HOGESCHOOL VAN AMSTERDAM

Report Graphics Programming

Assignments on different graphic topics

Rosa Corstjens, 500702627

April 2017

1. INTRODUCTION

This report and the corresponding projects have been created in commission by the Hogeschool van Amsterdam for the semester Game Technology, course Graphics Programming. The goal of these assignments was to gain knowledge about programming graphics in C++ for DirectX 11 with use of the Luna library. Therefore, each assignment has its own graphics topic to focus on and learn more about. The following topics with corresponding assignments are addressed:

- Primitives: rendering simple shapes;
- Textures: rendering a model of your mobile phone with textures;
- Lighting: rendering different light effects;
- Shading: implementing different shaders.

In this report, each of the listed assignments will be discussed. For each assignment the requirement will first be given. All assignments have three levels of requirements (basic, intermediate and advanced). To meet higher requirements, first the lower requirements must be met. Therefore, all levels of requirements will be listed for each assignment. Next, design choices and implemented techniques will be addressed. The chapters in this report follow the list of topics.

2. PRIMITIVES

Simple objects, also called primitives, can be defined by hand. For example, given a coordinate system, one can tell were the vertices and edges of a square should be placed to draw a square. More complex objects are called meshes and they are often created by artists. This chapter discusses the implementation of a program that generates and renders a primitive, namely a pyramid. Since this assignment is the first, its main goal is to get familiar with the library. The requirements are as following:

- Basic: Render a fixed 3D shape that is not included in the Luna library;
- Intermediate: Add dynamic user input to enable configuration of the primitive;
- Advanced: Render the primitive on the GPU with use of the geometry shader.

I choose a regular pyramid as my primitive. Since the different levels of requirements follow each other up, I worked incrementally on this project. The following sections discuss how the project progressed and what choices had to be made.

2.1 PYRAMID GENERATION

To start up the project, an empty scene has been set up with help of the Luna library and the book '3D Game Programming with DirectX11' by Frank D. Luna (from now on I will refer to this book as Luna's book). This project consisted of a few methods for initializing, resizing the window, updating and drawing the scene, reading mouse input. This most basic DirectX application will be used as a starting point for all further assignments. Since I did not fill the vertex and index buffers at this point, the scene was empty. Next the pyramid had to be generated and rendered. This section further discusses briefly how this algorithm works.

Since the vertex and index buffers don't need to change for this first implementation, immutable buffers were used and filled with generated data. The data for the vertices and indices were generated with the function **CreatePyramid** from the class **PyramidGenerator**. This function expects a radius, a height, an amount of sides as input and a pointer to the data to return, a **Meshdata** structure with the vertices and indices needed to draw the given pyramid. The buffers are filled with the data and created, after which the pyramid can be rendered every frame.

To generate the pyramid, the following algorithm has been implemented. The relevant code is included in code snippet 01:

1. Clear the vectors that hold the vertex and index data of the **Meshdata**;
2. Create the apex of the pyramid (the top) by pushing a vertex at position (0, height, 0) to the vertex vector;
3. Create the vertices at the sides of the pyramid and push them to the vertex vector. The x- and z-coordinates are calculated with cosine and sine based on the radius and amount of sides. This way the vertices are defined anti-clockwise from a top view of the pyramid. The y-coordinate is always 0;
4. Create the center of the base of the pyramid at (0, 0, 0) and push it to the vertex vector;
5. For each side minus the last one, with i as iterator, define two triangles anti- counterclockwise:
   a. The side triangle, by connecting the apex (vertex 0) with i + 2 and i + 1 in that order;

   b. The base triangle, by connecting the center of the base (last in list) with i + 1 and i + 2 in that order.
6. Define the last triangle outside a for loop since it connects the last side with the first:
   1. The side triangle, by connecting the apex with the first side vertex (vertex 1) and the last side vertex (vertex last – 1) in that order;
   2. The base triangle, by connecting the center of the base with the last side vertex and the first side vertex in that order.

