4. APPLICATION

To run the application, program needs another class wich will be a subclass of “Window” and “InputSystem”. So, the application class can create the window and get input data from the “InputSystem”.

At first, to define this subclass, “AppWindow” is created. This class will handle the window creation and updates.

After that, in the “main.cpp”, we call the “broadcast()” function to show the scene.

4.1 Clear Window Creation

First thing to do is creating a clear window. Let’s have a look at the scripts of “AppWindow” class for this: (Majumder & Gopi, 2018)

```
/* AppWindow.h */
#pragma once
#include "Window.h"
#include "GraphicsEngine.h"
#include "SwapChain.h"
#include "DeviceContext.h"

class AppWindow: public Window
{
public:
	AppWindow();
	~AppWindow();

	// Inherited via Window
	virtual void onCreate() override;
	virtual void onUpdate() override;
	virtual void onDestroy() override;
private:
	SwapChain * m_swap_chain;
};
```

Figure 4.1 Creating the AppWindow class and AppWindow.h

Header file of the “AppWindow” contains the inherited functions from “Window” class. Which “onCreate”, “onUpdate” and “onDestroy”.

And a “SwapChain” pointer as the private member.

```
/* AppWindow.cpp */
#include "AppWindow.h"

AppWindow::AppWindow()
{
}

AppWindow::~AppWindow()
{
}

void AppWindow::onCreate()
{
	Window::onCreate();
	GraphicsEngine::get()->init();
	m_swap_chain=GraphicsEngine::get()->createSwapChain();

	RECT rc = this->getClientWindowRect();
	m_swap_chain->init(this->m_hwnd, rc.right - rc.left, rc.bottom - rc.top);

}

void AppWindow::onUpdate()
{
	Window::onUpdate();
	GraphicsEngine::get()->getImmediateDeviceContext()->clearRenderTargetColor(this-
>m_swap_chain,
		0.125f, 0.025f, 0.125f, 1);
	m_swap_chain->present(true);
}

void AppWindow::onDestroy()
{
	Window::onDestroy();

	m_swap_chain->release();
	GraphicsEngine::get()->release();
}
```

Figure 4.2 AppWindow.cpp for the clear window

The constructor and destructor are responsible for initializing and cleaning up the AppWindow object. They don't have specific implementations in this code snippet, so they inherit the behavior from the base class “Window”.

“onCreate” is called when the window is created. It calls the “onCreate” method of the base class “Window” at first and then initializes the graphics engine and creates a swap chain.

“onUpdate” is called during the application's update loop. It first runs the “onUpdate” method of the base class Window. Then, it uses the graphics engine to clear the render target with a specific color (in this case, a shade of purple) and presents the content of the back buffer to the screen using the swap chain.

“onDestroy” is called when the window is being destroyed. Firstly, it invokes the “onDestroy” method of the base class Window. Then, it releases the swap chain and cleans up the graphics engine.

As the final result, in Figure 4.3, the window is created successfully:

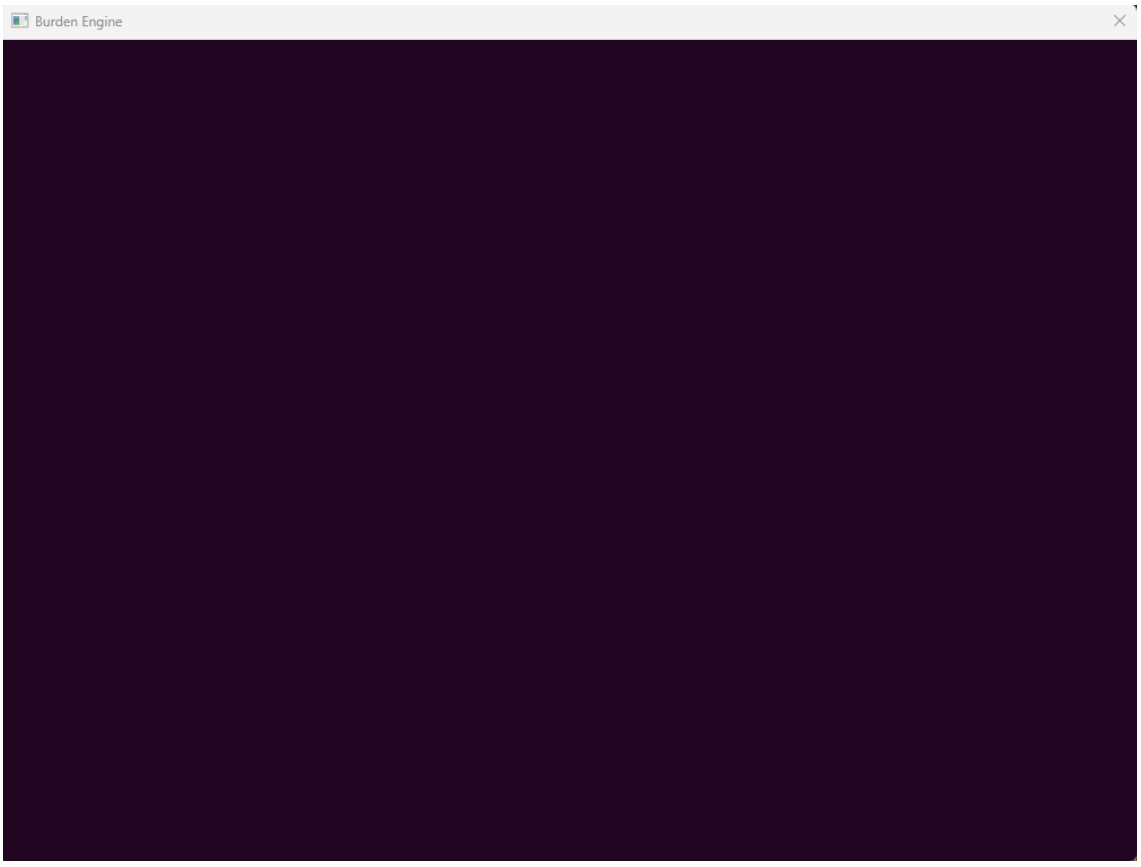

Figure 4.3 Clear window

4.2 Drawing Triangle

Most of the objects are drawed from triangles in computer graphics. So, drawing triangle is an important functionality.

Let’s have a look at the changes in the scripts of “AppWindow” class:

To draw triangles we need three vertices and a “VertexBuffer” object to hold the information of these vertices.

```
VertexBuffer* m_vertex_buffer;
```

Figure 4.4 Defining vertex buffer as a member variable

So, the “m_vertex_buffer” is defined as “VertexBuffer” pointer as a private member of the class.

In the source file, at first a vertex struct must be defined as:

```
struct vec3
{
	float x, y, z;
};

struct vertex
{
	vec3 position;
};
```

Figure 4.5 Pre-defined structs

```
	vertex list[] =
	{
		//X - Y - Z
		{-0.5f,-0.5f,0.0f}, // POS1
		{0.0f,0.5f,0.0f}, // POS2
		{ 0.5f,-0.5f,0.0f}
	};

	m_vb=GraphicsEngine::get()->createVertexBuffer();
	UINT size_list = ARRAYSIZE(list);

	GraphicsEngine::get()->createShaders();

	void* shader_byte_code = nullptr;
	UINT size_shader = 0;
	GraphicsEngine::get()->getShaderBufferAndSize(&shader_byte_code, &size_shader);

	m_vb->load(list, sizeof(vertex), size_list, shader_byte_code, size_shader);
```

Figure 4.6 Vertex list and shader creation

An array list is defined in the “onCreate”, after the creation of swap chain. Containing three vertices, each represented by a structure with x, y and z coordinates.

“m_vertex_buffer” is created as a vertex buffer using the “createVertexBuffer” method from the “GraphicsEngine” class.

Shaders are created using the “createShaders” method from the “GraphicsEngine” class.

The vertex data from the list array is loaded into the vertex buffer. The load method of the vertex buffer is called with the following parameters:

“list” is pointer to the vertex data array, “sizeof(vertex)” is size of each vertex structure in bytes, “size_list” is number of vertices in the array, “shader_byte_code” is pointer to the shader byte code and “size_shader” is size of the shader byte code.

In the “onUpdate” function, after the “clearRenderTargetColor”:

```
	//SET VIEWPORT OF RENDER TARGET IN WHICH WE HAVE TO DRAW
	RECT rc = this->getClientWindowRect();
	GraphicsEngine::get()->getImmediateDeviceContext()->setViewportSize(rc.right - rc.left,
rc.bottom - rc.top);
	//SET DEFAULT SHADER IN THE GRAPHICS PIPELINE TO BE ABLE TO DRAW
	GraphicsEngine::get()->setShaders();
	//SET THE VERTICES OF THE TRIANGLE TO DRAW
	GraphicsEngine::get()->getImmediateDeviceContext()->setVertexBuffer(m_vb);

	// FINALLY DRAW THE TRIANGLE
	GraphicsEngine::get()->getImmediateDeviceContext()->drawTriangleList(m_vb-
>getSizeVertexList(), 0);
	m_swap_chain->present(true);
```

Figure 4.7 Modifications on the onUpdate method

And in the “onDestroy”, all newly created objects are deallocated.

The first shader code, that program needs, can be a simple “fx” file:

```
float4 vsmain( float4 pos : POSITION ) : SV_POSITION
{
    return pos;
}
```

Figure 4.8 Simple shader program

“vsmain” stands for “VertexShaderMain”. It takes one argument as a float vector which contains four elements. And returns the argument.

So, in Figure 4.9, we can see the final result.

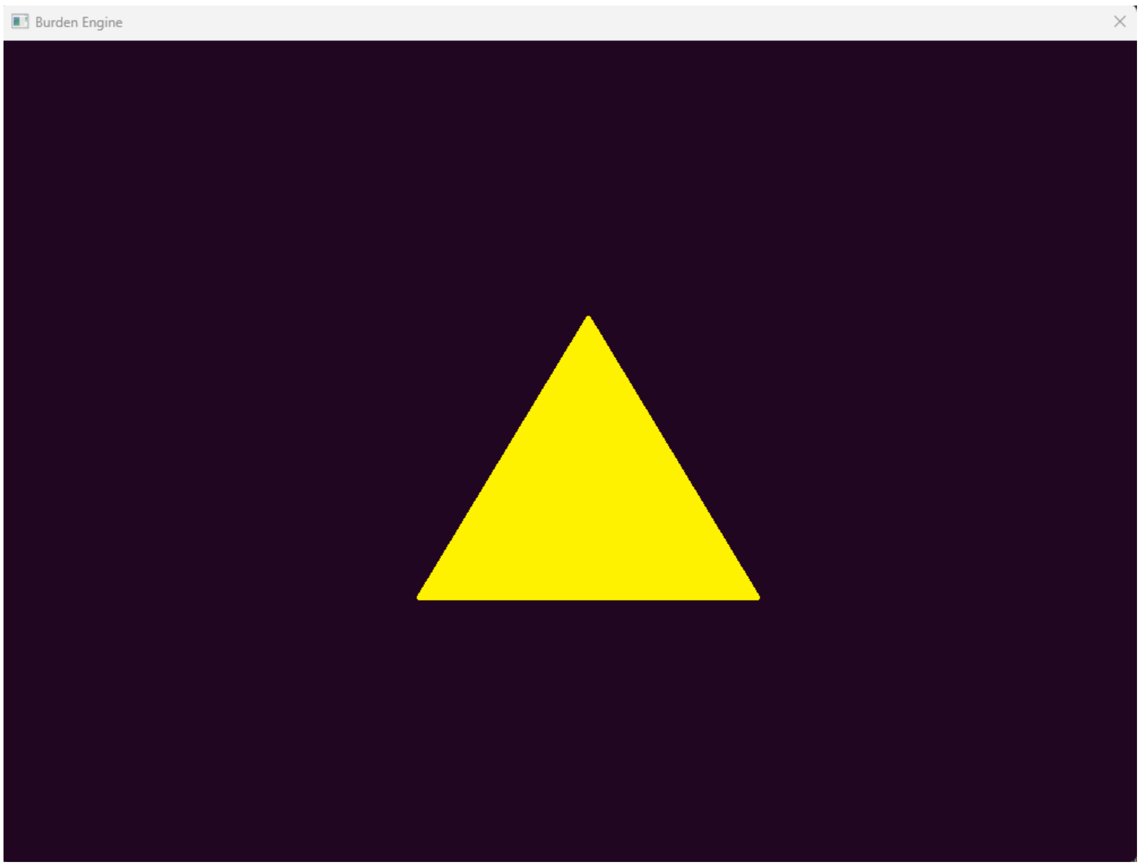

Figure 4.9 Drawing triangle

4.3 Drawing Quad

Quads are drawn by two triangles:

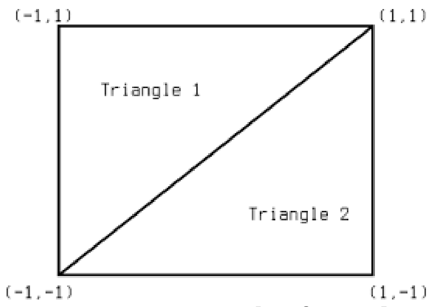

Figure 4.10 Two triangles formed a quad

There are two ways to draw a quad using two triangles: triangle list and triangle strip.

Triangle lists, hold every data of triangles' vertices. Drawing them one by one.

Triangle strip can be used when triangles have two common vertices. So strip can be used when closed shapes or objects like quads, cubes, teapots etc.

In this situation, triangle strip is used, which can be seen in the source file of the "AppWindow".

