***Introduction to Graphics and Mobile Gaming***

**Unity Lab 3: Part 1**

**Cube maps**

**Issue 1.0**

Contents

[1 Introduction 1](#_Toc36115499)

[2 Custom shaders 1](#_Toc36115500)

[2.1 What is a shader? 1](#_Toc36115501)

[2.2 Unity shaders 1](#_Toc36115502)

[2.3 Simple share code sample 2](#_Toc36115503)

[2.4 Performance 5](#_Toc36115504)

[2.5 Vertex shader 5](#_Toc36115505)

[2.6 What else can you do with vertex shaders? 6](#_Toc36115506)

[2.7 Input and output structures details 7](#_Toc36115507)

[2.8 Performance 7](#_Toc36115508)

[2.9 Graphical example 8](#_Toc36115509)

[2.10 Fragment shader 8](#_Toc36115510)

[2.11 Passing data to the shader 9](#_Toc36115511)

[2.12 Properties block 9](#_Toc36115512)

[2.13 Why do we use custom shaders? 10](#_Toc36115513)

[3 Back to the chess room 11](#_Toc36115514)

[3.1 Cube map 11](#_Toc36115515)

[4 Understanding cube mapping 12](#_Toc36115516)

[4.1 Local cube maps 13](#_Toc36115517)

[4.2 The benefits of using cube maps 14](#_Toc36115518)

# Introduction

This lab is divided into 2 parts:

* Part 1: understanding and creating custom shaders
* Part 2: creating and applying lights and dynamic shadows

# Custom shaders

## What is a shader?

**Vertex shader**

As a simple example, let’s take a primitive, like a cube. A cube is made up of vertices; the vertex shader receives each single one of these vertices and manipulates them. It is up to the shader what to do, but it must, at some point, set a 4D float vector, which represents the final position of the vertex on the screen.

**Fragment shader**

The fragment shader takes care of the colours, textures, and lights that we need to use, and we have applied to our cube. The fragment shader’s main objective is to return a 4D float vector, which represents the final colour of the fragment.

A fragment is the data provided by the 3 vertices for the purpose of drawing each pixel in the triangle.

## Unity shaders

As you may have noticed when creating new materials in Unity, the standard shader is automatically selected for you. However, this can be changed to a mobile shader from the drop-down menu in the Inspector panel. In Unity, you can create and use custom shaders; these allow much more flexibility as well as specificity of use. In this part of the lab, you will be creating and applying your shader for use in the chess room.

In Unity, there are typically two types of shaders:

**Surface shaders**

These are commonly used when shaders are affected by lights and shadows. Unity does the work related to the lighting model, enabling you to write more compact shaders.

**Vertex and fragment shaders**

These are the most flexible shaders. The Unity ShaderLab has more features than vertex and fragment shaders; however, these are the programmable parts of the graphics pipeline. So, this is where all of the shading is done, so it is important to know how to write custom vertex and fragment shaders.

## Simple share code sample

For this lab, you will be implementing vertex and fragment shaders; here follows a very easy example with a step-by-step explanation:

Shader "Custom/ctTextured"

{

 Properties

 {

\_AmbientColor ("Ambient Color", Color) = (0.2,0.2,0.2,1.0)

\_MainTex ("Base (RGB)", 2D) = "white" {}

 }

 SubShader

 {

Pass

{

 CGPROGRAM

 #pragma target 3.0

 #pragma glsl

 #pragma vertex vert

 #pragma fragment frag

 #include "UnityCG.cginc"

 uniform float4 \_AmbientColor;

 uniform sampler2D \_MainTex;

 struct vertexInput

 {

 float4 vertex : POSITION;

 float4 texCoord : TEXCOORD0;

 };

 struct vertexOutput

 {

 float4 pos : SV\_POSITION;

 float4 tex : TEXCOORD0;

 };

 **//** [**Vertex shader**](#vertexShader)**.**

 vertexOutput vert(vertexInput input)

 {

 vertexOutput output;

 output.tex = input.texCoord;

 output.pos = mul(UNITY\_MATRIX\_MVP, input.vertex);

 return output;

 }

 **//** [**Fragment shader**](#fragmentShader)**.**

 float4 frag(vertexOutput input) : COLOR

 {

 float4 texColor = tex2D(\_MainTex, float2(input.tex));

 return \_AmbientColor + texColor;

 }

 ENDCG

}

 }

 Fallback "Diffuse"

}

* **Shader "Custom/ctTextured"**

This defines the path where the shader is displayed in the drop-down window from the inspector panel, followed by the name you want to assign to it.

