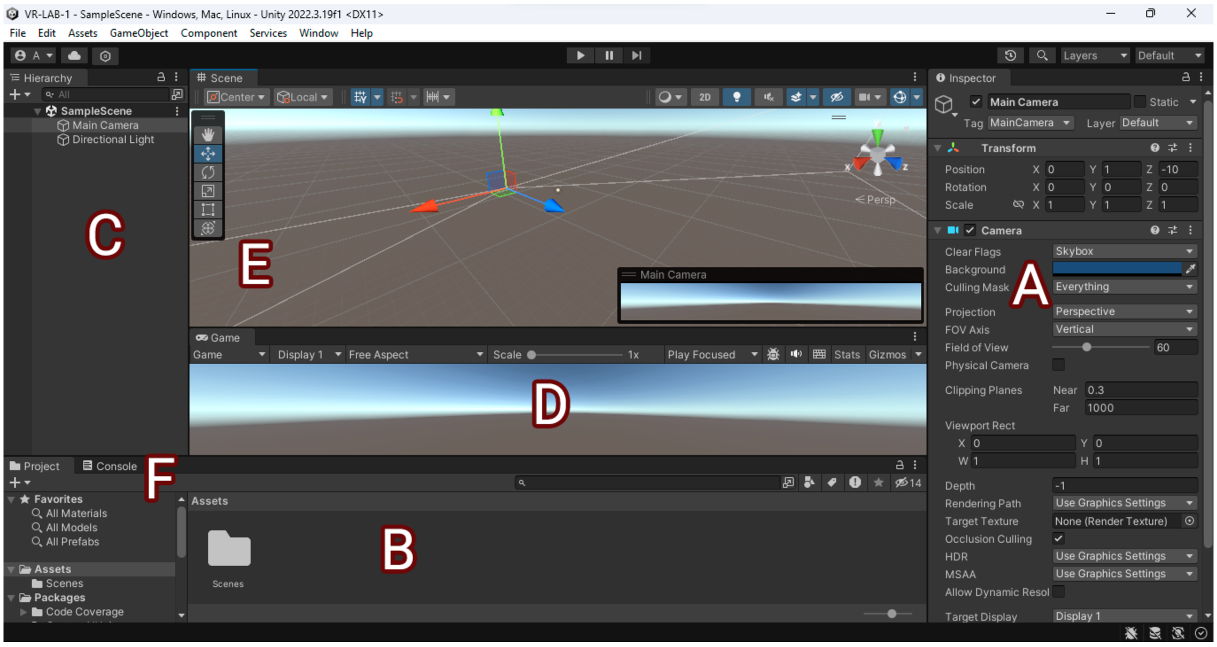
When you create an empty **new 3D project** in Unity (Figure 1.1), after the program loads, you will see a screen similar to the one shown in Figure 1.2 (although it may be arranged differently depending on the Unity version). You will see the following tabs:

- A - Inspector tab. The inspector tab contains the properties of the object you select in the virtual world. Here you can check and modify object's position, rotation, or scale; see attached scripts to an object, or its corresponding shaders.
- B - Project tab. It works like an common PC explorer: you can see the folder structure of

your project and the files contained in each of them. **Good practices: Create folders to separate each different type of component you work with: shaders, scripts, material, etc.**

- C - Hierarchy tab. All your scene objects will be included in this hierarchy. By default, all Unity projects start with a camera and a directional light.
- D - Game tab. You will see your running application in this screen.
- E - Scene tab. This tab contains the scene itself. You can drag and manipulate objects in this scene.
- F - Console tab. By default, you can alternate your scene and your project tab, although you can rearrange all tabs as you like. Console tab is extremely important: it will print any error message that happens during your program compilation or in running time.

Note that you can move, add, or remove any Unity tab to create your custom structure, but we recommend you to keep the default one, and add the tabs you may need.