At first, the four vertices of the quad in vertices list is defined:

```
vertex list[] =
{
        //X - Y - Z
        {-0.5f,-0.5f,0.0f}, // POS1
        {-0.5f,0.5f,0.0f}, // POS2
        { 0.5f,-0.5f,0.0f },// POS2
        { 0.5f,0.5f,0.0f}
};
```

Figure 4.11 Updating the vertex list

And in the "onUpdate", "drawTriangleList" method is changed with:

```
        GraphicsEngine::get()->getImmediateDeviceContext()->drawTriangleStrip(m_vb-
>getSizeVertexList(), 0);
```

Figure 4.12 Calling the draw method

The final result can be seen in Figure 4.4:

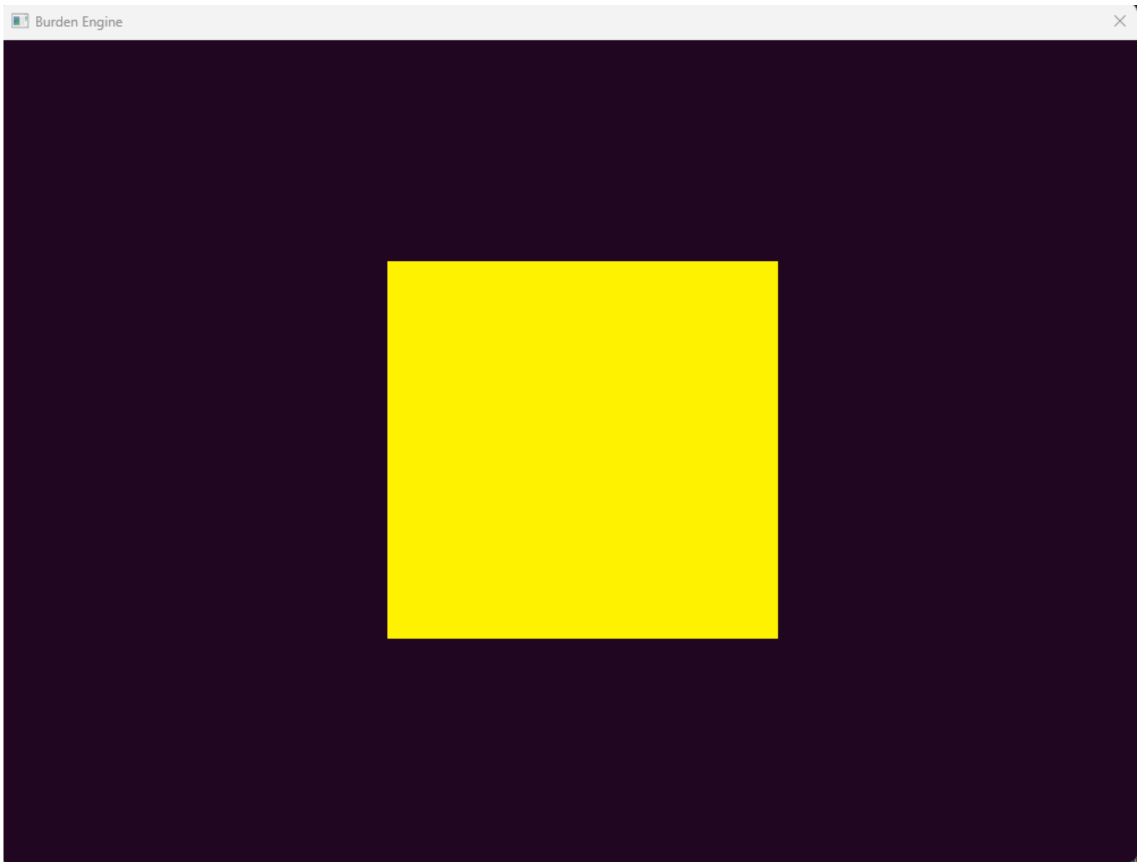

Figure 4.13 Drawing quad

4.4 VertexShader

To create vertex shader, DirectX library needs shader scripts written in HLSL. So, “VertexShader.hlsl” is created: (Pharr, Jakob, & Humphreys, 2023)

```
float4 vsmain(float4 position : POSITION) : SV_POSITION
{
	if (position.y > 0 && position.y < 1)
	{
		position.x += 0.25f;
	}


	if (position.y > 0 && position.y < 1 && position.x > -1 && position.x < 0)
	{
		position.y -= 0.25f;
	}
	return position;
}
```

Figure 4.14 VertexShader.hlsl

“vsmain” defines a function that takes a “float4” input parameter named “position” representing a 3D position and returns a “float4” value, which is the final screen space position (“SV_POSITION”).

The function contains two conditional statements based on the components of the input position.

First statement checks if the y component of the position is greater than 0 and less than 1, then add 0.25 to the x component of the position.

Second statement checks if the y component of the position is greater than 0 and less than 1 and the x component is greater than -1 and less than 0, then subtract 0.25 from the y component of the position.

Finally the position data is returned.

To compile HLSL files, “compileVertexShader” method is used, which explained previously.

Now let’s have a look at the “AppWindow” class:

We need to define a vertex shader class in header file by simply adding:

```
VertexShader* m_vertex_shader;
```

Figure 4.15 Defining vertex shader as a member variable

Now, let’s look at the source file. In the “onCreate” method:

```
m_vb = GraphicsEngine::get()->createVertexBuffer();
UINT size_list = ARRAYSIZE(list);

GraphicsEngine::get()->createShaders();

void* shader_byte_code = nullptr;
size_t size_shader = 0;
GraphicsEngine::get()->compileVertexShader(L"VertexShader.hlsl", "vsmain",
&shader_byte_code, &size_shader);

m_vs = GraphicsEngine::get()->createVertexShader(shader_byte_code, size_shader);
m_vb->load(list, sizeof(vertex), size_list, shader_byte_code, size_shader);

GraphicsEngine::get()->releaseCompiledShader();
```

Figure 4.16 Modifications on the onCreate method

As can be seen from comparing this code to previous, “compileVertexShader” is used. It simply compiles a vertex shader by specifying the HLSL file (“VertexShader.hlsl”), the entry point (“vsmain”) and retrieves the compiled bytecode and its size.

“createVertexShader” creates a vertex shader object using the compiled shader bytecode and its size.

“load” loads the vertex data (“list”) into the previously created vertex buffer (“m_vb”). It also associates the vertex shader bytecode with the vertex buffer.

Finally, releasing the compiled shader.

In the “onUpdate” method:

```
//SET DEFAULT SHADER IN THE GRAPHICS PIPELINE TO BE ABLE TO DRAW
GraphicsEngine::get()->setShaders();
GraphicsEngine::get()->getImmediateDeviceContext()->setVertexShader(m_vs);
```

Figure 4.17 Setting the vertex shader

Default shaders set in the graphics pipeline and specifically sets the vertex shader (m_vertex_shader). It's called before setting vertex buffer so the vertices get the returned position data from shader before set.

And the final result can be seen in Figure 4.18:

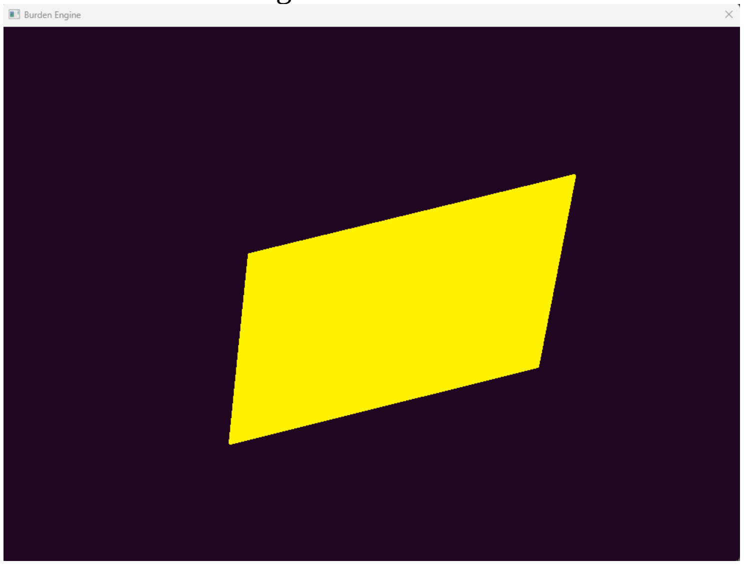

Figure 4.18 Quads vertices position is changed through vertex shader

4.5 PixelShader

As the vertex shader, "PixelShader.hlsl" is created: (Pharr, Jakob, & Humphreys, 2023)

```
struct PS_INPUT
{
	float4 position : SV_POSITION;
	float3 color : COLOR;
};


float4 psmain(PS_INPUT input) : SV_TARGET
{
	return float4(input.color,1.0f);
}
```

Figure 4.19 PixelShader.hlsl

At first, a structure named "PS_INPUT" defined to represent the input to the pixel shader. It has two members: "position" as "float4" represents the position with semantic "SV_POSITION" and "color" as "float3" represents the color with semantic "COLOR". Vertix transforms that done previously are not needed. So "VertexShader.hlsl" is modified according to this and "PixelShader.hlsl":

```
struct VS_INPUT
{
	float4 position : POSITION;
	float3 color : COLOR;
};

struct VS_OUTPUT
{
	float4 position : SV_POSITION;
	float3 color : COLOR;
};


VS_OUTPUT vsmain(VS_INPUT input)
{
	VS_OUTPUT output = (VS_OUTPUT)0;

	output.position = input.position;
	output.color = input.color;

	return output;
}
```

Figure 4.20 VertexShader.hlsl

The "VS_INPUT" structure encapsulates the input data that the vertex shader processes, has two members: "position" is a 4D vector ("float4") representing the vertex position with the semantic "POSITION". "color" is a 3D vector ("float3") representing the color associated with the vertex, labeled with the semantic "COLOR".

Conversely, the "VS_OUTPUT" structure defines the data that the vertex shader generates and forwards to subsequent stages. It contains: "position" which is again a 4D vector ("float4") denoting the transformed vertex position, marked with the semantic "SV_POSITION" and again "color" as a 3D vector ("float3") carrying the color information to be used in subsequent shading computations, associated with the semantic "COLOR".

"vsmain" function represents the core logic of the vertex shader. It takes an instance of "VS_INPUT" as input and produces an instance of "VS_OUTPUT". Inside the main, a new "VS_OUTPUT" variable, named output, is initialized.

The position in the output is set equal to the input position, directly inheriting the vertex coordinates. The color in the output is copied from the input, maintaining the original color information. The function concludes by returning the populated output structure.

And now, let's look at the AppWindow class:

Initialization and usage of pixel shader is similar to vertex shader. At first, a "PixelShader" object is created in the header file as:

```
	PixelShader* m_pixel_shader;
```

Figure 4.21 Defining pixel shader as a member variable

After the declaration, let's examine the source file:

```
        vertex list[] =
        {
                //X - Y - Z             R G B
                {-0.5f,-0.5f,0.0f,      0,0,0}, // POS1
                {-0.5f,0.5f,0.0f,       1,1,0}, // POS2
                { 0.5f,-0.5f,0.0f,      0,0,1},// POS2
                { 0.5f,0.5f,0.0f,       1,1,1}
        };

        GraphicsEngine::get()->compilePixelShader(L"PixelShader.hlsl", "psmain",
&shader_byte_code, &size_shader);
        m_ps = GraphicsEngine::get()->createPixelShader(shader_byte_code, size_shader);
        GraphicsEngine::get()->releaseCompiledShader();
```

Figure 4.22 Updating vertex list and compiling PS

In "onCreate" method, after the releasing the compiled vertex shader, pixel shader must be compiled, created and released, respectively. And now, color info must be defined in the vertex list.

```
        GraphicsEngine::get()->getImmediateDeviceContext()->setPixelShader(m_ps);
```

Figure 4.23 Setting the pixel shader

And in "onUpdate", pixel shader must be set after the setting of vertex shader.

```
        m_vs->release();
        m_ps->release();
```

Figure 4.24 Destroying the shaders

Finally, in "onDestroy", both of the shader objects are released.

Final result can be seen in the Figure 4.25.

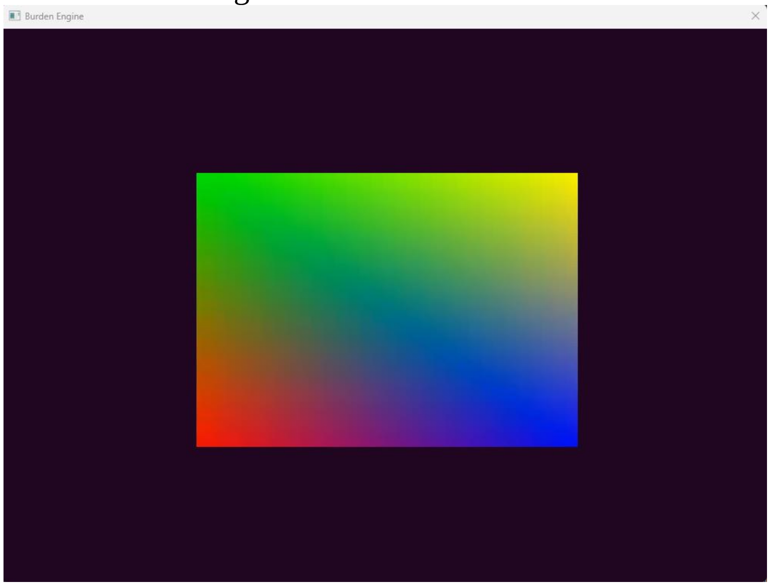

Figure 4.25 Quads vertices' color data is changed through pixel shader

4.6 Basic Animations and Transform Matrix

At first, the background color is changed to grey to see more clearly. This modification done by changing the RGB arguments of "clearRenderTargetColor" method from (0.125, 0.025, 0.125) to (0.25, 0.25, 0.25).

In this section, both vertex transformations and color changes explained together. Vertex transformation is operated by vertex shader and color changes are operated by pixel shaders. (PardCode, 2022)

We can add three animations with vertex transformations: transform, rescale and rotation.

Let’s check the HLSL scripts first:

```
struct VS_INPUT
{
	float4 position : POSITION;
	float4 position1 : POSITION1;
	float3 color : COLOR;
	float3 color1 : COLOR1;
};

struct VS_OUTPUT
{
	float4 position : SV_POSITION;
	float3 color : COLOR;
	float3 color1 : COLOR1;
};

cbuffer constant : register(b0)
{
	float m_angle;
};

VS_OUTPUT vsmain(VS_INPUT input)
{
	VS_OUTPUT output = (VS_OUTPUT)0;

	output.position = lerp(input.position, input.position1, (sin(m_angle) + 1.0f) / 2.0f);
	output.color = input.color;
	output.color1 = input.color1;
	return output;
}
```

Figure 4.26 VertexShader.hlsl

As can be seen in the code block, vertex shader input is defined with two position and color information to apply the animations to the rendering object’s vertices. One is the starting data, which is rendered first.

“cbuffer constant” declares a constant buffer named constant bound to register “b0”. This buffer holds a single float variable “m_angle”, which will be used in shader calculations. “b0” is a register space address in the context of HLSL. In graphics programming, registers are small, fast storage locations available to shaders for storing and accessing constant data. The “b0” register space is often used for constant buffers.

Inside the main function, first line initializes the output structure with zero values. “lerp” performs linear interpolation between “input.position” and “input.position1” based on the sine of “m_angle”. The result is assigned to the output position. And color data directly assigns the input color to the output color. Finally, returns the transformed vertex attributes in the “VS_OUTPUT” structure.

```
struct PS_INPUT
{
	float4 position : SV_POSITION;
	float3 color : COLOR;
	float3 color1 : COLOR1;
};

cbuffer constant : register(b0)
{
	float m_angle;
};

float4 psmain(PS_INPUT input) : SV_TARGET
{
	return float4(lerp(input.color, input.color1, (sin(m_angle) + 1.0f) / 2.0f),1.0f);
}
```

Figure 4.27 PixelShader.hlsl

In "PixelShader.hlsl", like in the vertex shader, input struct is defined with two color information. Again, a constant buffer is registered at "b0" that only holds "m_angle".

Main function is directly returning the lerp from "color" to "color1" based on the sine of "m_angle".

Now let's look at the "AppWindow" class:

In the header file, constant buffer is defined. And three other variables defined to send elapsed time data to shaders to run animations accordingly. And an angle variable is defined again for the same reason:

```
	ConstantBuffer* m_cb;

	unsigned long m_old_time = 0;
	float m_delta_time = 0;
	float m_angle = 0;
```

Figure 4.28 Defining constant buffer and other member variables related to time and angle

In the source file, first changes made in the vertex list. To apply linear interpolation, function needs start and end data. So, position and color data is defined two for each:

```
	vertex list[] =
	{
		//X - Y - Z
		{-0.5f,-0.5f,0.0f,    -0.32f,-0.11f,0.0f,   0,0,0,  0,1,0 },
		{-0.5f,0.5f,0.0f,     -0.11f,0.78f,0.0f,    1,1,0,  0,1,1 },
		{ 0.5f,-0.5f,0.0f,     0.75f,-0.73f,0.0f,   0,0,1,  1,0,0 },
		{ 0.5f,0.5f,0.0f,      0.88f,0.77f,0.0f,    1,1,1,  0,0,1 }
	};
```

Figure 4.29 Updating vertex list to apply lerp

```
__declspec(align(16))
struct constant
{
	float m_angle;
};
```

Figure 4.30 A data type defined to send to constant buffer

A data struct named “constant” is defined and declared 16 bytes of space. This type will be send to constant buffer to used in the shaders.

```
	constant constant_obj;
	constant_obj.m_angle = 0;

	m_cb = GraphicsEngine::get()->createConstantBuffer();
	m_cb->load(&constant_obj, sizeof(constant));
```

Figure 4.31 Creating the constant buffer

In the “onCreate” method, constant buffer is created with constant object. Angle is set to zero.

```
	unsigned long new_time = 0;
	if (m_old_time) new_time = ::GetTickCount() - m_old_time;
	m_delta_time = new_time / 1000.0f;
	m_old_time = ::GetTickCount();

	m_angle += 1.57f * m_delta_time;
	constant constant_obj;
	constant.m_angle = m_angle;

	m_cb->update(GraphicsEngine::get()->getImmediateDeviceContext(), & constant);

	GraphicsEngine::get()->getImmediateDeviceContext()->setConstantBuffer(m_vs, m_cb);
	GraphicsEngine::get()->getImmediateDeviceContext()->setConstantBuffer(m_ps, m_cb);
```

Figure 4.32 Calculating the angle according to elapsed time

In the “onUpdate” method, the angle is updating according to time to apply the animations.