* **Properties{}**

This is a list of parameters that are going to be visible and accessible in the inspector.

* **Subshaders**

Each shader in Unity consists of a list of subshaders. When Unity renders a mesh, it looks for the shader to use and selects the first subshader that can run on the graphics card. This way, shaders are executed correctly on different graphics cards that support different shader models. This feature is important because GPU hardware and APIs are constantly evolving. For example, you can write your main shader targeting a Mali Midgard GPU to make use of the latest features of OpenGL ES 3.0, while in a separate subshader, write a replacement shader for graphics cards supporting OpenGL ES 2.0 and below.

If Unity cannot find a subshader in the body of the shader, it will read the Fallback statement at the end of the shader code and roll back to the shader defined by it. In this case, it would roll back to the Diffuse built-in shader.

* **Pass**

This block causes the geometry of an object to be rendered one time; a shader can contain more than one pass; multiple passes can be used on hold hardware or to achieve special effects.

* **Compilation directives**

The ‘#pragma’ directive is the method specified by the C standard for providing additional information to the compiler, beyond what is conveyed in the language itself. Each compilation directive must contain at least the directives to compile the vertex and the fragment shader: #pragma vertex name, #pragma fragment name.

The directive #pragma target enables shaders to be compiled into other capability levels. If the shader becomes too large, you get an error of the following type:

“Shader error in 'Custom/MyShader': Arithmetic instruction limit of 64 exceeded; 83

arithmetic instructions needed to compile program;”

If this is the case, you must change from shader model 2.0 to shader model 3.0 by adding the #pragma target 3.0 statement. Shader model 3.0 has a much higher instruction limit.

**#pragma glsl**

This statement allows to pass several varyings from vertex shader to fragment shader. This directive converts Cg or HLSL code into GLSL.

The **#pragma only\_renderers directive**.

Unity supports several rendering platforms such as *gles, gles3, opengl, d3d11, d3d11\_9x, xbox360, ps3,* and *flash*. By default, shaders are compiled to all these platforms unless you explicitly limit this number using the *#pragma only\_renderers* followed by the render APIs you want leaving a blank space between them.

If you are targeting mobile devices only, limit shader compilations to gles and gles3. You must also add the opengl and d3d11 renderers used by Unity Editor*: #pragma only\_renderers gles gles3 [opengl, d3d11]*

* **Include**

It is possible to add include files in the shader to make use of Unity predefined variables and helper functions.

You can see the available includes in *C:\Program Files \Unity\Editor\Data\CGIncludes*. For example, in the include **UnityCG.cginc,** you can find several useful helper functions and macros used in many standard shaders. To use them, declare the include in your shader.

A number of Unity built-in variables are available to shaders. They are located in the include UnityShaderVariables.cginc. You are not required to include this file in your shader because Unity does this automatically.

## Performance

To ***improve performance***, it is sometimes preferable to execute an operation in the CPU and pass the result to the GPU instead of executing it in the vertex shader for every vertex. For example, this is the case of multiplications of matrix uniforms.

**Structures: vertexInput and vertexOutput**

These 2 structures contain and collect the vertex position, respectively, in local space (input) and screen space (output), and the texture coordinates

## Vertex shader

The vertex shader example runs once for each vertex of the geometry. The purpose of the vertex shader is to transform the 3D position of each vertex, given in the local coordinates of the object, to the projected 2D position in screen space and calculate the depth value for the Z-buffer.

In the example, the vertex shader receives as input, the vertex coordinates in local space and the texture coordinates. Vertex coordinates are transformed from local to screen space using the Model View.