VR-LAB-1 - SampleScene - Windows, Mac, Linux - Unity 2022.3.19f1 <DX11>

Main Camera
Tag MainCamera Layer Default

Transform
Position X 0 Y 1 Z -10
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

Camera
Clear Flags Skybox
Culling Mask Everything
Projection Perspective
FOV Axis Vertical
Field of View 60
Clipping Planes Near 0.3 Far 1000
Viewport Rect X 0 Y 0 W 1 H 1
Depth -1
Rendering Path Use Graphics Settings
Target Texture None (Render Texture)
Occlusion Culling ✓
HDR Use Graphics Settings
MSAA Use Graphics Settings
Target Display Display 1

Fig. 1.2: Default Unity screen. See the text in Section 1.2 for an explanation of each tab.

1.3. Unity and VR

Unity provides a variety of software development kits (SDKs) to support a broad range of XR devices, which encompass virtual reality (VR), augmented reality (AR), and mixed reality (MR) technologies. Depending on the operating system (OS) your application will run on and the specific device you plan to use (such as Google VR for Android/iOS, Oculus, Open VR, and others), the process of setting up your scene may differ slightly. Fortunately, Unity offers extensive documentation and sample scenes that can assist you throughout the process[1]. For the moment, we are not going to worry about this and we are going to create a simple scene to visualize 360º content in our computer.

1.4. 360º content: The basics

One of the key aspects of VR is that it offers an immersive environment all around the viewer. Unlike traditional media, where the content is displayed in a 2D screen in front of the viewer, content in VR is displayed in all the 360 degrees that surround the user. One of the most common ways to represent this content are the so-called equirectangular (or 360º) panoramas.

Panoramas depict a 2D projection of a three-dimensional scene, similar to an unfolded world map (see Figure 1.3). You can think of them as if you were laying out an spherical Earth globe into a 2D world map. In this lab, we are going to work with this kind of representation, which is very useful in VR, especially (but not only) when working with captured content. You will learn a bit more in Lesson 4 of this course.

Some existing datasets have sample panoramas publicly available [1, 2]. You have some example panoramas from these datasets in Moodle. **Download some equirectangular panoramas** (from these sources or others); we will need them for the next steps in the lab.

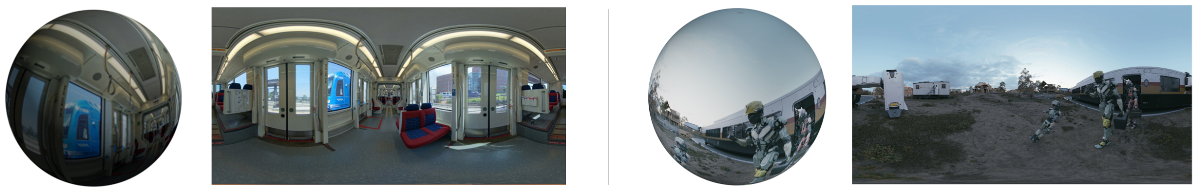


Fig. 1.3: Examples of equirectangular panoramas [1, 2] reprojected to a sphere. Note that the distortions present in the 2D projection are not visible when projecting into the sphere.

Equirectangular panoramas' inherent structure allows a **simple projection to a spherical geometry** (see Figure 1.3). The first task on this lab assignment is to project equirectangular panoramas to a spherical geometry in Unity, to visualize the 360º scene.

1.4.1. Creating the scene

- Go to the *scene tab*, and click on the camera. The *inspector tab* now show main camera's properties. Make sure your camera's position and rotation are $(0, 0, 0)$, so that it is easier to have a clear reference at the center of the scene.

- We want to visualize 360º panoramas. As we have learned before, we will need an **spherical geometry**. Right click in the *hierarchy tab*, select *3D object* and then **Sphere**. A new sphere called "Sphere"will appear in the scene and in the *hierarchy tab*.

[1] https://docs.unity3d.com/Manual/XR.html

- Click the sphere, go to the *inspector tab*, and change its position to $(0, 0, 1)$. If you now look at the *game tab*, you should see the sphere in front of you.

- Now, we have to import the panorama we previously downloaded to the project. Go to the *project tab*, right click and choose **Create** and then **Folder**, and rename it “Panoramas”. Go to your computer’s folder where you saved your panoramas, and drag them to the recently created folder.

- To project the panorama into the sphere, just drag your panorama to the **Sphere** object in the *hierarchy tab*. Unity will automatically project the panorama and you will see the sphere re-textured.