This code block firstly initializes a new variable “new_time”, to store the current time.

Then, checks if “m_old_tim”e has a non-zero value. If true, it calculates the time passed since the last frame using “::GetTickCount()”.

“m_delta_time = new_time / 1000.0f;” converts the time difference to seconds and assigns it to “m_delta_time”. This variable is commonly used to ensure smooth animations regardless of the frame rate.

Then updating “m_old_time” with the current time for the next frame.

“m_angle += 1.57f * m_delta_time;” increments the angle based on the elapsed time.

Then, creates the constant object and sets its “m_angle” member to the calculated angle.

“m_constant_buffer->update” updates the constant buffer with the new constant data.

Then sets the constant buffer both for the vertex shader and pixel shader in the immediate device context.

Final result can be seen in the Figure 4.33.

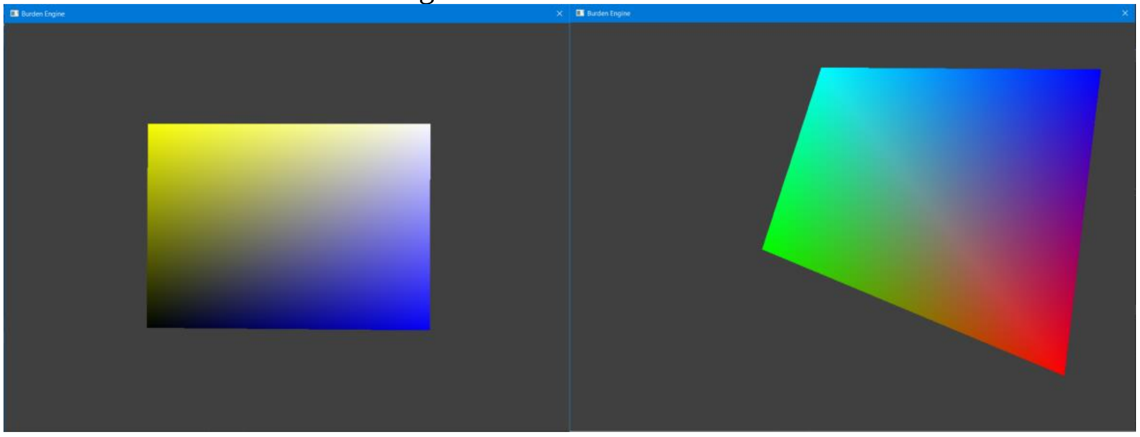

Figure 4.33 Applying first animations with lerp

In computer graphics, the use of transform matrices is often regarded as a fundamental and logical approach when implementing animations. Transform matrices provide a concise and efficient representation of spatial transformations, encapsulating translation, rotation, and scaling operations in a unified structure. When applying animations, especially those involving complex movements or deformations, employing a transform matrix allows for a streamlined and coherent representation of the entire transformation pipeline. This not only enhances code readability but also facilitates easier manipulation of the object's transformation state. Without a transform matrix, the implementation of animations may become cumbersome, involving manual adjustments of position, rotation, and scale parameters independently. The use of matrices simplifies the animation process, promoting a modular and organized coding practice that aligns with the principles of clarity and efficiency in computer graphics development.

Now, let’s modify the codes to apply transform matrices.

```
struct VS_INPUT
{
	float4 position: POSITION;
	float4 position1: POSITION1;
	float3 color: COLOR;
	float3 color1: COLOR1;
};

struct VS_OUTPUT
{
	float4 position: SV_POSITION;
	float3 color: COLOR;
	float3 color1: COLOR1;
};

cbuffer constant: register(b0)
{
	row_major float4x4 m_world;
	row_major float4x4 m_view;
	row_major float4x4 m_proj;
	unsigned int m_time;
};

VS_OUTPUT vsmain(VS_INPUT input)
{
	VS_OUTPUT output = (VS_OUTPUT)0;

	//WORLD SPACE
	output.position = mul(input.position, m_world);
	//VIEW SPACE
	output.position = mul(output.position, m_view);
	//SCREEN SPACE
	output.position = mul(output.position, m_proj);


	output.color = input.color;
	output.color1 = input.color1;
	return output;
}
```

Figure 4.34 VertexShader.hlsl

The constant buffer contains transformation matrices (“m_world”, “m_view”, “m_proj”) and a time variable (“m_time”). The “row_major” qualifier indicates that the data is stored in row-major order. In row-major order, the elements of each row of the matrix are stored sequentially in memory.

For a 2x3 matrix, the storage order would be: a11, a12, a13, a21, a22, a23. This means that consecutive elements in a row are contiguous in memory. The reason behind the defining as row-major order is because the graphical library. Column-major order is preferred in OpenGL as an example.

Main function transforms the position from object space to world space, then to view space, and finally to screen space using the provided matrices. The transformed position and colors are stored in the “VS_OUTPUT” structure, which is sent to the next stage in the graphics pipeline.

```
struct PS_INPUT
{
	float4 position: SV_POSITION;
	float3 color: COLOR;
	float3 color1: COLOR1;
};

cbuffer constant: register(b0)
{
	row_major float4x4 m_world;
	row_major float4x4 m_view;
	row_major float4x4 m_proj;
	unsigned int m_time;
};

float4 psmain(PS_INPUT input) : SV_TARGET
{
	return float4(lerp(input.color, input.color1, (float)((sin((float)(m_time /
(float)500.0f)) + 1.0f) / 2.0f)),1.0f);
}
```

Figure 4.35 PixelShader.hlsl

Like in the vertex shader, in the “PixelShader.hlsl”, transform matrices is defined in the constant buffer struct.

The main function takes the interpolated color values and performs linear interpolation. The interpolation is based on a sine function that varies with time.

Now, let’s check the “AppWindow” class.

In the header file, a new method is defined as “updateQuadPosition” to handle the updates of the rendered quad and member variables is defined to get the changes for every animation (position change, rescaling and rotation) as follows:

```
	void updateQuadPosition();

private:
	long m_old_delta;
	long m_new_delta;
	float m_delta_time;

	float m_delta_pos;
	float m_delta_scale;
	float m_delta_rot;
```

Figure 4.36 Modifications on the AppWindow.h

In the source file, constant struct is modified accordingly to applying transform matrices:

```
__declspec(align(16))
struct constant
{
	Matrix4x4 m_world;
	Matrix4x4 m_view;
	Matrix4x4 m_proj;
	unsigned int m_time;
};
```

Figure 4.37 Updating constant struct

Now, in the “updateQuadPosition” method:

```
void AppWindow::updateQuadPosition()
{
	constant constant_obj;
	constant_obj.m_time = ::GetTickCount();
```
Figure 4.38 Creating constant object

At first, a constant object created and set the time variable with "::GetTickCount" method.

```
	m_delta_pos += m_delta_time / 10.0f;
	if (m_delta_pos > 1.0f)
		m_delta_pos = 0;
```
Figure 4.39 Changing the position data

Then, to move the quad, position change constantly increased according to elapsed time. The condition block is there to set the position to start when the quad is fully out of the screen.

```
	Matrix4x4 temp;

	m_delta_scale += m_delta_time / 0.15f;

	constant_obj.m_world.setScale(Vector3D::lerp(Vector3D(0.5, 0.5, 0), Vector3D(1.0f, 1.0f, 
0), (sin(m_delta_scale) + 1.0f) / 2.0f));

	temp.setTranslation(Vector3D::lerp(Vector3D(-1.5f, -1.5f, 0), Vector3D(1.5f,1.5f, 0), 
m_delta_pos));

	constant_obj.m_world *= temp;
```
Figure 4.40 Scaling and transformation

Matrices are manipulated to achieve scaling and translation animations. The "constant_obj.m_world matrix" is first scaled using lerp between two vector values. The translation is then applied to "temp", and the resulting matrix is multiplied into "constant_obj.m_world".

```
	constant_obj.m_view.setIdentity();
	constant_obj.m_proj.setOrthoLH
	(
		(this->getClientWindowRect().right - this->getClientWindowRect().left)/400.0f,
		(this->getClientWindowRect().bottom - this->getClientWindowRect().top)/400.0f,
		-4.0f,
		4.0f
	);
```
Figure 4.41 Setting the orthogonal projection

The view matrix is set to identity, and the projection matrix is set using an orthographic projection with screen dimensions and a depth range of -4.0 to 4.0.

```
	m_cb->update(GraphicsEngine::get()->getImmediateDeviceContext(), &constant_obj);
}
```

Figure 4.42 Updating constant buffer

Finally, the constant buffer is updated with the new values from the "constant_obj" structure.

In the “onCreate” method, vertex list is updated with “Vector3D”:

```
        vertex list[] =
        {
                //X - Y - Z
                {Vector3D(-0.5f,-0.5f,0.0f),    Vector3D(-0.32f,-0.11f,0.0f),   Vector3D(0,0,0),
Vector3D(0,1,0) }, // POS1
                {Vector3D(-0.5f,0.5f,0.0f),     Vector3D(-0.11f,0.78f,0.0f),   Vector3D(1,1,0),
Vector3D(0,1,1) }, // POS2
                { Vector3D(0.5f,-0.5f,0.0f),     Vector3D(0.75f,-0.73f,0.0f), Vector3D(0,0,1),
Vector3D(1,0,0) },// POS2
                { Vector3D(0.5f,0.5f,0.0f),     Vector3D(0.88f,0.77f,0.0f),    Vector3D(1,1,1),
Vector3D(0,0,1) }
        };
```

Figure 4.43 Updating vertex list

And in the “onUpdate” method:

```
        updateQuadPosition();

        m_old_delta = m_new_delta;
        m_new_delta = ::GetTickCount();

        m_delta_time = (m_old_delta)?((m_new_delta - m_old_delta) / 1000.0f):0;
```

Figure 4.44 onUpdate modifications

“updateQuadPosition” is called before setting the constant buffer.

The delta time is calculated as the difference between “m_new_delta” and “m_old_delta”, divided by 1000 to convert milliseconds to seconds. If “m_old_delta” is 0 (indicating the first frame or an uninitialized state), “m_delta_time” is set to 0.

Final result can be seen in the Figure 4.45.

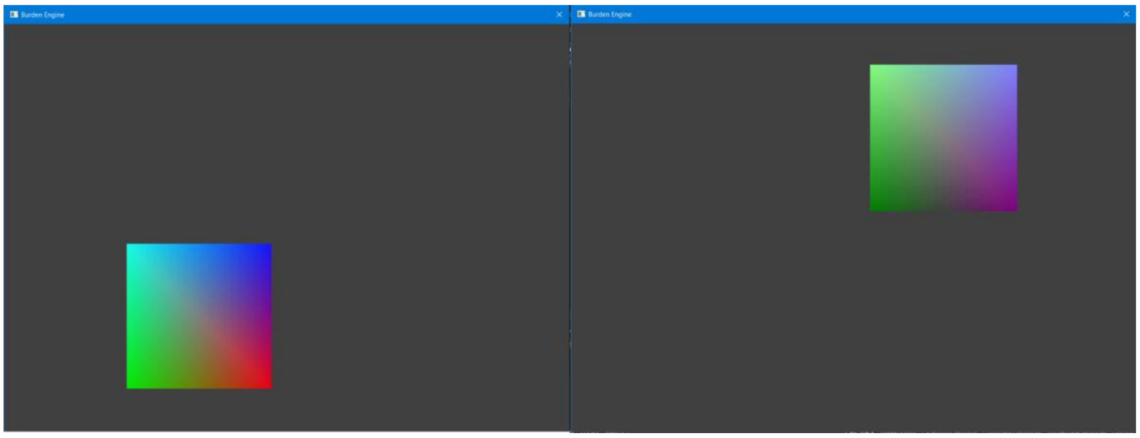

Figure 4.45 Applying transform matrix to quad

4.7 Rendering Cube and Enhance with Animations

To render a 3D object like a cube, we need index buffer to render the faces. The cube has eight actual vertices and this can be obtained by defining two parallel quads. If the distance between the quads set as the length of the quad’s lines, a cube can be fromed.

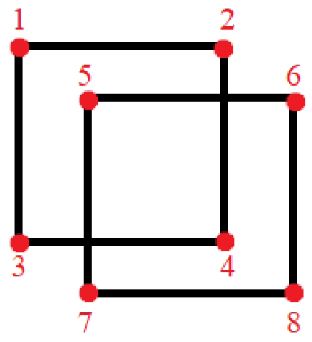

Figure 4.46 Defining two quads to obtain a cube

As can be seen in the Figure 4.9, eight vertices is obtained by defining two quads. To render all, the faces is defined with the indices. As an example, if index list in the Figure 4.9 is considering, the right face of the cube can be defined as (1,3,7) and (7,5,1) to render the face as a quad with triangle strip.

The HLSL files are the same with chapter 4.6. So, we can directly look at the AppWindow class.

In the header file, only need is to define index buffer object:

```
IndexBuffer* m_index_buffer;
```

Figure 4.47 Defining index buffer as a member variable

In the source file, both front and back face must be defined to the vertex list that defined in “onCreate” mehod:

```
vertex vertex_list[] =
{
        //X - Y - Z
        //FRONT FACE
        {Vector3D(-0.5f,-0.5f,-0.5f),    Vector3D(1,0,0),  Vector3D(0.2f,0,0) },
        {Vector3D(-0.5f,0.5f,-0.5f),    Vector3D(1,1,0), Vector3D(0.2f,0.2f,0) },
        { Vector3D(0.5f,0.5f,-0.5f),   Vector3D(1,1,0),  Vector3D(0.2f,0.2f,0) },
        { Vector3D(0.5f,-0.5f,-0.5f),     Vector3D(1,0,0), Vector3D(0.2f,0,0) },

        //BACK FACE
        { Vector3D(0.5f,-0.5f,0.5f),    Vector3D(0,1,0), Vector3D(0,0.2f,0) },
        { Vector3D(0.5f,0.5f,0.5f),    Vector3D(0,1,1), Vector3D(0,0.2f,0.2f) },
        { Vector3D(-0.5f,0.5f,0.5f),   Vector3D(0,1,1),  Vector3D(0,0.2f,0.2f) },
        { Vector3D(-0.5f,-0.5f,0.5f),     Vector3D(0,1,0), Vector3D(0,0.2f,0) }

};
```

Figure 4.48 Updating vertex list

Then after the creation of vertex buffer, an index list is created to draw the faces with triangle strip one by one:

```
unsigned int index_list[]=
{
        //FRONT SIDE
        0,1,2,  //FIRST TRIANGLE
        2,3,0,  //SECOND TRIANGLE
        //BACK SIDE
        4,5,6,
        6,7,4,
        //TOP SIDE
        1,6,5,
        5,2,1,
        //BOTTOM SIDE
        7,0,3,
        3,4,7,
        //RIGHT SIDE
        3,2,5,
        5,4,3,
        //LEFT SIDE
        7,6,1,
        1,0,7
};

m_index_buffer = GraphicsEngine::get()->createIndexBuffer();
```

Figure 4.49 Defining index list

After the definition of index list, index buffer is created by calling the necessary function from "GraphicsEngine".

```
UINT size_index_list = ARRAYSIZE(index_list);

m_index_buffer->load(index_list, size_index_list);
```

Figure 4.50 Loading data to index list

After getting the size of index list, loading method is called to index buffer to send the information to drawing methods.