Projection matrix UNITY\_MATRIX\_MVP that is a Unity built-in value:

[output.pos = mul(UNITY\_MATRIX\_MVP, input.vertex);](#outputpos)

Texture coordinates are passed to fragment shaders as a varying, but this does not mean that they are not transformed.

The vertexShader “magic” happens inside the *vert* function. This takes an input vertex structure and returns an output vertex structure. As you can notice, in this example, the [texture coordinates](#textureCoordinates) are transferred from input to output with no manipulation, while the vertex is transformed to local screen as seen above.

**Normals** are transformed from object space to world space in a different manner. To guarantee that the normal is still normal to the triangle after a non-uniform scaling operation, it must be multiplied by the transpose of the inverse of the transformation matrix. To apply the transpose operation, you flip the order of factors in the multiplication. The inverse of the local to world matrix is the built-in World2Object Unity matrix. It is a 4x4 matrix, so you must build a four-component vector from the three-component normal input vector.

float4 normalWorld = mul(float4(input.normal, 0.0), \_World2Object);

When building the four-component vector, you add a zero as the fourth component. This is necessary to handle vector transformation correctly in the fourth dimensional space, while for coordinates, the fourth component must be a unit.

## What else can you do with vertex shaders?

Most of the graphics effects are implemented in the fragment shader, but you can also do some effects in the vertex shader. **Vertex Displacement Mapping**, also known as **Displacement Mapping,** is a well-known technique enabling you to ***deform a polygonal mesh using a texture to add surface detail*** *(this is what we have seen in the* [*advanced section of lab 2*](file:///C%3A%5CUsers%5Cmasvla01%5CDocuments%5CAUP%20unity%20project%5CLab3%5CLab%5CLab%202.docx) *when defining how materials work)*, for example, in terrain generation using height maps. To have access in the vertex shader to this texture, also known as displacement map, you must add the pragma directive #pragma target 3.0 because it is only available in shader model 3.0.

In the vertex shader, **you can also animate vertices using “procedural animation”** techniques. You can use the time variable in shaders enabling you to modify the vertex coordinates as a function of time. Mesh skinning is another type of functionality implemented in the vertex shader. Unity uses this to animate the vertices of the meshes associated with character skeletons.

## Input and output structures details

struct vertexInput

{

 float4 vertex : POSITION;

 float4 tangent : TANGENT;

 float3 normal : NORMAL;

 float4 texcoord : TEXCOORD0;

 float4 texcoord1 : TEXCOORD1;

 fixed4 color : COLOR;

};

**Input**

In this example, we only declared the vertex attributes position and texture coordinates, and you can define more attributes; here is the full vertex structure (on side)

More attributes can be defined with the semantics of this structure, a semantic is a string attached to a shader input or output that provides information about the use of a parameter. Incorrect semantics will generate errors.

## Performance

For **performance**, only specify the parameters in the input structure that you strictly require. Unity has some predefined input structures for the most common cases of input parameter combinations: ***appdata\_base, appdata\_tan, and appdata\_full***. These are described in the UnityCG.cginc includefile. The previous vertex input structure example corresponds to appdata\_full. In this case, you are not required to declare the structure, only declare the include file.

**Output**

Vertex shader output is defined in an output structure that must contain the vertex transformed coordinates.

struct vertexOutput

{

 float4 pos : SV\_POSITION;

 float4 tex : TEXCOORD0;

 float4 texSpecular : TEXCOORD1;

 float3 vertexInWorld : TEXCOORD2;

 float3 viewDirInWorld : TEXCOORD3;

 float3 normalInWorld : TEXCOORD4;

 float3 vertexToLightInWorld : TEXCOORD5;

 float4 vertexInScreenCoords : TEXCOORD6;

 float4 shadowsVertexInScreenCoords : TEXCOORD7;

};

This code lists the semantics supported by Unity for the output vertex structure.

The semantic **SV\_POSITION** defined the transformed vertex coordinates.

More textures and vertexes are also passed to the fragment shader calling **TEXCOORD**n.

TEXCOORD0 is typically reserved for UVs and TEXCOORD1 for light map UVs, but technically, you can send anything from TEXCOORD0 to TEXCOORD7 to the fragment shader.