- We now have a sphere with a equirectangular panorama projected on it, but our camera is out of the sphere and we are not seeing any virtual environment (yet). Move the **Sphere** to the $(0, 0, 0)$ position, and change its scale to $(2, 2, 2)$. At the moment, you should see that the sphere disappears and only the background shows.

Why is this happening? By default, OpenGL renders the scene by a process called **Face Culling**. Everything in the graphic pipeline is rendered by triangles. Triangle primitives, after all transformation steps, have a particular facing. This is defined by the order of the three vertices that make up the triangle, as well as their apparent order on-screen. By default, OpenGL discards objects based on their apparent facing. As the **Sphere** normals are pointing to its outside by default, Unity will not render it when our camera is looking at it from the inside. Note: If you are new to the graphics pipeline, have a look into Appendix A for a brief overview.

- We want OpenGL to render the sphere from the inside. To this, we are programming a *shader*. Go to the *project tab* and create a new folder called “Shaders”. Go in, right-click, choose “Create”, “Shader”, and then “Image Effect Shader”. Rename it so you easily identify it.

- Double-click the *shader* to see its content. First of all, change the first line to rename your *shader*. You can call it:

```
Shader "Hidden/Panorama"
```

Then, you will see a properties section.

```
Properties
{
    _MainTex ("Texture", 2D) = "white" {}
}
```

Shaders accept **Textures** as input arguments. You can pass a shader as many textures as you want. You will then see a Subshader section, where other graphic pipeline’s properties can be set.

```
    SubShader
    {
        // No culling or depth
        Cull Off ZWrite Off ZTest Always
```

Then, some macros are included in the sader.

```
    Pass
    {
        CGPROGRAM
        #pragma vertex vert
        #pragma fragment frag

        #include "UnityCG.cginc"
```

No modification should be done here. After this, we can see the definition of the data types that are passed trough the pipeline process.

```
    struct appdata
    {
        float4 vertex : POSITION;
        float2 uv : TEXCOORD0;
    };

    struct v2f
    {
        float2 uv : TEXCOORD0;
        float4 vertex : SV_POSITION;
    };
```

The *appdata* struct represents the mesh vertex data that is passed to the **vertex shader**. In this case, it contains information about the **position** of the vertex, and the **texture coordinates** that corresponds to that vertex. Analogously, the *v2f* struct contains the data that will be passed from the vertex shader to the **fragment shader**, which will render the final image to be displayed.

We now have two functions, which define the steps the vertex and fragmetn shader execute.

```
    v2f vert (appdata v)
    {
        v2f o;
        o.vertex = UnityObjectToClipPos(v.vertex);
        o.uv = v.uv;
        return o;
    }
    fixed4 frag (v2f i) : SV_Target
    {
        fixed4 col = tex2D(_MainTex, i.uv);
        // just invert the colors
        col.rgb = 1 - col.rgb;
        return col;
    }
    ENDCG
```

In this case, the **vertex shader** just calculates in which coordinates of the screen will the vertex be projected, and assigns it a texture coordinate. On the other hand, the **fragment shader** recovers the corresponding color for a given vertex from the texture coordinates previously calculated, and inverts its color. The fragment shader will return, for each vertex, the final color it will have when displayed in the screen.

- Now, we have the basic concepts about how shaders work. We want to visualize the sphere from the inside. In order to do this, in the ´´SubShader" properties, we are going to change the culling process (as aforementioned, OpenGL render process depends on the geometry's normals directions). **We are going to change "Cull Off" to "Cull Front", and delete the other two options "ZWrite" and "ZTest"**. We will also **delete the inverting color line in the fragment shader**, so that the panorama final RGB is not manipulated.

- We can now save the shader and return back to Unity.