And inside of the "updateQuadPosition" method, after the position condition check:

```
m_delta_scale += m_delta_time / 0.55f;

const_obj.m_world.setScale(Vector3D(1, 1, 1));
```

Figure 4.51 Scale

"m_delta_scale" controls the rotation animation. It is incremented by a fraction of "m_delta_time". The constant buffer's world matrix ("const_obj.m_world") is set to an identity scale matrix.

```
temp.setIdentity();
temp.setRotationZ(m_delta_scale);
const_obj.m_world *= temp;

temp.setIdentity();
temp.setRotationY(m_delta_scale);
const_obj.m_world *= temp;

temp.setIdentity();
temp.setRotationX(m_delta_scale);
const_obj.m_world *= temp;
```

Figure 4.52 Calling rotations

Three rotation matrices are created (“temp” which is a “Matrix4x4”) for rotations around the z, y and x axes. These matrices are then multiplied with the world matrix in “const_obj.m_world”. This sequence of rotations allows for a complex rotation effect.

```
        const_obj.m_view.setIdentity();
        const_obj.m_proj.setOrthoLH
        (
                (this->getClientWindowRect().right - this->getClientWindowRect().left)/300.0f,
                (this->getClientWindowRect().bottom - this->getClientWindowRect().top)/300.0f,
                -4.0f,
                4.0f
        );
```

Figure 4.53 Orthogonal projection

The view matrix is set to an identity matrix, and the projection matrix is set to an orthographic projection using the window dimensions and near/far plane values.

```
        m_constant_buffer->update(GraphicsEngine::get()->getImmediateDeviceContext(),
&const_obj);
```

The constant buffer is updated with the values in “const_obj”, reflecting the changes in position, rotation, and projection.

Finally, in the “onUpdate” method:

```
        GraphicsEngine::get()->getImmediateDeviceContext()-> drawIndexedTriangleList(
                m_index_buffer->getSizeIndexList(),0, 0
        );
```

Figure 4.54 Calling draw method

“drawIndexedTriangleList” method is called to draw with index buffer. Final result can be seen in the Figure 4.55:

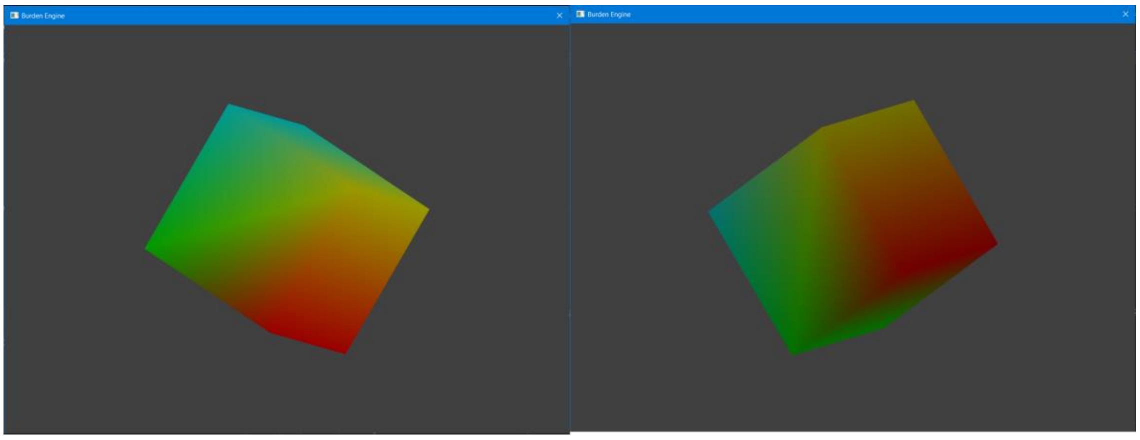

Figure 4.55 Rendering cube and applying the animations

4.8 Keyboard and Mouse Usage

This section is about how to implement the methods that described in “InputSystem” can be implement to “AppWindow” class.

To apply keyboard, at first “AppWindow” class is set to be a sub-class of “InputSystem” class in the header file:

```
class AppWindow: public Window, public InputListener
```

Figure 4.56 Start of the AppWindow.h

After that, iheriting the methods:

```
        // Inherited via InputListener
        virtual void onKeyDown(int key) override;
        virtual void onKeyUp(int key) override;
```

Figure 4.57 Inheriting InputListener methods

And on the source file, first the listeners must be added to “onCreate” method:

```
        InputSystem::get()->addListener(this);
        InputSystem::get()->showCursor(false);
```

Figure 4.58 addListener and showCursor

“showCursor” method takes the parameter false to not show the cursor when the active window is the program.

```
void AppWindow::onKeyDown(int key)
{
        if (key == 'W')
        {
                m_rotation_x += 3.1415f*m_delta_time;
        }
        else if (key == 'S')
        {
                m_rotation_x -= 3.1415f*m_delta_time;
        }
        else if (key == 'A')
        {
                m_rotation_y += 3.1415f*m_delta_time;
        }
        else if (key == 'D')
        {
                m_rotation_y -= 3.1415f*m_delta_time;
        }
}

void AppWindow::onKeyUp(int key)
{

}
```

Figure 4.59 Assignment of the keyboard keys

“onKeyDown” and “onKeyUp” functions can be used directly. To test the implementation, “m_rotation_x” and “m_rotation_y” is defined as float variables. When pressed the interested button (WASD in that case) the rendered cube is rotating through increase of rotation variables, which they used as parameters in “setRotation” methods inside the “updateQuadPosition” method.

To implement the mouse buttons, more functions from InputSystem is inherited and a variable for changing the scale on mouse button event is added:

```
        virtual void onMouseMove(const Point& delta_mouse_pos) override;

        virtual void onLeftMouseDown(const Point& mouse_pos) override;
        virtual void onLeftMouseUp(const Point& mouse_pos) override;
        virtual void onRightMouseDown(const Point& mouse_pos) override;
        virtual void onRightMouseUp(const Point& mouse_pos) override;

private:
        float m_scale_cube = 1.0f;
```

Figure 4.60 Inherited methods from InputListener

And in the source file:

```
void AppWindow::onMouseMove(const Point & delta_mouse_pos)
{
        m_rotation_x -= delta_mouse_pos.m_y*m_delta_time;
        m_rotation_y -= delta_mouse_pos.m_x*m_delta_time;
}
```

Figure 4.61 onMouseMove

“onMouseMove”, rotates the rendered cube according to mouse position change.

```
void AppWindow::onLeftMouseDown(const Point & mouse_pos)
{
        m_scale_cube = 0.5f;
}

void AppWindow::onLeftMouseUp(const Point & mouse_pos)
{
        m_scale_cube = 1.0f;
}

void AppWindow::onRightMouseDown(const Point & mouse_pos)
{
        m_scale_cube = 2.0f;
}

void AppWindow::onRightMouseUp(const Point & mouse_pos)
{
        m_scale_cube = 1.0f;
}
```

Figure 4.62 Mouse button events

Button functions change the size of the cube when pressed. Left-click of the mouse changes the scale by dividing in half. And the right-click multiplies by two. When the pressed mouse button is released, the size of the cube returns the default value (which is 1.0f in this case.).

All of these variables (for rotation and scale) are sending as parameters of rotation and scale functions of the cube, which are called in the “updateQuadPosition” function.

4.9 First-Person Camera and Movement

To feel the experience while playing games or simulating a rover, camera control is a crucial aspect of the game engines. To move around with keyboard and to look around with mouse movement is the simplest way to experience the scene that created in the program.

There are several POVs in games: third-person, first-person, side, isometric, top-down etc. In this chapter, implementing the first-person camera to Burden Engine is explained and how to navigate with keyboard and mouse for a free roam.

To start, "updateQuadPosition" method's name is changed to "update", and necessary variables to move the camera is added to header file of "AppWindow":

```
	void update();
private:
	float m_forward = 0.0f;
	float m_horizontal_move_coefficient = 0.0f;
	Matrix4x4 m_world_cam;
```

Figure 4.63 Modifications on AppWindow.h

A world cam variable is added to represent the world-to-camera transformation matrix. It's defining the position and orientation of the camera in the world space.

After that, in source file, "update" function is modified:

```
void AppWindow::update()
{
	constant const_obj;
	const_obj.m_time = ::GetTickCount();

	m_delta_pos += m_delta_time / 10.0f;
	if (m_delta_pos > 1.0f)
		m_delta_pos = 0;

	Matrix4x4 temp;

	m_delta_scale += m_delta_time / 0.55f;
```

Figure 4.64 update method

Start of the function is the same with previous method.

```
	const_obj.m_world.setIdentity();

	Matrix4x4 world_cam;
	world_cam.setIdentity();
```

Figure 4.65 creating the world cam

The identity matrix is assigned to "m_world" member of "const_obj". A new "Matrix4x4" named "world_cam" is created and initialized to the identity matrix.

```
	temp.setIdentity();
	temp.setRotationX(m_rot_x);
	world_cam *= temp;

	temp.setIdentity();
	temp.setRotationY(m_rot_y);
	world_cam *= temp;
```

Figure 4.66 Rotations

Rotations around the x and y axes is applied to "world_cam" through the "temp" matrix.

```
        Vector3D new_position = m_world_cam.getTranslation() + (world_cam.getZDirection() *
(m_forward * .3f) + world_cam.getXDirection() * (m_horizontal_move_coefficient * .3f));
```

Figure 4.77 Calcuating the camera position

A new position vector is calculated based on the translation of "m_world_cam", its forward direction and horizontal movement coefficient.

```
        world_cam.setTranslation(new_position);
        m_world_cam = world_cam;
        world_cam.inverse();

        const_obj.m_view = world_cam;
```

Figure 4.78 Updating the world cam

The translation of "world_cam" is updated, and then the camera's matrix is updated to "m_world_cam". Finally, the inverse of "world_cam" is taken.

In computer graphics, objects are usually defined in their own local coordinate systems (like the Burden's in-engine coordinate system) and these objects need to be transformed into the world space and then into the camera space for rendering. The camera matrix represents the transformation from world space to camera space. To transform an object from camera space to world space, the inverse of the camera matrix is needed.

So, after taking inverse of the world cam, "const_obj.m_view" is assigned the value of "world_cam".

```
        int width = (this->getClientWindowRect().right - this->getClientWindowRect().left);
        int height = (this->getClientWindowRect().bottom - this->getClientWindowRect().top);

        const_obj.m_proj.setPerspectiveFovLH(1.57079632f, ((float)width / (float)height), 0.1f,
                                             100.0f);
```

Figure 4.79 Setting the perspective field of view

The width and height of the window client area are calculated. Then, the perspective projection matrix is set in "const_obj.m_proj" using the field of view (1.57079632f radians, which equals to $\pi / 2$), aspect ratio and near/far plane values.

In Figure 4.11, the effect of perspective field of view can be seen. The frame on the left is the start of the program and the frame on the right is obtained by mouse movement, which will be explained later in this section.

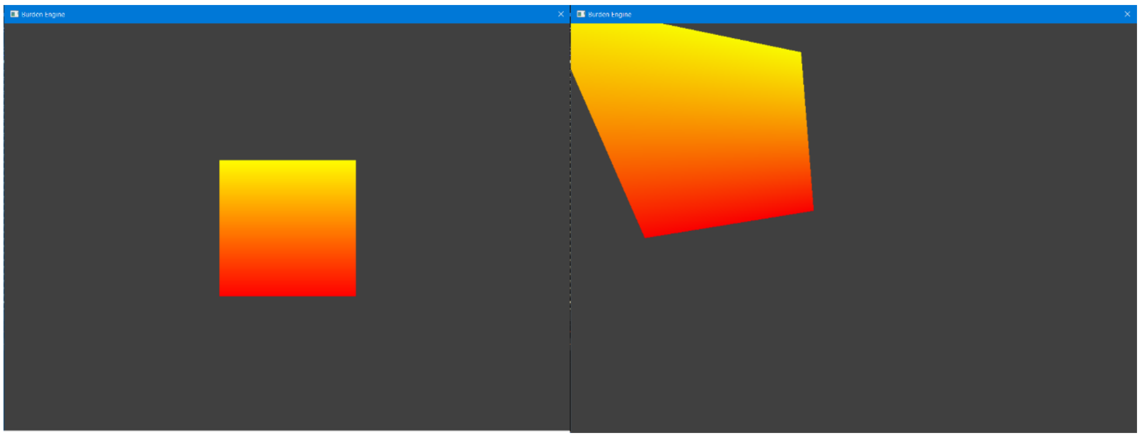

Figure 4.80 Perspective field of view effect

```
        m_constant_buffer->update(GraphicsEngine::get()->getImmediateDeviceContext(),
&const_obj);
}
```

Figure 63 Updating constant buffer

Finally, constant buffer is updated with the information in "const_obj".

```
void AppWindow::onKeyDown(int key)
{
        if (key == 'W')
        {
                m_forward = 1.0f;
        }
        else if (key == 'S')
        {
                m_forward = -1.0f;
        }
        else if (key == 'A')
        {
                m_horizontal_move_coefficient = -1.0f;
        }
        else if (key == 'D')
        {
                m_horizontal_move_coefficient = 1.0f;
        }
        else if (key == VK_ESCAPE)
        {
                m_is_run = false;
        }
}
```

Figure 4.82 Updating key assigments to move the camera

"onKeyDown" method is modified to change the move coefficients between positive or negative according to the pressed button. To move, user can use WASD buttons of the keyboard. WASD is the common movement keys when using the keyboard. W and S keys are responsible for the move forwards or backwards. A and D keys are responsible for the horizontal movement, or simply can be defined as left or right. ESCAPE key is setting "m_is_running" as false to quit the program immediately.

```
void AppWindow::onKeyUp(int key)
{
	m_forward = 0.0f;
	m_horizontal_move_coefficient = 0.0f;
}
```

Figure 4.83 Stopping the camera when the buttons are released

In “onKeyUp” function, both movement coefficients set to zero to prevend the movement while the interested keys are not pressed.

```
void AppWindow::onMouseMove(const Point& mouse_pos)
{
	int width = (this->getClientWindowRect().right - this->getClientWindowRect().left);
	int height = (this->getClientWindowRect().bottom - this->getClientWindowRect().top);

	m_rotation_x += (mouse_pos.m_y- (height / 2.0f))*m_delta_time*0.1f;
	m_rotation_y += (mouse_pos.m_x - (width / 2.0f))*m_delta_time*0.1f;

	InputSystem::get()->setCursorPosition(Point((int)(width / 2.0f), (int)(height / 2.0f)));
}
```

Figure 4.84 Updating variables to change the angle of camera when mouse move triggered

In “onMouseMove”, first window dimensions are calculated.

Then, rotation angles are updated based on the change in mouse position. The rotation is influenced by the vertical (“m_rotation_x”) and horizontal (“m_rotation_y”) movement of the mouse. The “m_delta_time” factor indicates that the rotation change is proportional to the elapsed time.

Finally, the cursor position reset to the center of the window. This is a common practice in first-person camera controls to keep the mouse cursor from reaching the window edges, ensuring continuous mouse input.