The function that generates the pyramid is called before the vertex and index buffer are defined and created. Vectors are filled with the vertex and buffer data which is passed at creation time of both buffers.

```
// clear the meshdata member variables.
meshdata.Vertices.clear();
meshdata.Indices.clear();

// create apex
meshdata.Vertices.push_back(Vertex(XMFLOAT3(+0.0f, height, +0.0f)));

// calculate all other vertices
for (int i = 0; i < sides; ++i)
{
        meshdata.Vertices.push_back(Vertex(XMFLOAT3(radius * cosf(2 * MathHelper::Pi * i / sides),
+0.0f, radius * sinf(2 * MathHelper::Pi * i / sides))));
}

// create center of base
meshdata.Vertices.push_back(Vertex(XMFLOAT3(+0.0f, +0.0f, +0.0f)));

int vertexcount = meshdata.Vertices.size() - 1;

// amount of indices = (sides * 3) * 2
// since every side needs three indices for the side
// and three indices for the bottom.

// so for each side a side and bottom triangle will be constructed
for (int i = 0; i < sides - 1; ++i)
{
        // first create side triangle
        meshdata.Indices.push_back(0);          // each triangle starts with apex
        meshdata.Indices.push_back(i + 2);
        meshdata.Indices.push_back(i + 1);

        // than create bottom triangle
        meshdata.Indices.push_back(vertexcount);// each triangle starts with center base
        meshdata.Indices.push_back(i + 1);
        meshdata.Indices.push_back(i + 2);
}

// add the last two triangles for the last side separately.
// first create side triangle
meshdata.Indices.push_back(0);                  // each triangle starts with apex
meshdata.Indices.push_back(1);
meshdata.Indices.push_back(vertexcount - 1);

// then create bottom triangle
meshdata.Indices.push_back(vertexcount);        // each triangle starts with center base
meshdata.Indices.push_back(vertexcount - 1);
meshdata.Indices.push_back(1);
```

Code snippet 01 – Generating the pyramid's vertices and indices.

2.2 DYNAMIC USER INPUT

The last section discussed the algorithm for defining the pyramid. The algorithm already takes multiple parameters as input – height, radius and amount of sides – and generates a different pyramid based on those. The next step is to start the program with a default pyramid and enable the user to change the parameters on runtime. To support this, the following changes had to be made:

- The main class **Primitives** needs a few extra member variables to keep track of the current parameters and update the pyramid;

- The vertex and index buffers must be dynamic so that their content can change and be initialized with an appropriate size;
- In the update loop the keyboard input must be captured and reacted upon;
- When input has been detected, a new pyramid must be generated and the vertex and index buffers must be updated.

Each of these changes will be addressed in this section.

Firstly, the main class **Primitives** gets new member variables:

- Five Booleans to keep track of the input state last frame per button, since three buttons (1, 2 and 3) are used to set the parameter to change and two buttons (left and right arrow keys) are used to increase and decrease the value of the selected parameter;
- An integer to keep track of the current selected parameter to change;
- Two variables to store the amount to decrease or increase the selected parameter by, since this value is different for floating point values than it is for integers;
- Four variables to store the minimum values for the parameters and the maximum value for the amount of sides. Only the amount of sides has a maximum value, since this is needed for creating the vertex and index buffers, as discussed later in this section;
- Two arrays of floating point values to store the current and last set of parameters. This is used to change the parameters and to see if the pyramid needs to be updated instead of updating it every frame;
- An instance of the **PrimitiveGenerator** class and the **Meshdata** structure to enable reusing them for each time the pyramid needs to be recalculated.

Second the buffers need to be changed. They need to be dynamic, which enables changing their content on runtime. Note that only their content can be changed, not their size. Therefore, the method **InitGeometryBuffers** starts with generating a pyramid with maximum value for the amount of sides. Since the other two parameters, height and radius, won't alter the amount of vertices and indices needed to draw the pyramid, it doesn't matter what their values are. The returned pyramid holds the maximum amount of vertices and indices needed given the maximum value for the amount of sides. These values will be saved for later. Next, the descriptions for the vertex and index buffer are filled in. To enable dynamic buffers the following variables should be set like this:

- **Usage** is D3D11_USAGE_DYNAMIC;
- **ByteWidth** is the size of the **Vertex** structure times the maximum amount of vertices in the case of the vertex buffer. In the case of the index buffer it is the size of the UINT times the maximum amount of indices.
- **CPUAccessFlags** is D3D11_CPU_ACCESS_ WRITE so that the CPU can change the contents.