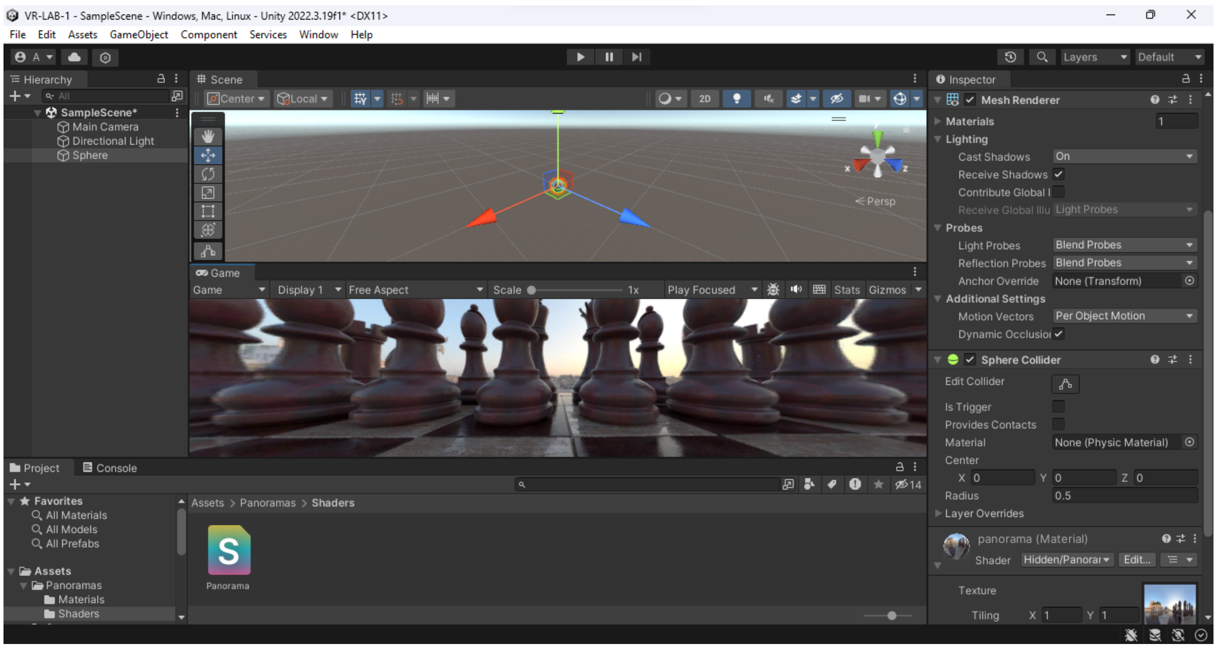
- Although the shader is ready, it is not attached to our **Sphere** yet. Select it and go to the *inspector tab*. When the panorama was attached to the sphere, a new component appeared in the tab, called as the panorama's filename you chose before. To add our brand new shader, **drag the shader file over the component name**.

Right now, you should see part of the scene in the *game tab* (see Figure 1.4). Note: You may see some artifacts, especially near the poles. The equirectangular projection used to represent the panoramas (also called latitude-longitude or spherical coordinates) is non-linear: the same amount of pixel space on the image represents a smaller and smaller area on the sphere as you move from the equator towards the poles. Therefore, the simple mapping we are doing (UV mapping) can only approximate it. This is especially noticeable near the poles. You don't need to worry about this, but if you are interested in fixing it, you can search online. There are multiple approaches to fix this issue by modifying the shader, as this is a very common problem.

Now that our panorama is projected into the sphere, let's make some changes to it to experiment with the shaders:

[MT-0] We are going to try some different shaders. Create a copy of your current shader and add the necessary changes for each of the following modifications: (i) **swap** R, G and B channels, (ii) **grayscale** the scene, and (iii) **flip** the panorama in render time either vertically, horizontally, or both ways.

[MT-1] Unity allows to include text boxes in the scene. Investigate how to add a *canvas* with some text (e.g., in the upper-left corner), and write the latitude and the longitude at which the user is looking at in each frame. Remember that your camera has an orientation property. Think



Mesh Renderer
Materials 1
Lighting
Cast Shadows: On
Receive Shadows: ✓
Contribute Global I: ☐
Receive Global Illu: Light Probes
Probes
Light Probes: Blend Probes
Reflection Probes: Blend Probes
Anchor Override: None (Transform)
Additional Settings
Motion Vectors: Per Object Motion
Dynamic Occlusion: ✓
Sphere Collider
Edit Collider
Is Trigger: ☐
Provides Contacts: ☐
Material: None (Physic Material)
Center: X 0 Y 0 Z 0
Radius: 0.5
Layer Overrides
panorama (Material)
Shader: Hidden/Panora
Texture
Tiling: X 1 Y 1

Fig. 1.4: Our project so far. We have an sphere with an equirectangular panorama projected on it, and a shader that allows it to be watched from inside.

which axes are you interested in.