With this, now user can free-roam with keyboard and mouse to examine the cube. Rendered cube can be seen in various angles in Figure 4.85.

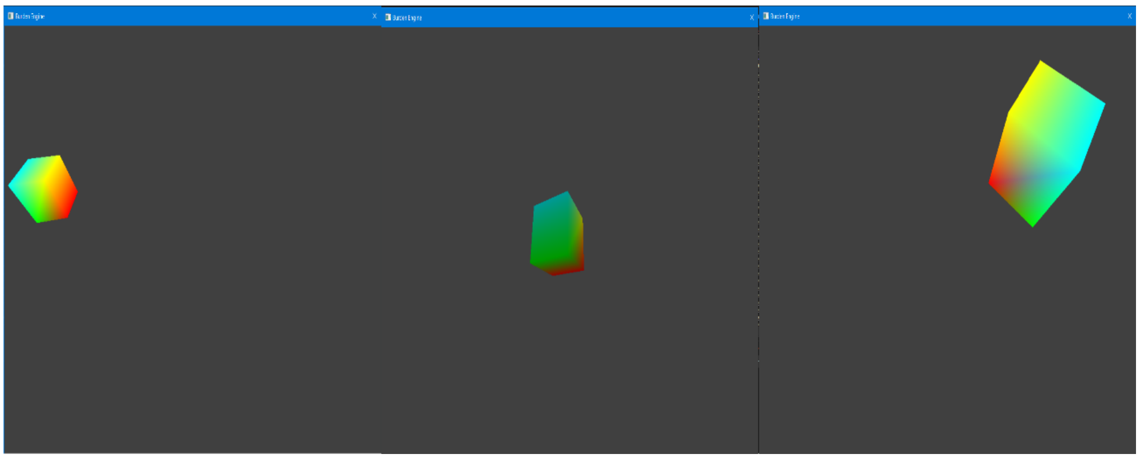

Figure 4.85 Roaming around the cube with mouse and keyboard

4.10 Texture from File

As mentioned before, textures are image files that used to cover the rendered object.

At first, we need to adjust the “PixelShader.hlsl” and “VertexShader.hlsl”:

```
Texture2D Texture: register(t0);
sampler TextureSampler: register(s0);

struct PS_INPUT
{
	float4 position: SV_POSITION;
	float2 texcoord: TEXCOORD0;
};

cbuffer constant: register(b0)
{
	row_major float4x4 m_world;
	row_major float4x4 m_view;
	row_major float4x4 m_proj;
	unsigned int m_time;
};

float4 psmain(PS_INPUT input) : SV_TARGET
{
	return Texture.Sample(TextureSampler,input.texcoord);
}
```

Figure 4.86 PixelShader.hlsl

First line in the Figure declares a 2D texture named “Texture” and assigns it to register “t0”. This texture is defined to store image data.

Second line declares a sampler named “TextureSampler” and assigns it to register “s0”. The sampler defines how texture coordinates are sampled, specifying parameters like filtering and addressing modes.

Pixel shader input is changed to get texture coordinates. A two membered float vector is defined to hold texture coordinates. “position” is the same.

Finally, pixel shader main is returns the texture and texcoord to draw textures to a specific location in the scene. In this example, the textures will be drawn on the cube’s surfaces.

```
struct VS_INPUT
{
	float4 position: POSITION0;
	float2 texcoord: TEXCOORD0;
};

struct VS_OUTPUT
{
	float4 position: SV_POSITION;
	float2 texcoord: TEXCOORD0;
};

cbuffer constant: register(b0)
{
	row_major float4x4 m_world;
	row_major float4x4 m_view;
	row_major float4x4 m_proj;
	unsigned int m_time;
};

VS_OUTPUT vsmain(VS_INPUT input)
{
	VS_OUTPUT output = (VS_OUTPUT)0;
	//WORLD SPACE
	output.position = mul(input.position, m_world);
	//VIEW SPACE
	output.position = mul(output.position, m_view);
	//SCREEN SPACE
	output.position = mul(output.position, m_proj);

	output.texcoord = input.texcoord;
	return output;
}
```

Figure 64 VertexShader.hlsl

As in the pixel shader, vertex shader input and output is changed accordingly to hold the texcoord data. And main function of vertex shader transforms the vertex position from object space to world space, then to view space, and finally to screen space. The texture coordinates are passed through without transformation.

In the “AppWindow” class, at first a “TexturePtr” is defined in the header file:

```
	TexturePtr m_woodenbox_tex;
```

Figure 65 Defining the texture holder variable

This variable “m_woodenbox_tex” will hold the texture image. In the source file of the class:

```
struct vertex
{
	Vector3D position;
	Vector2D texcoord;
};
```

Figure 66 vertex struct that defined in the AppWindow.cpp

Again like in the shaders, “vertex” struct is changed accordingly to hold the texcoord data.

```
        m_woodenbox_tex = GraphicsEngine::get()->getTextureManager()-
>createTextureFromFile(L"..\\Assets\\Textures\\wood.jpg");
```

Figure 67 Getting the texture from file

To organize the assets more easily, a folder named “Assets” and another folder named “Textures” are added to the same directory with the project. A texture image (“wood.jpg”) is added to the “Textures” folder. “createTextureFromFile” is called in the “onCreate” method. Argument of the method is the path to the “wood.jpg”. And this is stored in the “m_woodenbox_tex”.

```
        GraphicsEngine::get()->getRenderSystem()->getImmediateDeviceContext()->setTexture(m_ps,
m_wood_tex);
```

Figure 68 Calling setTexture

To apply the textures to the cube, “setTexture” method is called. As shader, pixel shader is used.

Final result of these adjustments can be seen in the Figure 4.92:

(jitspoe,)

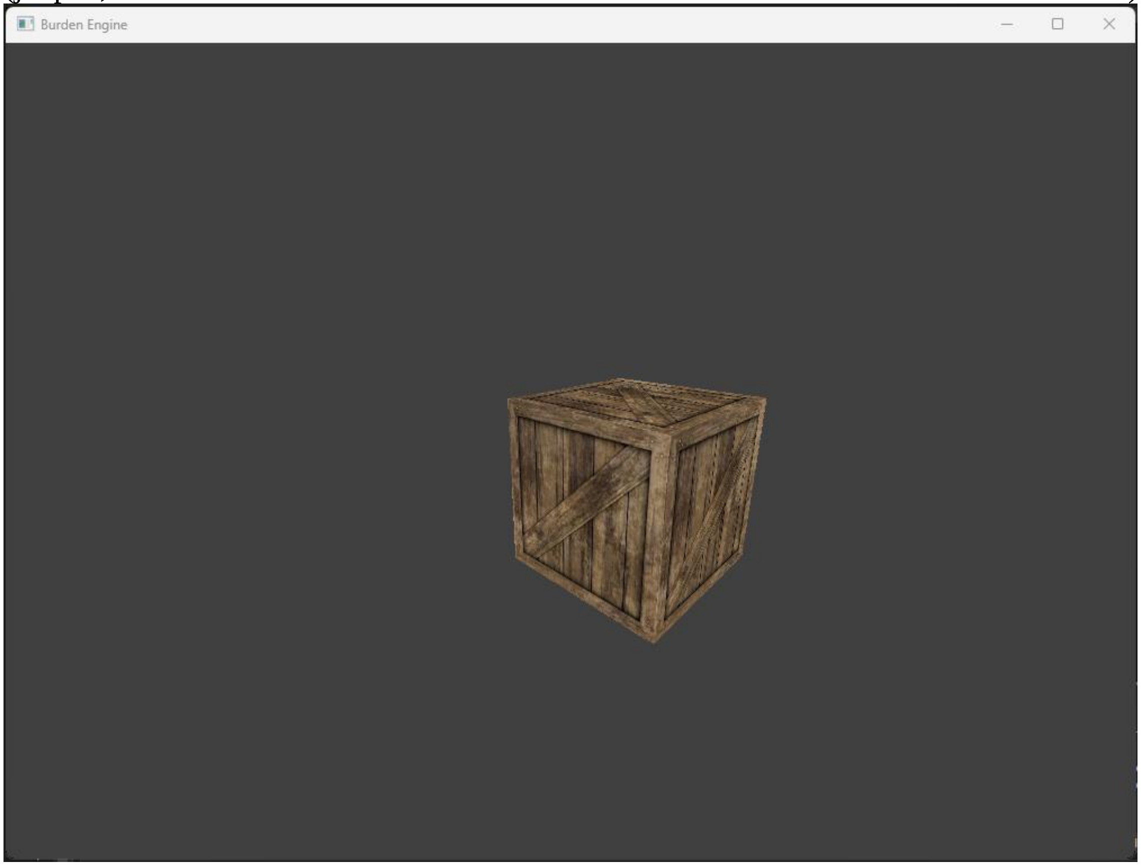

Figure 69 Adding texture to the cube

4.11 3D Object Rendering from File

Object rendering is another crucial step of the game engines. Most of the components in the scenes are designed and created by the 2D/3D artists. So, game engines must be able to render different objects from files. The object files' data is generally holding in the OBJ files. In this section, the system of the Burden Engine that renders OBJ files to the screen is explained.

For the first object that will be rendered via Burden Engine will be "The Utah Teapot". The teapot is a 3D test model that has become a standard reference object and an in-joke within the computer graphics community. It is a mathematical model of an ordinary Melitta-brand teapot that appears solid with a nearly rotationally symmetrical body. Using a teapot model is considered the 3D equivalent of a "Hello, World!" program, a way to create an easy 3D scene with a somewhat complex model acting as the basic geometry for a scene with a light setup. It was designed by Martin Newell in 1975 at the University of Utah. (Hall, 2005)

To apply this system, at first a "MeshPtr" variable must be defined to hold the OBJ data. So in the header file of the "AppWindow":

```
        MeshPtr m_mesh;
```

Figure 70 MeshPtr defined

After that, in the source file of the class:

```
        m_mesh = GraphicsEngine::get()->getMeshManager()-
>createMeshFromFile(L"..\\Assets\\Meshes\\teapot.obj");
```

Figure 71 Getting mesh from file

Like the texture, "createMeshFromFile" method is called. Again a "Meshes" folder created in the "Assets" folder to store the OBJ files. The "teapot.obj"s path is passed as an argument of the method.

```
        //SET THE VERTICES OF THE TRIANGLES TO DRAW
        GraphicsEngine::get()->getRenderSystem()->getImmediateDeviceContext()-
>setVertexBuffer(m_mesh->getVertexBuffer());
        //SET THE INDICES OF THE TRIANGLES TO DRAW
        GraphicsEngine::get()->getRenderSystem()->getImmediateDeviceContext()-
>setIndexBuffer(m_mesh->getIndexBuffer());
```

Figure 72 Setting vertices and indices of the OBJ

These lines sets the vertices and indices of the triangles that create the model. These triangles are calculated via "tinyobjloader" which is explained in the related sub-section of Chapter 3.

```
        GraphicsEngine::get()->getRenderSystem()->getImmediateDeviceContext()-
>drawIndexedTriangleList(m_mesh->getIndexBuffer()->getSizeIndexList(),0, 0);
```

Figure 73 Draw method is called

Finally draw method is called. It uses the index list of the "m_mesh" to draw triangles.

Final result can be seen in the Figure 4.97, the codes for the texture is the same, so the teapot is covered with the “m_woodenbox_tex”:

(Bennet, 2014)

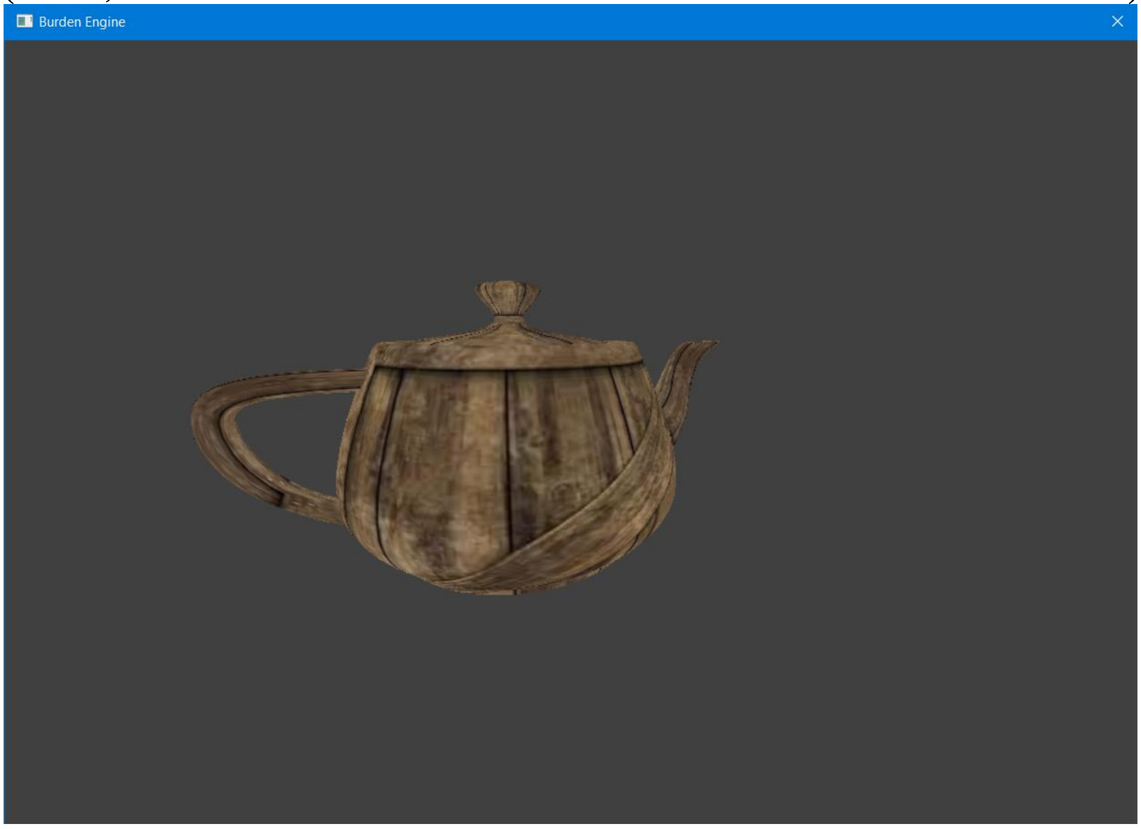

Figure 74 Rendered 3D model of teapot

4.12 Lighting with Phong Shading

Phong shading is a shading model in computer graphics that is used to simulate the way light interacts with surfaces. It was developed by Bui-Tuong Phong and introduced in his 1973 paper "Illumination for Computer-Generated Pictures". (Perlin, 2005)

(Vries, 2014)

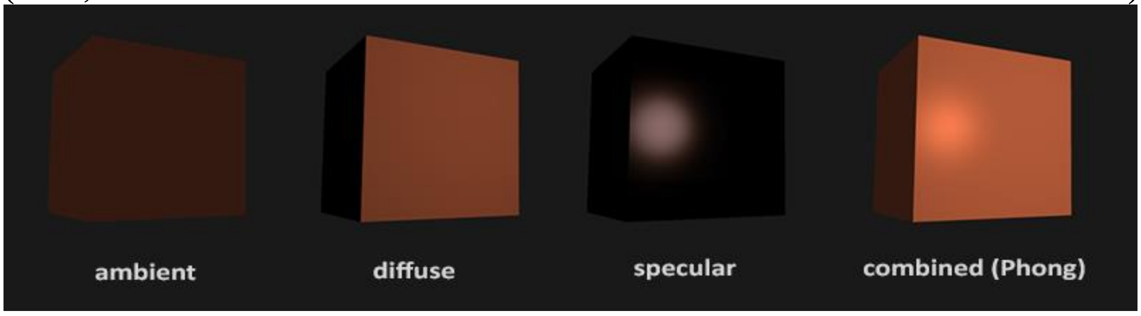

Figure 75 Illustration of Phong Shading

Phong shading combines the three reflection types:

Ambient reflection represents the light that is scattered in all directions, creating a uniform illumination across the entire surface. This component is not affected by the direction of the light source or the viewing angle. It is a constant term that adds a base level of illumination to the object.