## Graphical example

Everything you send from the vertex shader to the fragment shader is linearly interpolated by default. For every pixel in the triangle defined by the vertices V1, V2, and V3, the rasterizer, located in the graphic pipeline between vertex and fragment shaders, calculates the pixel coordinates as a linear interpolation of the vertices coordinates using the barycentric coordinates λ1, λ2, and λ3.



The following diagram shows the results of colour interpolation in a triangle with vertex colours red, green, and blue.



The same interpolation is applied to any varying passed from the vertex to the fragment shader. This is a very powerful tool because there is a hardware linear interpolator. For example, if you have a plane and you want to apply a colour as a function of the distance to the centre C, you pass the coordinate of the centre C to the vertex shader, calculate the squared distance from the vertex to C, and pass that magnitude to the fragment shader. The value of the distance is automatically interpolated for you in every pixel of every triangle.

##  Fragment shader

The fragment shader from the code example is pretty simple; it takes the input structure and returns a float4.

The fragment shader is the graphics pipeline stage after primitive rasterization.

For each sample of the pixels covered by a primitive, a fragment is generated. The fragment shader code is executed for each generated fragment. There are many more fragments than vertices, so you must take care about the number of operations performed in the fragment shader.

In the fragment shader, you can access the fragment coordinates in the windows space among other values that contains all interpolated per-vertex output values from the vertex shader.

In the shader example, the fragment shader receives the interpolated texture coordinates from the vertex shader and performs a texture lookup to obtain the colour at these coordinates. It combines this colour with the ambient colour to produce the final output colour. From the declaration of the fragment shader float4 frag(vertexOutput input) : COLOR, it is clear that it is expected to produce the fragment colour.

##  Passing data to the shader

Anything you include in the pass block is available to both vertex and fragment shader. Whatever you declare as *uniform* can be considered as a constant and so not modifiable in the shader.

You can supply this uniform to the shader in the following ways:

* Using the Properties block
* Programmatically from a script

##  Properties block

When you use the **properties block**, as seen before, the properties you set will be visible in the inspector panel, which means you effectively passing data to your shader; this can be really useful during the development process since it allows you to change data interactively and see the results at runtime

You can put the following types of variables in the Properties block:

* Float
* Color
* Texture 2D
* Cube map
* Rectangle
* Vector

The Properties block is not a useful way of passing data if, for example, data is required from a previous calculation or data is required to be passed at specific point in time.

An alternative method of passing data to the shaders is programmatically from a **script**.

A material class already has some useful methods that can be used to pass data to the shader; here is a list of the most common ones:

SetColor (propertyName: string, color: Color);

SetFloat (propertyName: string, value: float);

SetInt (propertyName: string, value: int);

SetMatrix (propertyName: string, matrix: Matrix4x4);

SetVector (propertyName: string, vector: Vector4);

SetTexture (propertyName: string, texture: Texture);

The following code shows how matrices and textures are sent to the shader by means of materials contained in the list shwMats.

// Called before object is rendered.

public void OnWillRenderObject()

{

 // Perform different checks.

 ...

 CreateShadowsTexture();

 // Set up shadows camera shwCam.

 ...

 // Pass matrices to the shader

 for(int i = 0; i < shwMats.Count; i++)

{

 shwMats[i].SetMatrix("\_ShwCamProjMat", shwCam.projectionMatrix);

 shwMats[i].SetMatrix("\_ShwCamViewMat", shwCam.transform.worldToLocalMatrix);

}

 // Render shadows texture

 shwCam.Render();

 for(int i = 0; i < shwMats.Count; i++)

{

 shwMats[i].SetTexture( "\_ShadowsTex", m\_ShadowsTexture );

}

 s\_RenderingShadows = false;

}

Good, now you should be able to understand how a simple custom shader works. If you are interested in knowing more, please check out the *“arm guide for unity developers”* provided in the folder for this lab.

##  Why do we use custom shaders?

Custom shaders allow us to obtain much better performance when rendering the scene on a mobile device.