[OT-0] Shaders can be used for further manipulations. Create a *shader* that adds a parameterizable number of parallels and meridians (lines) overlaid to the image, creating the illusion of a spherical mesh over the scene. Note that you may need to take into account some concepts such as latitude, longitude, radians...

1.4.2. Exploring the scene

We now have a virtual 360º scene based on a equirectangular panorama. However, if we press "Play", we cannot rotate, hence we always see the same scene part. Although when visualizing the content with an HMD this is trivial (Unity takes care of manipulating the camera following the HMD orientation), we may want to visualize it manually when no HMD is available, or just for a quick debug of our application. Thus, we are creating a manual movement script.

- We will go to the *project tab* and create a new folder called "Scripts". Inside it, we will right-click, select "Create" and choose "C Script". Name it so you can identify it will be the script related to the movement.

- Open the script. By default, two functions are included but empty: Start() and Update(). The first one is a function that is executed only once, as the program begins. The latter is executed at the end of each frame. If your application runs in 60 FPS (as many videogames), it means this function will be called 60 times per second.

- We will use WASD keys to rotate the user, simulating a real head rotation. Add a **float** variable for the **rotation speed**; the higher the value, the faster your camera will rotate. We need the program to check frame by frame if any of those keys have been pressed. We are adding two lines in the Update() function:

```
    var direction = new Vector3(-Input.GetAxis("Vertical"),
                                Input.GetAxis("Horizontal"), 0.0f);
```

In this first line, we are calculating the direction of the rotation, given the input keys. If no keys are pressed, direction will be $(0, 0, 0)$, hence no rotation will happen. As any of the inputs is pressed, the vector will change towards that direction. Note that Unity has default binding for inputs: “Horizontal” axis is bound to A-D and left-right arrows, whereas “Vertical” axis is bound to W-S and up-down arrow.

```
    var euler = transform.eulerAngles + direction * speed;
    transform.eulerAngles = euler;
```

With this second line, we are **rotating** the camera in the aforecalculated direction, with the previously fixed speed.

- The rotation script is now ready. You can close the editor and go back to Unity. Although the script is ready, it is not attached to the camera yet. To this, drag the script to the “Main Camera” object in the *hierarchy tab*.

Now we are ready to run our experience. Press the “Play” button in the top of the Unity interface. The program will start running in the *game tab*. Yo can now rotate your camera with the keyboard.

[MT-2] Add a new script to the camera so you can rotate it with the mouse too.

[OT-1] Once your visualizer is ready and you can move, you can try new different content. As most of you know, Google Street View allows a 360º viewing of almost any place on Earth. Search for any source to download equirectangular panoramas from Google Street View and import them into your visualizer.

Finally, in Annex B you can find instructions on how to build your application for a smartphone device, mimicking VR setups (stereoscopic, one image for each eye). This is not required for this lab session, but we highly recommend that you try it so that you can use the cardboard devices that we will lend you later in the course.

1.5. Reporting your results

You should submit a **report** (in .PDF format) including your results for the tasks (results should be clearly labeled following the labels of the tasks they correspond to). There is no required style or fixed structure for your report, you will have to choose how to report the work you did. The quality of the report is important and will influence the final grade, so make sure to invest some effort into its preparation. Here are some guidelines that you need to follow:

- The report should not be longer than **three pages** and should at least address the mandatory tasks. The report file should be named `labXX-mainReport-YYYYYY.pdf`, with `XX` the number of the lab, and `YYYYYY` your NIP.
- Do not include whole snippets of code in the report: You may submit an additional .ZIP file with the code and shaders you wrote, and indicate in the report its function with a couple sentences at most.
- Including in the report the **main difficulties, thoughts, or insights** you had through the whole laboratory, as well as its relation to the lectures of the course, will be positively evaluated.
- If there is any other part of your work that cannot fit in the report (e.g., short videos or sets of a large number of screenshots), you should submit them in a separated, supplementary .ZIP file, **but always adequately referencing it in the main report, so we are aware of the existence of that file**.
- We will not look at submitted items that have not been referenced in the main report.