Diffuse reflection models the way light scatters when it strikes a surface, creating a matte appearance. The intensity of the diffuse reflection depends on the angle between the incoming light direction and the surface normal. It is maximal when the light is perpendicular to the surface and decreases as the angle between the light and the surface normal increases.

Specular reflection accounts for the shiny highlight observed on surfaces when illuminated. The intensity of the specular reflection depends on the viewing angle, light direction, and surface properties. It is most pronounced when the viewer is aligned with the reflection direction of the light.

So, to apply the Phong Shading, both of the shader scripts must be modified:

```
struct VERTEX_SHADER_INPUT
{
    float4 pos : POSITION0;
    float2 texcoord : TEXCOORD0;
    float3 normal : NORMAL0;
};

struct VERTEX_SHADER_OUTPUT
{
    float4 pos : SV_POSITION;
    float2 texcoord : TEXCOORD0;
    float3 normal : TEXCOORD1;
    float3 direction_to_cam : TEXCOORD2;
};

cbuffer constant : register(b0)
{
    row_major float4x4 m_world;
    row_major float4x4 m_view;
    row_major float4x4 m_projection;
    float4 m_light_direction;
    float4 m_cam_position;
};

VERTEX_SHADER_OUTPUT vsmain(VERTEX_SHADER_INPUT input)
{
    VERTEX_SHADER_OUTPUT output = (VERTEX_SHADER_OUTPUT) 0;

    //output.pos = lerp(input.pos, input.pos1, (sin(m_angle) + 1.0f) / 2.0f);

    /* WORLD SPACE */
    output.pos = mul(input.pos, m_world); //mul(): matrix multiplication method
    output.direction_to_cam = normalize(output.pos.xyz - m_cam_position.xyz);
    /* VIEW SPACE */
    output.pos = mul(output.pos, m_view);
    /* PROJECTION SPACE */
    output.pos = mul(output.pos, m_projection);

    output.texcoord = input.texcoord;
    output.normal = input.normal;

    return output;
}
```

Figure 76 VertexShader.hlsl

At first, the input and output structs of the vertex shader is updated. “normal” is the normal vectors of the vertices. “direction_to_cam” is the vector that represents the direction from the vertex to the camera.

In constant buffer, “m_light_direction” and “m_cam_position” vectors are added.

Main function of the shader performs transformations from object space to world space, then to view space, and finally to projection space like previously. Additionally, The direction_to_cam is calculated as the normalized vector pointing from the transformed vertex position to the camera position. The function also passes through texture coordinates and normal from the input to the output.

And, for the “PixelShader.hlsl”:

```
Texture2D Texture : register(t0);
sampler TextureSampler : register(s0);

struct PIXEL_SHADER_INPUT
{
    float4 pos : SV_POSITION;
    float2 texcoord : TEXCOORD0;
    float3 normal : TEXCOORD1;
    float3 direction_to_cam : TEXCOORD2;
};

cbuffer constant : register(b0)
{
    row_major float4x4 m_world;
    row_major float4x4 m_view;
    row_major float4x4 m_projection;

    float4 m_light_direction;
};

float4 psmain(PIXEL_SHADER_INPUT input) : SV_TARGET
{
    float alpha = 1.0f;
    /* ambient */
    float ka = .1f;
    float3 ia = float3(1.0f, 1.0f, 1.0f);

    float3 ambient_light = ka * ia;

    /* diffuse */
    float kd = 0.5f;
    float3 id = float3(1.0f, 1.0f, 1.0f);
    float3 diffuse_light_amount = max(.0f, dot(m_light_direction.xyz, input.normal));

    float3 diffuse_light = kd * diffuse_light_amount * id;

    /* specular */
    float ks = 1.0f;
    float3 is = float3(1.0f, 1.0f, 1.0f);

    float3 reflected_light = reflect(m_light_direction.xyz, input.normal);
    float brightness = 10.0f;
    float specular_light_amount = pow(max(.0f, dot(reflected_light, input.direction_to_cam)), brightness);

    float3 specular_light = ks * specular_light_amount * is;

    /* phong shading */
    float3 result_light = ambient_light + diffuse_light + specular_light;

    return float4(result_light, alpha);
}
```

Figure 77 PixelShader.hlsl

Like in the “VertexShader.hlsl”, pixel shader’s input and constant buffer struct is updated accordingly.
In the main function, Phong Shading is applied by using the equation:

$$I_p = k_a i_a + \sum_{m}^{lights} (k_d \mathrm{i}_{m,d}(\mathrm{N} \times \mathrm{L}) + k_s i_{m,s}(R \times V)^{\alpha})$$

All of the light colors set as white (“float3(1,1,1)”).

Diffuse light amount is calculated with the dot product of the light direction and the normal vector, clamped to the range *[0, 1]*.

The reflected light direction calculated using the “reflect” function.

The specular light amount calculated using the dot product of the reflected light direction and the direction to the camera, raised to the power of the specular exponent.

Finally it returns the result light alongside the alpha.

For the “AppWindow” class, in the “update” method:

```
Matrix4x4 m_light_rot_matrix;
m_light_rot_matrix.setIdentity();
m_light_rot_matrix.setRotationY(m_light_rot_y);

m_light_rot_y += 0.707f * m_delta_time;

cc.m_light_direction = m_light_rot_matrix.getZDirection();
```

Figure 78 Light rotation

A *4x4* matrix is defined to rotate the light direction. The rotation coefficient is calculated with the elapsed time. And a new light direction vector is added to “constant” struct.

Final result can be seen in the Figure 4.102:

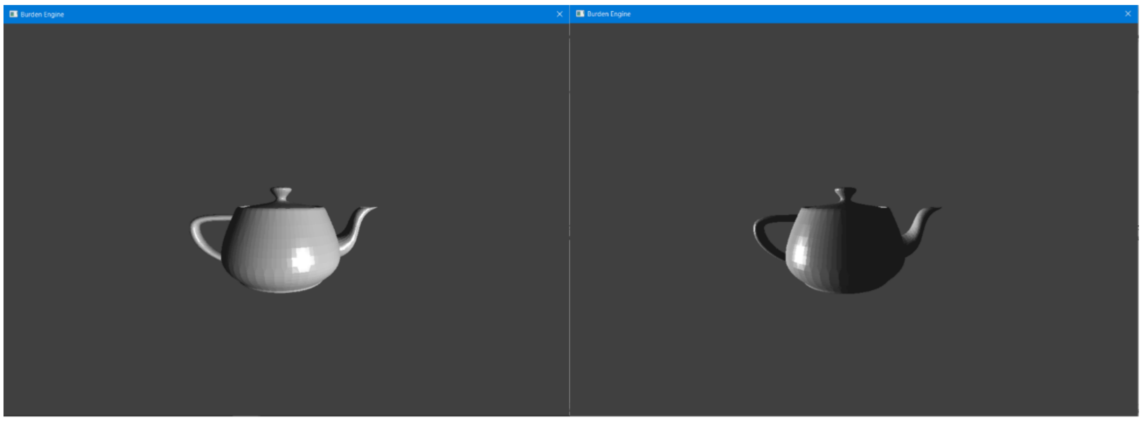

Figure 79 Phong shading on the teapot

4.13 Skybox

A skybox is a technique used in computer graphics to simulate the appearance of a distant background or environment surrounding a 3D scene.

In this section, applying a skybox will be explained. At first, to render the skybox, new constant buffer and fragment shader will be described. So, let's look at the shader file of the skybox:

```
Texture2D Texture : register(t0);
sampler TextureSampler : register(s0);

struct PIXEL_SHADER_INPUT
{
    float4 pos : SV_POSITION;
    float2 texcoord : TEXCOORD0;
    float3 normal : NORMAL0;
    float3 direction_to_cam : TEXCOORD1;
};

float4 psmain(PIXEL_SHADER_INPUT input) : SV_TARGET
{
    return Texture.Sample(TextureSampler, 1.0f - input.texcoord);  // (1.0f -) for the inverting
the skybox texture
}
```

Figure 80 SkyboxShader.hlsl

Skybox shader is very similar to first pixel shaders that explained in previous chapters. It registers the "Texture2D" and "sampler" as the start. Input struct holds position, texture coordinates, normal and direction to camera. Finally, main function simply returns the sample. Texture coordinate subtracted from one to inverting the texture upside down.

In the "AppWindow" class, new methods and member variables defined. New methods are there for the several usage of the same code blocks:

```
        void updateModel();
        void updateCamera();
        void updateSkybox();

        void drawMesh(const MeshPtr& mesh, const VertexShaderPtr& vertex_shader, const
PixelShaderPtr& pixel_shader, const ConstantBufferPtr& constant_buffer, const TexturePtr&
texture);
```

Figure 81 New methods in the AppWindow.h

Three new update methods are defined. All of them is responsible for the update of related components.

"drawMesh" method is taking all the necessary arguments to draw a OBJ to screen. With this method, user will be able to use it whenever a new OBJ draw.

```
	//graphicsEngine components
	PixelShaderPtr m_skybox_shader;
	ConstantBufferPtr m_skybox_constant_buffer;

	//textures
	TexturePtr m_texture1;
	TexturePtr m_texture_skybox;

	//meshes
	MeshPtr m_mesh;
	MeshPtr m_skybox_mesh;

	Matrix4x4 m_world_cam;
	Matrix4x4 m_view_cam;
	Matrix4x4 m_projection_cam;
```

Figure 82 New member variables of the AppWindow

All of the related member variables are written together to read the code easily.

A new pixel shader pointer, which will be hold the “SkyboxShader.hlsl”, is defined. A separate constant buffer pointer is defined for the skybox. Texture and mesh of the skybox will be not the same with other objects, so a separate pointers of these are defined. Finally, two new *4x4* matrices are defined to hold the view and projection camera data.

And, in the source file of the class:

```
void AppWindow::drawMesh(const MeshPtr& mesh, const VertexShaderPtr& vertex_shader, const
PixelShaderPtr& pixel_shader, const ConstantBufferPtr& constant_buffer, const TexturePtr&
texture)
{
	//set constant buffer
	GraphicsEngine::get()->getRenderSystem()->getImmediateDeviceContext()-
>setConstantBuffer(vertex_shader, constant_buffer);
	GraphicsEngine::get()->getRenderSystem()->getImmediateDeviceContext()-
>setConstantBuffer(pixel_shader, constant_buffer);

	//implement the prepared shaders to graphic pipeline to be able to draw
	GraphicsEngine::get()->getRenderSystem()->getImmediateDeviceContext()-
>setVertexShader(vertex_shader);
	GraphicsEngine::get()->getRenderSystem()->getImmediateDeviceContext()-
>setPixelShader(pixel_shader);

	GraphicsEngine::get()->getRenderSystem()->getImmediateDeviceContext()-
>setTexture(pixel_shader, texture);

	//set the vertices of the object to draw
	GraphicsEngine::get()->getRenderSystem()->getImmediateDeviceContext()-
>setVertexBuffer(mesh->getVertexBuffer());

	//set the indices of the object
	GraphicsEngine::get()->getRenderSystem()->getImmediateDeviceContext()-
>setIndexBuffer(mesh->getIndexBuffer());

	/* drawing */
	GraphicsEngine::get()->getRenderSystem()->getImmediateDeviceContext()-
>drawIndexedTriangleList(mesh->getIndexBuffer()->getSizeIndexList(), 0, 0);
}
```

Figure 83 drawMesh

“drawMesh” method is the same codes that written in the Chapter 4.11. It simplifies the drawing the separate OBJ in one scene.

```
void AppWindow::updateCamera()
{
        Matrix4x4 world_cam, temp;
        world_cam.setIdentityMatrix();

        temp.setIdentityMatrix();
        temp.setRotationXMatrix(m_rotation_x);
        world_cam *= temp;

        temp.setIdentityMatrix();
        temp.setRotationYMatrix(m_rotation_y);
        world_cam *= temp;

        Vector3D new_position = m_world_cam.getTranslation() + (world_cam.getZDirection() *
(m_forward * .1f) + world_cam.getXDirection() * (m_horizontal_move_coefficient * .1f));

        world_cam.setTranslationMatrix(new_position);

        m_world_cam = world_cam;

        world_cam.inverse();

        m_view_cam = world_cam;

        int width = this->getClientWindowRect().right - this->getClientWindowRect().left;
        int height = this->getClientWindowRect().bottom - this->getClientWindowRect().top;

        m_projection_cam.setPerspectiveFovLH(1.57f, ((float)width / (float)height), .1f, 100.0f);
}

void AppWindow::updateModel()
{
        constant model_constant;
        Matrix4x4 m_light_rotation_matrix;

        m_light_rotation_matrix.setIdentityMatrix();
        m_light_rotation_matrix.setRotationYMatrix(m_light_rotation_y);

        m_light_rotation_y += .785f * m_delta_time;  // pi / 4 * delta_t

        model_constant.m_world.setIdentityMatrix();
        model_constant.m_view = m_view_cam;
        model_constant.m_projection = m_projection_cam;
        model_constant.m_cam_position = m_world_cam.getTranslation();
        model_constant.m_light_direction = m_light_rotation_matrix.getZDirection();

        m_constant_buffer->update(GraphicsEngine::get()->getRenderSystem()-
>getImmediateDeviceContext(), &model_constant);  //constant buffer updating
}
```

Figure 84 Update methods for camera and model

Like “drawMesh”, both of the “updateModel” and “updateCamera” is holding the same code blocks that written in the “update” method.

These methods must be called in the “update” method in the right order to avoid rendering bugs.

```
void AppWindow::updateSkybox()
{
	constant skybox_constant;

	skybox_constant.m_world.setIdentityMatrix();
	skybox_constant.m_world.setScaleMatrix(Vector3D(100.0f, 100.0f, 100.0f));
	skybox_constant.m_world.setTranslationMatrix(m_world_cam.getTranslation());
	skybox_constant.m_view = m_view_cam;
	skybox_constant.m_projection = m_projection_cam;

	m_skybox_constant_buffer->update(GraphicsEngine::get()->getRenderSystem()-
>getImmediateDeviceContext(), &skybox_constant);  //constant buffer updating
}
```

Figure 85 updateSkybox

“updateSkybox” method has the same structure as the other update methods. Most important difference is the translation of the skybox. To make the user feel under sky, the center of the OBJ that will be drawn for the skybox is matched with the position of the camera. And whenever the camera moves, the skybox OBJ will be follow the same translation.

```
	m_texture_skybox = GraphicsEngine::get()->getTextureManager()-
>createTextureFromFile(L"Assets\\Textures\\sky.jpg");

	m_skybox_mesh = GraphicsEngine::get()->getMeshManager()-
>createMeshFromFile(L"Assets\\Meshes\\sphere.obj");
```

Figure 86 Skybox texture and mesh

The skybox will be a sphere OBJ, which obtained from NVIDIA FX. The texture of the skybox will be rendered inside of the sphere. So, “ceateFromFile” methods are called in the “onCreate” method.

```
	/* SKYBOX SHADER */
	GraphicsEngine::get()->getRenderSystem()->compilePixelShader(L"SkyboxShader.hlsl",
"psmain", &shader_byte_code, &size_shader);
	m_skybox_shader = GraphicsEngine::get()->getRenderSystem()-
>createPixelShader(shader_byte_code, size_shader);  //is a pixelShaderPtr
	GraphicsEngine::get()->getRenderSystem()->releaseCompiledShader();

	m_skybox_constant_buffer = GraphicsEngine::get()->getRenderSystem()-
>createConstantBuffer(&const_obj, sizeof(constant));  //skybox const buff
```

Figure 87 Skybox shader compilation

The skybox shader compilation is done exactly like the pixel shader. And the skybox constant buffer created like the other constant buffer.

```
void AppWindow::update()
{
	updateCamera();
	updateModel();
	updateSkybox();
}
```

Figure 88 update method

In “update”, the update methods called in the order camera, model and skybox.

```
        update();

        //rendering the model
        GraphicsEngine::get()->getRenderSystem()->setRasterizerState(false);
        drawMesh(m_mesh, m_vertex_shader, m_pixel_shader, m_constant_buffer, m_texture1);

        //rendering the skybox mesh which is a sphere
        GraphicsEngine::get()->getRenderSystem()->setRasterizerState(true);
        drawMesh(m_skybox_mesh, m_vertex_shader, m_skybox_shader, m_skybox_constant_buffer,
m_texture_skybox);
```

Figure 89 In the onUpdate method

In the "onUpdate" method, after the "update" method is called, rasterizer states are determined to render the texture outside or the inside of the OBJ. The skybox texture will be rendered inside of the sphere, so rasterizer state method takes the true argument. This method is explained in the related chapter.