For example, in the next part of the lab, we will apply shaders for dynamic shadows and other shaders for anything else. This is done so that everything that will not move or change during the gameplay is rendered only once at the beginning, while all the elements that are moved will need their shadows to be rendered again.

This process allows optimization, better performance, and faster processing.

# Back to the chess room

Now that we have covered the theory of Shaders, you will add them to the project and look through the code.

## Cube map

This project uses a cube map (which is in the lab3 scripts folder) to render the shadows created when the light from the PointLight game object enters the chess room through the windows. The transparency of the walls of the room has been rendered in the alpha channel of the cube map.

The info stored in the alpha channel is like a map of the regions where the light can enter and reach the chess room. In this case, the light can come through the windows in the wall and from the skylight.

At runtime, the shaders that implement this technique use cube maps to render the shadows from the static geometry (windows).

The fragment-to-light vector is used to retrieve the texel from the cube map. If the retrieved alpha value is zero, it means that in the direction of the fragment-to-light vector, there is either no geometry or the geometry is totally transparent, i.e., the fragment can be reached by the light, and it is lit.

If the retrieved alpha value is 1, it means that in the direction of the fragment-to-light vector, there is an opaque geometry, i.e., the fragment cannot be reached by the light, and it is shadowed. Any intermediate value of alpha will modulate the intensity of the light that reaches the fragment.

# Understanding cube mapping

Cube mapping is a simple method to map the environment by using the six faces of a cube. The environment is projected onto the cube faces and stored; this is done by rendering the scene six times from the viewpoint.



Let’s take the left image as an example: The viewpoint in the lower scene on the left and on the right is the projection of the scene from the viewpoint registered on the cube map, and the top part of the images shows the net of the cube mapping as seen from that viewpoint.

The advantage is that it is a simpler process than the previous method (sphere mapping), similar to ray tracing but much more computationally efficient.

When inserting a new object in the scene or moving an existing one, the reflections and/or the cube map must be re-rendered. Cube maps are exactly how the skybox works.

## Local cube maps

**There are two main kinds of cube maps:** local and infinite cube maps.

***Infinite cube maps*** are used to represent the lighting from a distant environment; the position of the cube map is not relevant and they are mainly used for outdoor lighting such as skyboxes.

***Local cube maps***are used to represent lighting for a finite local environment; the position of the cube map in now relevant. The lighting is correct only at the location where the cube map was created, and finally, local correction must be applied to adapt to the local environment.



To render the chess room environment, we use local cube maps. Local cube maps are NOT in the same position as the viewpoint.

When we are looking at an object, we can’t just let the cube map use the same vector as the one from us to the object to render the view because due to the different position, this vector would point to something different from what we look at. This means that to render the correct scene and obtain the correct vector from the box map to the object, we need a ***local correction***.

This is the process explained in the images above.

In order to find the correct vector, we have to find where the vector from the viewpoint intersects with the boundaries, and from here, we have to find the cube map and so obtain the “intersection-cube map” vector. This is not easy since boundaries can easily change a lot from case to case; to overcome this problem, we have to approximate and introduce a ***bounding box*** that surrounds the boundaries of the scene. Once the intersection point is found, we can easily obtain the correct vector we need to pass to the cube map to render the scene for the viewpoint.

**Lastly,** a cube map can be used to store any kind of information; in this lab, we will use cube maps to render shadows and reflections.

## The benefits of using cube maps

The static nature of the local cube map does have a positive impact in that it allows for faster and higher quality rendering.

For example,

* shadows based on local cube maps are 1.3–1.5 times faster than shadow mapping.
* Reflections based on local cube maps are up to 2.8 times faster than planar reflections rendered at runtime.

The fact that we use the same texture every frame guarantees high-quality shadows and reflections with no pixel instabilities, which are present with other runtime rendering techniques.

Finally, as there are only read operations involved when using static local cube map, the bandwidth used is halved. This feature is especially important in mobile devices where bandwidth must be carefully balanced at runtime.

The conclusion here is that when possible, use rendering techniques based on local cube map. When combining with other techniques, they allow us to achieve higher quality at very low cost.

***This lab continues in the Lab 3-part 2 document with more interactive tasks***