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.

You should submit your report (and supplementary material) via Moodle, uploading them in the corresponding Moodle task. **Deadline: February 25th, 2025 - 23:59**.

Bibliografía

[1] Jesús Gutiérrez et al. “Introducing UN Salient360! Benchmark: A platform for evaluating visual attention models for 360° contents”. En: *2018 Tenth International Conference on Quality of Multimedia Experience (QoMEX)*. IEEE. 2018, págs. 1-3.

[2] Vincent Sitzmann et al. “Saliency in VR: How do people explore virtual environments?” En: *IEEE Transactions on Visualization and Computer Graphics* (2017).

Appendix A. The graphics pipeline

This section covers a very basic introduction of the graphics pipeline. The graphics pipeline is a fundamental concept in computer graphics that describes the process by which 3D geometry and textures are transformed into a 2D image that can be displayed on a screen. The pipeline is composed of a series of programmable stages, each of which is responsible for performing a specific set of operations on the input data. A simplified overview of these steps can be seen in Fig. A.1

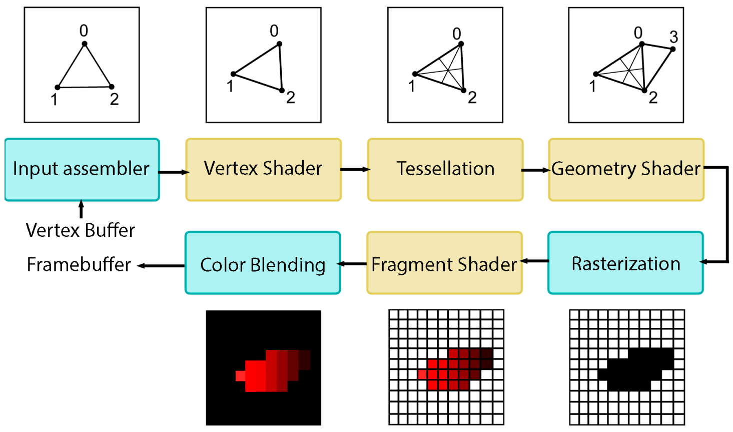


Fig. A.1: Simplified overview of the steps in the graphics pipeline. Stages in green are fixed-function stages. These allow you to tweak their operations using parameters, but the way they work is usually predefined. Stages in yellow are programmable, which means that you can code your own vertex shaders or geometry shaders for instance.

The input data to the graphics pipeline typically consists of 3D models and textures, along with other metadata such as lighting and camera parameters. These data are fed into the pipeline, where they are processed in a series of stages to generate the final output image.

The pipeline is typically divided into several stages, each of which performs a specific function on the input data. These stages include the input assembler, vertex shader, tessellation shaders, geometry shader, rasterization, fragment shader, and color blending. Each stage provides opportunities to optimize performance and improve visual quality, making it critical for graphics developers to understand the pipeline and its individual stages.

Input Assembler: The input assembler collects raw vertex data from the application and assembles it into primitives. It also reads from an index buffer to repeat certain elements without duplicating the vertex data.

Vertex shader: The vertex shader is executed for every vertex in a primitive and applies transformations to the vertex positions. These transformations convert the vertices from model space to screen space. These transformations can include scaling, rotation, and translation, as well as projection onto a 2D plane. In addition to transformations, the vertex shader can also perform other operations on per-vertex data, such as texture coordinates and normals, which are often used in later stages of the pipeline to calculate lighting and texture mapping. After processing each vertex, the vertex shader outputs the transformed vertex position, along with any per-vertex data, to the next stage of the pipeline.

Tessellation shader: Tessellation shaders are responsible for subdividing input primitives into smaller, more detailed primitives based on certain rules. This technique is often used to enhance the appearance of surfaces by adding geometric detail such as bumps, creases, or wrinkles.