After setting the rasterizer state, "drawMesh" method is called to render the OBJs.

Figure 4.113 shows the teapot in different distances and angles. As can be seen, the skybox is rendered correctly.

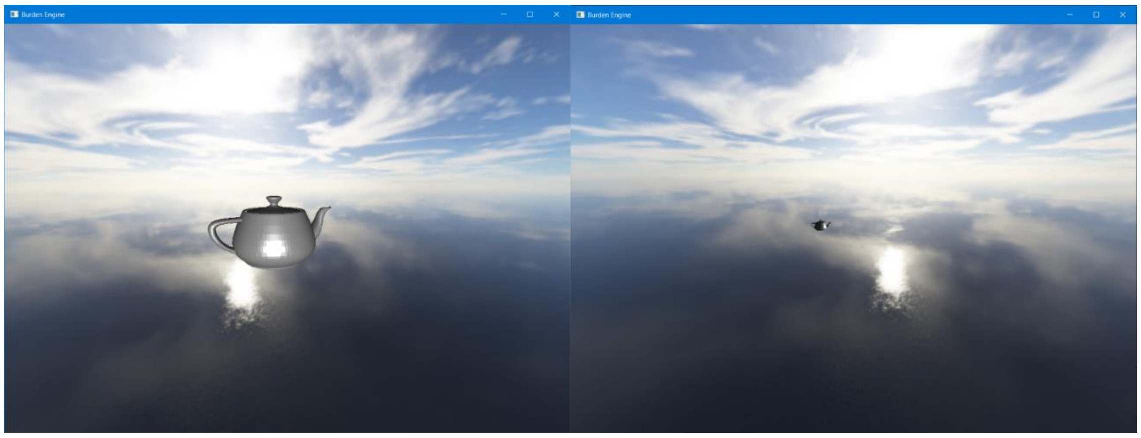

Figure 90 Rendering the skybox

4.14 Fullscreen

This section explains allowing resizing and make the window fullscreen.

Let's delve into the "AppWindow"'s header file:

```
        virtual void onSize() override;
        void render();
        //state flags
        bool m_resize_state = false;
        bool m_fullscreen_flag = false;
```

Figure 91 Inherited method and app state flags

"onSize" method is inherited from the "Window" class. A new method called "render" is defined with the similar reasons of the update components methods. And two Boolean variables are defined to control the window states.

Let’s check the source file:

```
void AppWindow::render()
{
	/* clearing the render target */
	GraphicsEngine::get()->getRenderSystem()->getImmediateDeviceContext()-
>clearRenderTargetColor(this->m_swap_chain, .25f, .25f, .25f, 1);  //0.125f, 0.025f, 0.125f

	/* set viewport of render target (which to draw) */
	RECT rc = this->getClientWindowRect();
	GraphicsEngine::get()->getRenderSystem()->getImmediateDeviceContext()-
>setViewportSize(rc.right - rc.left, rc.bottom - rc.top);

	/* constant buffer updates */
	// flag update
	update();

	//rendering the model
	GraphicsEngine::get()->getRenderSystem()->setRasterizerState(false);
	drawMesh(m_mesh, m_vertex_shader, m_pixel_shader, m_constant_buffer, m_texture1);

	//rendering the skybox mesh which is a sphere
	GraphicsEngine::get()->getRenderSystem()->setRasterizerState(true);
	drawMesh(m_skybox_mesh, m_vertex_shader, m_skybox_shader, m_skybox_constant_buffer,
m_texture_skybox);

	m_swap_chain->present(false);

	m_old_delta = m_new_delta;
	m_new_delta = ::GetTickCount();
	m_delta_time = (m_old_delta) ? ((m_new_delta - m_old_delta) / 1000.0f) : 0;
}
```

Figure 92 render method

As can be seen, “render” method has the code block that written in the “onUpdate” method. With this, user can be able to call only “Window” and “InputSystem” classes’ updates and “render” in the “onUpdate”.

```
void AppWindow::onSize()
{
	RECT rc = this->getClientWindowRect();
	m_swap_chain->resize(rc.right, rc.bottom);
	this->render();
}
```

Figure 93 onSize

“onSize” method get’s the window client area rectangle of the window. After that, “resize” method is called from swap chain to allowing the resizing. Finally “render” method is called.

```
        if (key == 'P')
        {
                m_resize_state = (m_resize_state) ? false : true;
                InputSystem::get()->showCursor(!m_resize_state);
        }
        else if (key == VK_F11)
        {
                m_fullscreen_flag = (m_fullscreen_flag) ? false : true;

                RECT screen_size = this->getScreenSize();

                m_swap_chain->setFullscreen(m_fullscreen_flag, screen_size.right,
screen_size.bottom);
        }
```

Figure 94 Assigning new keys to change states

In “onKeyUp” method, two new keys defined. “P” is for the change to resize state. With this state, the cursor will be shown and mouse move will not effect anything. To make mouse move disabled, a simple condition block can be added to “onMouseMove” method’s beginning to check the resize state flag. If the flag gives the resize state true, a return is written to terminate the process of “onMouseMove”.

“F11” key is for the change fullscreen state. When pressed, the app window will go fullscreen or vice versa.

Results can be seen in the Figure 4.118:

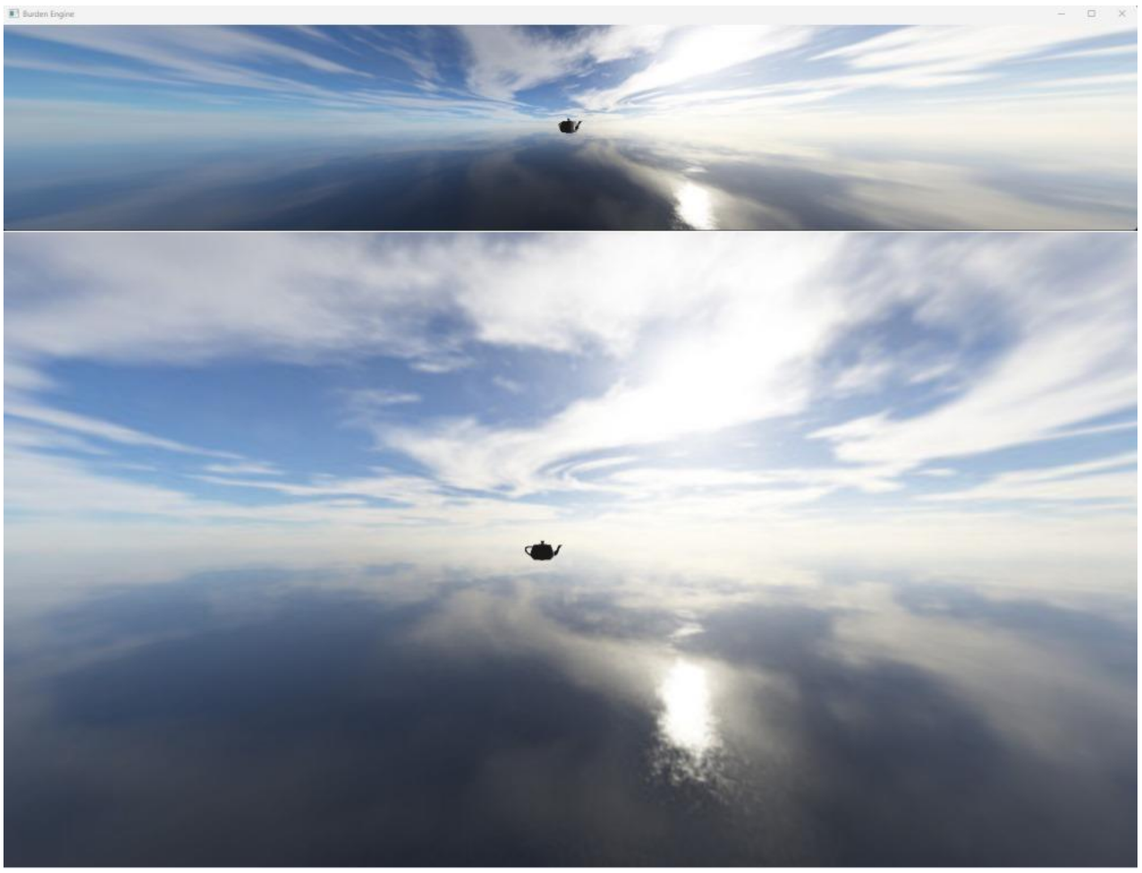

Figure 95 Resized and fullscreen windows

4.15 Point Light

In Section 4.12, Phong Shading is used to apply directional lighting. In this section, applying the point light is described.

A directional light is a type of light source that emits illumination uniformly in a specific direction, simulating the characteristics of sunlight or a distant light source. In computer graphics and 3D rendering, directional lights are implemented by emitting parallel light rays, making them ideal for scenes where the light source is significantly larger and farther away than the objects it illuminates. Commonly used to simulate outdoor environments, directional lights provide consistent illumination across the entire scene. In shader programs, the direction of the light is often represented as a normalized vector, encapsulating the light's orientation in 3D space.

In contrast, a point light is a light source that emits light uniformly in all directions from a specific point in space. Point lights are characterized by their finite distance and attenuation of light intensity following the inverse square law. They are commonly employed to simulate localized light sources such as bulbs or lamps in indoor scenes, where the light source is relatively close to the objects it illuminates. The position of the point light is a critical factor in its implementation, and attenuation factors are frequently utilized to model how light intensity diminishes with distance. Point lights contribute to realistic lighting effects, including specular highlights, enhancing the visual fidelity of rendered scenes.

Two new shader scripts are created just for the point light. These shaders nearly the same with the previous ones. Main difference is the operations for the Phong Shading, which will be explained while examining the pixel shader script. Let’s look at the newly created shaders: (GoldSrc Prototype Repository, n.d.)

```
struct VERTEX_SHADER_INPUT
{
    float4 pos : POSITION0;
    float2 texcoord : TEXCOORD0;
    float3 normal : NORMAL0;
};
struct VERTEX_SHADER_OUTPUT
{
    float4 pos : SV_POSITION;
    float2 texcoord : TEXCOORD0;
    float3 normal : NORMAL0;
    float3 world_position : TEXCOORD1;
};
cbuffer constant : register(b0)
{
    row_major float4x4 m_world;
    row_major float4x4 m_view;
    row_major float4x4 m_projection;
    float4 m_light_direction;
    float4 m_cam_position;
    float4 m_light_position;
    float m_light_radius;
    float m_time;
};
VERTEX_SHADER_OUTPUT vsmain(VERTEX_SHADER_INPUT input)
{
    VERTEX_SHADER_OUTPUT output = (VERTEX_SHADER_OUTPUT) 0;
    /* WORLD SPACE */
    output.pos = mul(input.pos, m_world); //mul(): matrix multiplication method
    output.world_position = output.pos.xyz;
    /* VIEW SPACE */
    output.pos = mul(output.pos, m_view);
    /* PROJECTION SPACE */
    output.pos = mul(output.pos, m_projection);
    output.texcoord = input.texcoord;
    output.normal = input.normal;

    return output;
}
```

Figure 96 PointLightVertexShader.hlsl

New members for the constant buffer is defined. Point light needs the position and radius data to work. “m_time” is defined to apply simple translation animation with the useage of elapsed time.

```
Texture2D Texture : register(t0);
sampler TextureSampler : register(s0);

struct PIXEL_SHADER_INPUT
{
    float4 pos : SV_POSITION;
    float2 texcoord : TEXCOORD0;
    float3 normal : NORMAL0;
    float3 world_position : TEXCOORD1;
};

cbuffer constant : register(b0)
{
    row_major float4x4 m_world;
    row_major float4x4 m_view;
    row_major float4x4 m_projection;
    float4 m_light_direction;
    float4 m_cam_position;
    float4 m_light_position;
    float m_light_radius;
    float m_time;
};
```

Figure 97 Registerations and structs of PointLightPixelShader.hlsl

Constant buffer structs must have the same members with definitions of them in the other shaders. Texture and sampler is registered to render a single texture.

```
float4 psmain(PIXEL_SHADER_INPUT input) : SV_TARGET
{
    float alpha = 1.0f;

    float4 texture_color = Texture.Sample(TextureSampler, (1.0f - input.texcoord) * 2.0f);

    /* ambient */
    float ka = 1.5f;
    float3 ia = float3(.09f, .08f, .07f);
    ia *= (texture_color.rgb);

    float3 ambient_light = ka * ia;

    /* diffuse */
    float kd = 0.5f;
    float3 id = float3(1.0f, 1.0f, 1.0f);

    id *= (texture_color.rgb);

    float3 light_direction = normalize(m_light_position.xyz - input.world_position.xyz);

    float distance_from_light_to_obj = length(m_light_position.xyz - input.world_position.xyz);
    float fade_area = max(0, distance_from_light_to_obj - m_light_radius);

    //calculating the attenuation
    float constant_ = 1.0f;
    float linear_ = 1.0f;
    float quadratic_ = 1.0f;
                            //const                  linear                          quadratic
    float attenuation = constant_ + (fade_area * linear_) + (fade_area * fade_area * quadratic_);

    float3 diffuse_light_amount = max(0, dot(light_direction.xyz, input.normal));

    float3 diffuse_light = (kd * id * diffuse_light_amount) / attenuation;

    /* specular */
    float ks = 1.0;
    float3 is = float3(1.0f, 1.0f, 1.0f);

    float3 direction_to_cam = normalize(input.world_position.xyz - m_cam_position.xyz);

    float3 reflected_light = reflect(light_direction.xyz, input.normal);
    float brightness = 50.0f;
    float specular_light_amount = pow(max(.0f, dot(reflected_light, direction_to_cam)), brightness);

    float3 specular_light = (ks * specular_light_amount * is) / attenuation;

    /* phong shading */
    float3 result_light = ambient_light + diffuse_light + specular_light;


    return float4(result_light, alpha);
}
```

Figure 98 Main function of the PointLightPixelShader.hlsl

Main function of the pixel shader can be inspected in three parts:

The ambient reflection represents the constant background illumination and is calculated by multiplying the ambient reflection coefficient (“ka”) with the ambient light color (“ia”). The ambient light color is modulated with the color sampled from the texture (“texture_color.rgb”). The result is the ambient light contribution, representing a uniform illumination across all surfaces.