Geometry shader: The geometry shader is executed on every primitive and can output additional primitives or discard them altogether. The geometry shader is used to add or remove geometry, modify the geometry or its attributes, and create more complex geometry.

Rasterization: The rasterization stage takes the output from the previous stage and converts it into fragments that represent individual pixels on the screen. Any fragments outside the screen are discarded, and attributes passed from the vertex shader are interpolated across fragments.

Fragment Shader: The fragment shader is executed for every visible fragment and determines its final color and other properties. The fragment shader typically takes as input interpolated data from the vertex shader, such as texture coordinates, normals, and lighting information. It can also use depth values to determine which fragments should be rendered in front of others, which is important for rendering transparent or translucent objects.

Color Blending: The final stage of the pipeline combines the output of the fragment shader with the contents of the framebuffer. This stage determines how to blend the colors of multiple fragments that map to the same pixel, and the final result is written to the framebuffer.

Depending on the requirements of a graphics application, certain stages of the pipeline may be optional. For instance, the tessellation and geometry stages can be omitted if the goal is to render simple geometry. Similarly, the fragment shader stage can be disabled if only depth values are needed, which is often the case when generating shadow maps.

Appendix B. Running on a smartphone

Until now, we have designed and implemented a 360º viewer in Unity. If we want to visualize our scene on a different device, we need to make some adjustments to the project. Each device will have its own documentation to help you with this task; in this course, we will focus on running our projects on a smartphone (Android or iOS devices). We will use the Google Cardboard for Unity plugin so we can visualize the scene in a split screen on the smartphone. This way, the scene will be rendered in stereoscopic 3D (one image for each eye, see Figure B.1) mimicking VR, and then we can use a cardboard-like device to insert our phone and visualize the project (we will provide these later in the course).

B.0.1. Installing the Google Cardboard XR Plugin for Unity

Please follow carefully the instructions in this link to build your project for Android (or iOS): `https://developers.google.com/cardboard/develop/unity/quickstart`. It is **very important that you follow all steps**, otherwise your compilation may fail.

Requirements:

- Unity 2022 LTS with Android SDK. If you have not installed Android SDK when installing Unity, you can update your installation in the Unity Hub by going to Installs → Click on the cogwheel next to your installation (configure) → Add modules.
- Git for installing the Carboard XR Plugin package according to the instructions.

B.0.2. Configuring your scene

The Cardboard XR Unity Plugin contains a sample scene (HelloCardboard) that showcases a minimal working project setup. To import this into your project, when you are still in the Package Manager where you installed the plugin, you will see a *Samples* tab. There you will see the *Hellow Cardboard* example. Click on *import* and it will be imported to the assets in your project. Now if you come back to your project, in the Project tab go to Assets → Samples → Google Cardboard XR Plugin for Unity → Version → Hello Cardboard → Scenes. Double click in the *HelloCardboard* scene and it will open in your editor. It is good practice that you first try to build and test this scene into your phone so that you can make sure everything is working first and test the pipeline before trying to build the scenes you create.

Once you have tested this scene, there are a couple of ways to adapt your own scene for running on your phone:

Option A) In a Cardboard project, you generally use the standard Unity Camera (you will most likely already have one of these in your scene). The XR Plugin automatically adjusts the camera's behavior to render the scene in stereoscopic 3D, mimicking VR. Ensure the Tag of the camera in your scene is set to *MainCamera*. Then click in this camera in the Hierarchy → Add component → XR → Tracked Pose Driver. Now your camera should be ready to behave like a VR camera when you build your scene.

Option B) This one is probably safer since you will take directly the camera from the HelloCardboard scene and import it in your own scene. Open the HelloCardboard sample scene. In the Hierarchy you will see a *Player* component. Drag it to your Assets, and it will become a prefab. Then go back to your scene and drag the prefab in the Assets to your Hierarchy. You can now remove your old camera, and this one you just imported will become your main camera. Make sure it is positioned in the right location of the scene.

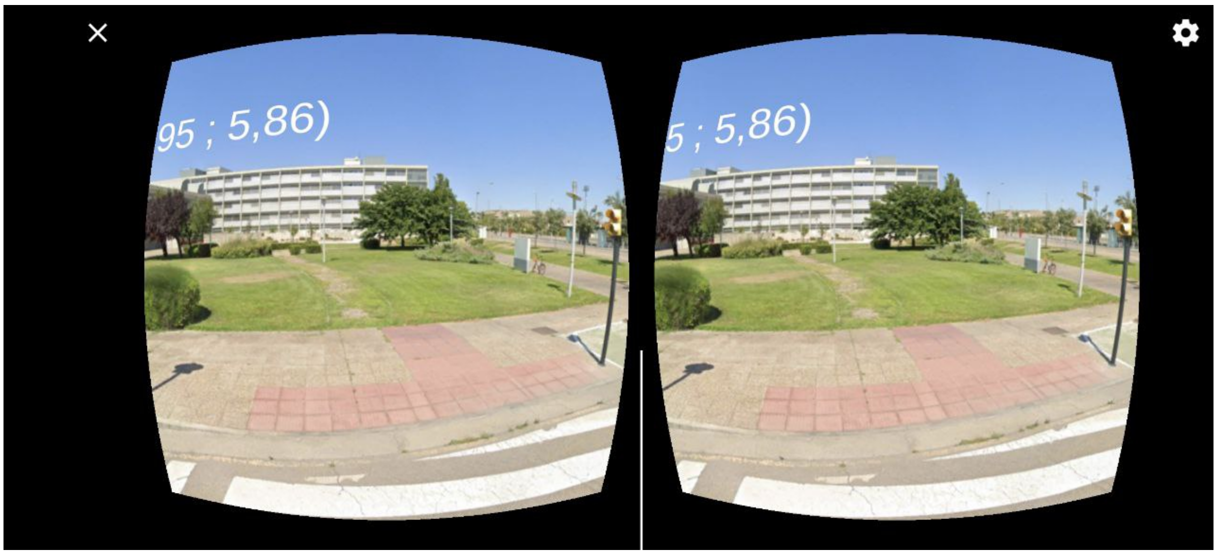


Fig. B.1: Example of a scene from the first lab session running on a smartphone with the Cardboard Plugin splitting the screen for VR visualization.

B.0.3. Building your project

If you carefully follow of the necessary configurations described in the instructions, you should be able to build an APK. Then, you can transfer this APK to your phone and install it. If everything goes well, you should be able to see your scene in your phone. Make sure to start with simple scenes and controlled environments (HelloCardboard, then the scene from the first lab session, then more complex scenes) so you can throubleshoot any problems.

Troubleshooting and tips: These are some issues we encountered even when carefully following the instructions. Other issues you may encounter will probably have a solution if you google for it.

- Double check that you followed all steps and theat all fields in the build settings are correct.
- Set target API level 34 (as opposed to what is specified in the tutorial). If you are using a version of Unity that officially supports a different target API level, Unity might normally give you a warning saying that version 34 is not supported. If the compilation process does not continue, try to suppress the warning by adding *android.suppressUnsupportedCompileSdk=34* to the file *gradleTemplate.properties*.
- If the build fails for any reason, make a clean build.
- Follow all instructions. When selecting a target API level 33 or higher as described in the instructions, Unity may ask to update AndroidSDK. Do it.
- If you APK is building correctly but it does not display right in your phone, try setting the Minimum API level to 31. It seems that lower versions may not be supported anymore (that was the case with my phone).
- It seems that OpenGLES2 is deprecated, if it gives compatibility problems, remove from the Graphics APIs.
- Open Unity with administrator permissions (apparently very important). This is not straightforward because Unity Hub launches Unity for you. Locate both your version's Unity.exe (default at *Program Files/Unity/Hub/Editor/Version/Editor/Unity.exe*) and UnityHub.exe (default at *Program Files/Unity Hub/Unity Hub.exe*). Then, for each of those two: right click → Properties → Compatibility → Change configuration for all users. There, select the *execute program as administrator* option. Make sure you close all Unity and Unity Hub instances after doing this process and open them again. You may need to check the task administrator because Unity Hub tends to linger open. This issue produced many errors while trying to compile the first time that magically disappeared after being able to run Unity as administrator :)