The diffuse reflection is determined by multiplying the diffuse reflection coefficient (“kd”) with the diffuse light color (“id”). The diffuse light color is modulated with the color sampled from the texture. The direction from the pixel to the light source is calculated (“light_direction”). Attenuation is applied based on the distance between the light source and the object being illuminated, considering a fade area. The diffuse light amount is computed based on the angle between the light direction and the surface normal (“diffuse_light_amount”). The final diffuse light contribution is adjusted by the attenuation factor.

Specular reflection represents the shiny highlights on a surface. It is determined by multiplying the specular reflection coefficient (“ks”) with the specular light color (“is”). The specular light color is modulated with the color sampled from the texture. The direction from the pixel to the camera is calculated (“direction_to_cam”). The reflection direction is computed using the “reflect” function. Specular brightness is controlled by the “brightness” factor. The specular light amount is calculated using the dot product between the reflected light direction and the direction to the camera. The specular light contribution is adjusted by the attenuation factor.

So, in the “AppWindow” class:

```
	float m_time = .0f;
```

Figure 99 New member variable in the AppWindow.h

“m_time” is defined to hold the elapsed time. In the source file:

```
struct constant
{
	Matrix4x4 m_world;
	Matrix4x4 m_view;
	Matrix4x4 m_projection;
	Vector4D m_light_direction;
	Vector4D m_cam_position;
	Vector4D m_light_position = Vector4D(0,1,0,0);
	float m_light_radius = 4.0f;
	float m_time = .0f;
};
```

Figure 100 constant struct

“constant” struct is updated accordingly to the definitions that been made in the shader scripts.

```
        float distance_from_origin = 1.0f;
        model_constant.m_light_position = Vector4D(cos(m_light_rotation_y) *
distance_from_origin, 1.0f, sin(m_light_rotation_y) * distance_from_origin, 1.0f);
        model_constant.m_light_radius = m_light_radius;

        model_constant.m_light_direction = m_light_rotation_matrix.getZDirection();
        model_constant.m_time = m_time;
```

Figure 101 updateModel

“distance_from_origin” is set to one, representing the distance of the light source from the origin. Using trigonometric functions (cos and sin) to calculate the x and z coordinates of the light position based on the rotation angle. The resulting position is stored in “model_constant.m_light_position” as a “Vector4D” with the w-component set to one.

“m_light_radius” is assigned to “model_constant.m_light_radius”, representing the radius of the light source. This parameter is using to simulate the influence area of the light.

“m_light_rotation_matrix” is the rotation matrix and getZDirection() retrieves the forward direction after applying the rotation. The resulting direction is assigned to “model_constant.m_light_direction”. This vector indicates the direction in which the light source is pointing.

“m_time” is assigned to “model_constant.m_time”, indicating a time parameter.

```
        /* VERTEX SHADER */
        GraphicsEngine::get()->getRenderSystem()-
>compileVertexShader(L"PointLightVertexShader.hlsl", "vsmain", &shader_byte_code, &size_shader);
        m_vertex_shader = GraphicsEngine::get()->getRenderSystem()-
>createVertexShader(shader_byte_code, size_shader);
        GraphicsEngine::get()->getRenderSystem()->releaseCompiledShader();

        /* PIXEL SHADER */
        GraphicsEngine::get()->getRenderSystem()-
>compilePixelShader(L"PointLightPixelShader.hlsl", "psmain", &shader_byte_code, &size_shader);
        m_pixel_shader = GraphicsEngine::get()->getRenderSystem()-
>createPixelShader(shader_byte_code, size_shader);
        GraphicsEngine::get()->getRenderSystem()->releaseCompiledShader();
```

Figure 102 Shader compilation

In the “onCreate” method, new shaders (“PointLightShaders”) send to compile method.

Also, skybox texture is changed to render stars. Another texture pointer is holding a brick texture.

The mesh pointer holds a plane and a sphere, donut shape and cube together. This model is created with the NVIDIA FX.

So, a point light will be placed on the middle of the shapes and draws a circle as animation. Final result can be seen in the Figure :

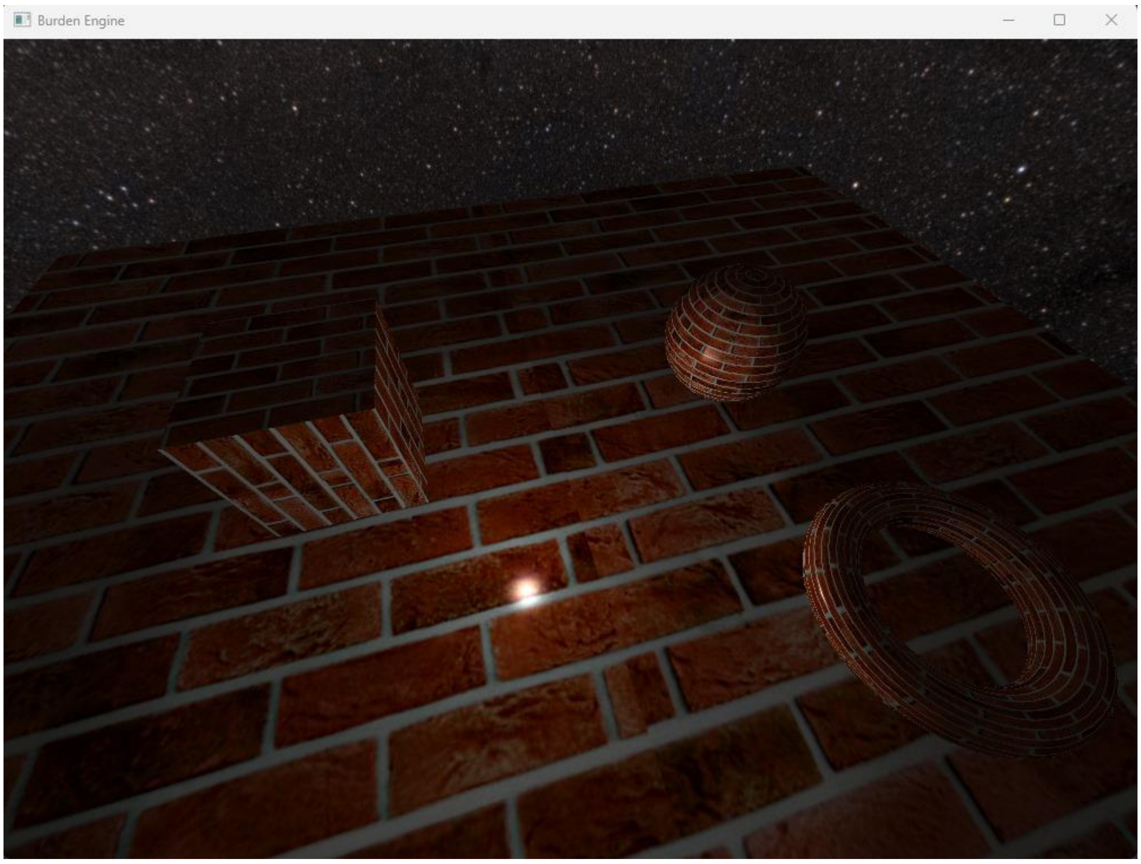

Figure 103 Point light demonstration

```
else if (key == VK_UP)
{
        m_light_radius += 4.0f * m_delta_time;
}
else if (key == VK_DOWN)
{
        m_light_radius -= 4.0f * m_delta_time;
}
```

Figure 104 New key assigments

Two new key assigned to change the light radius scale. Up arrow key will increment the radius and down arrow key will descend it.

The change of the radius is shown in the Figure :

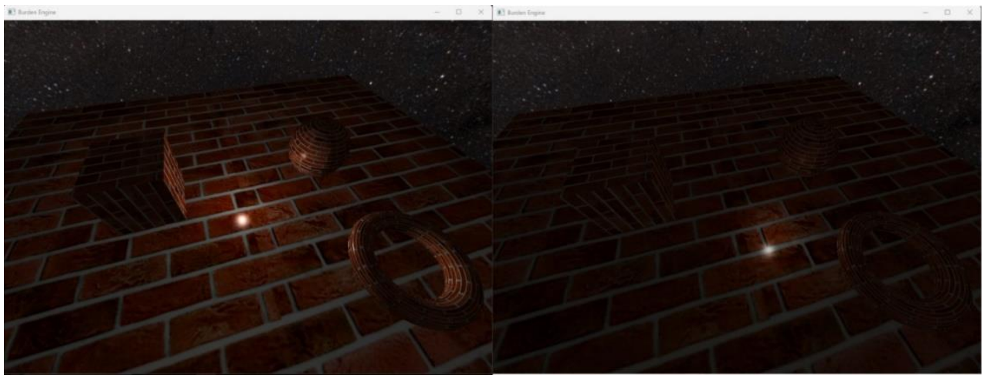

Figure 105 Changing the radius of the point light

4.16 Multi-Layering and First Technology Demo

In this section, a multi-layered and animated world design and application is explained.

In this model, directional lighting is used. So, the examined HLSL scripts are not "PointLightShader"s.

At first, in "AppWindow" class header, new texture pointers are defined as:

```
//textures
TexturePtr m_world_morning_tex;
TexturePtr m_world_night_tex;
TexturePtr m_world_clouds_tex;
TexturePtr m_world_specular_tex;
```

Figure 106 Defining textures

Morning texture will hold the morning state of the world. Likewise, night will hold the night state. Clouds will be rendered as another layer and will be added rotation animation. Specular will hold the specular map of the world. With this texture, specular reflection will not be rendered on the land parts of the world.

```
Texture2D WorldMorning : register(t0);
sampler WorldMorningSampler : register(s0);

Texture2D WorldSpecular : register(t1);
sampler WorldSpecularSampler : register(s1);

Texture2D WorldClouds : register(t2);
sampler WorldCloudsSampler : register(s2);

Texture2D WorldNight : register(t3);
sampler WorldNightSampler : register(s3);
```

Figure 107 Registering the textures and samplers in PixelShader.hlsl

All of the defined textures and samplers of them are registered in "PixelShader.hlsl" first. Each one has its own unique address.

```
float4 psmain(PIXEL_SHADER_INPUT input) : SV_TARGET
{
    float alpha = 1.0f;
    float4 world_morning = WorldMorning.Sample(WorldMorningSampler, 1.0f - input.texcoord);
    float world_specular = WorldSpecular.Sample(WorldSpecularSampler, 1.0f - input.texcoord).r;
    float world_clouds = WorldClouds.Sample(WorldCloudsSampler, 1.0f - input.texcoord + float2(m_time / 50.0f,
0)).r;
    float4 world_night = WorldNight.Sample(WorldNightSampler, 1.0f - input.texcoord);
    /* ambient */
    float ka = 1.5f;
    float3 ia = float3(.09f, .08f, .07f);
    ia *= (world_morning.rgb);
    float3 ambient_light = ka * ia;
    /* diffuse */
    float kd = 0.5f;
    float3 id_morning = float3(1.0f, 1.0f, 1.0f);
    id_morning *= (world_morning.rgb + world_clouds);
    float3 id_night = float3(1.0f, 1.0f, 1.0f);
    id_night *= (world_night.rgb + world_clouds * .2f);
    float3 diffuse_light_amount = dot(m_light_direction.xyz, input.normal);
    float3 id = lerp(id_night, id_morning, (diffuse_light_amount + 1.0f) / 2.0f);
    float3 diffuse_light = kd * id;
    /* specular */
    float ks = world_specular;
    float3 is = float3(1.0f, 1.0f, 1.0f);
    float3 reflected_light = reflect(m_light_direction.xyz, input.normal);
    float brightness = 10.0f;
    float specular_light_amount = pow(max(.0f, dot(reflected_light, input.direction_to_cam)), brightness);
    float3 specular_light = ks * specular_light_amount * is;
    /* phong shading */
    float3 result_light = ambient_light + diffuse_light + specular_light;
    return float4(result_light, alpha);
}
```

Figure 108 PixelShader main

The alpha variable, initialized to 1.0, establishes full opacity for the pixel color.

Various textures representing different aspects of the World's surface are sampled using corresponding samplers. These include the morning texture (“WorldMorning”), specular highlights (“WorldSpecular”), clouds (“WorldClouds”), and the night texture (“WorldNigh”t). Each texture is adjusted using its respective sampler. The resulting colors are stored in variables.

Ambient lighting is computed, reflecting the overall illumination of the scene. The ambient reflection coefficient is multiplied by the ambient light color. The ambient light color is further influenced by the morning texture. The resulting ambient light is stored in the “ambient_light” variable.

Diffuse lighting, representing the scattered light on the surface, is calculated. The dot product between the light direction and the surface normal is computed. This value influences the blending of the sampled morning and clouds textures. The diffuse reflection coefficient modulates this result, producing the “diffuse_light” component.

Specular lighting, responsible for highlight reflections, is determined. The reflected light direction is computed using the light direction and surface normal. The brightness of the specular highlight is controlled by an exponent (“brightness”). The specular reflection coefficient is applied to this result. The resulting specular light is stored in the “specular_light” variable.

The Phong shading model is employed to combine ambient, diffuse, and specular lighting components. This model creates a visually appealing and realistic shading effect on the Earth's surface. The combined result is stored in the “result_light” variable.

The final color for the pixel is obtained by summing up the ambient, diffuse, and specular lighting components. This comprehensive representation of the World’s surface illumination is encapsulated in a float4 constructor, with the alpha channel set to the pre-initialized value of 1.0.

With these calculations in the pixel shader, four textures will be drawn to one sphere OBJ. All of the textures can be able to modify separately. For example, the clouds will rotate around the globe. Directional light will be rotating (application explained in Chapter 4.12) and the parts that gets the light will render the morning texture while the parts do not get the light will be rendered as night. For the smooth pass between the morning and night, linear interpolation is used. The land will not reflect the specular due to the specular map texture.

```
TexturePtr texture_list[4];
texture_list[0] = m_world_morning_tex;
texture_list[1] = m_world_specular_tex;
texture_list[2] = m_world_clouds_tex;
texture_list[3] = m_world_night_tex;

drawMesh(m_mesh, m_vertex_shader, m_pixel_shader, m_constant_buffer, texture_list, 4);
```

Figure 109 Texture list

All the textures stored in a list to render one by one with “drawMesh” method.

Final result can be seen in the Figure 4.133:

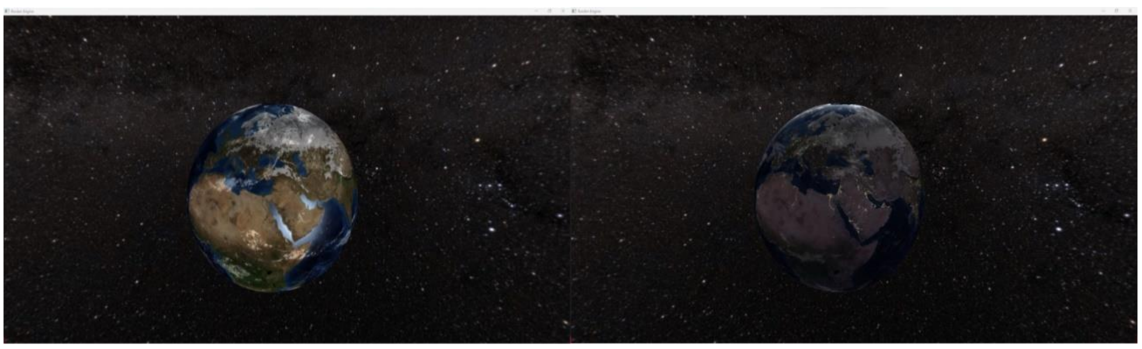
Figure 110 First demo