Finally, the method **InitGeometryBuffers** calls **SetGeometryBuffers**, discussed later in this section, which actually fills the buffers with data: in this case the default. Note that the **InitGeometryBuffers** and **SetGeometryBuffers** methods replace the method **BuildGeometryBuffers**. Creating the buffers is done in the first method, setting their contents in the second. This way, the content can easily be set without redefining the buffers.

To enable actually changing the configuration of the pyramid, input must be captured. This is added in the update loop of the application. For every used button the following steps are executed:

- If the given key is pressed, proceed, else set the Boolean for pressed last frame of this key to false;
- If the given key was not pressed last frame, proceed, else break;
- React on the input and set the Boolean for pressed last frame of this key to true.

For the buttons 1, 2 and 3, the variable **inputstate** will be set to the corresponding parameter, so that another parameter can be altered by the user. For the buttons left and right arrow the value of the selected parameter is altered. After all input is captured, the function **BoundInputParams** is called to ensure all values lie between their minimum and maximum. The described steps altered the values of the array containing the current parameters. If differences occur between this array and the array containing the array with parameters from last frame, the program updates the values in the array with parameters from last frame. At this point the new pyramid based on the given parameters will be generated.

The generation of a new pyramid when needed is handled by the function **SetGeometryBuffers**. In **SetGeometryBuffers** a pyramid with the current parameters is generated, after which the vertex and index data is read from it and mapped to the dynamic buffers. See code snippet 02 for the code that unmaps and maps the buffers.
With these alternations I was able to give input and change the radius, height and amount of sides of the pyramid on runtime. The next step is to this shape on the GPU.

```
// create a new pyramid
_generator.CreatePyramid(_lastParams[0], _lastParams[1], _lastParams[2], _pyramid);

// update the vertex buffer with the new input
D3D11_MAPPED_SUBRESOURCE mappedDataVertex;
HR(md3dImmediateContext->Map(_vertexBuffer, 0, D3D11_MAP_WRITE_DISCARD, 0, &mappedDataVertex));

Vertex* v = reinterpret_cast<Vertex*>(mappedDataVertex.pData);

XMFLOAT4 black(0.0f, 0.0f, 0.0f, 1.0f);

for (UINT i = 0; i < _pyramid.Vertices.size(); ++i)
{
        v[i].Position = _pyramid.Vertices[i].Position;
        v[i].Color = black;
}

v[0].Color = XMFLOAT4(0, 1, 0, 1.0f);

md3dImmediateContext->Unmap(_vertexBuffer, 0);

// update the index buffers
D3D11_MAPPED_SUBRESOURCE mappedDataIndex;
HR(md3dImmediateContext->Map(_indexBuffer, 0, D3D11_MAP_WRITE_DISCARD, 0, &mappedDataIndex));

UINT* in = reinterpret_cast<UINT*>(mappedDataIndex.pData);

for (UINT i = 0; i < _pyramid.Indices.size(); ++i)
{
        in[i] = _pyramid.Indices[i];
}

md3dImmediateContext->Unmap(_indexBuffer, 0);

_vertexCount = _pyramid.Vertices.size();
_indexCount = _pyramid.Indices.size();
```

Code snippet 02 – Changing the contents of the dynamic buffers.

2.3 RENDERING ON GPU

The advanced requirement states that the primitive should be drawn on the GPU with a geometry shader. To achieve this a few changes had to be made and multiple geometry shaders had to be programmed. This section will first list the made changes, followed by an explanation about the geometry shaders.

The most important change made is the new **PyramidVertex** structure, see code snippet 03. This structure contains the information needed to draw the pyramid on the GPU, namely a position, color and size. The size contains both the radius and the height. Note that the amount of sides is not included in the vertex. This will be explained when I address the geometry shaders and problems that occurred while developing them. The **PyramidVertex** structure needs a new input layout and when creating the vertex buffer the byte width should be set to the size of this structure.

```
struct PyramidVertex
{
        XMFLOAT3 Position;              // 3D vector containing a position
        XMFLOAT4 Color;                 // 4D vector containing a color
        XMFLOAT2 Size;                  // 4D vector containing a color
};
```

Code snippet 03 – The new vertex structure for the pyramid.

Since this new structure contains all information for the pyramid and the geometry of the pyramid should be calculated on the GPU, only one vertex needs to be send to the GPU in the vertex buffer. This means that there's also no need for an index buffer. The vertex buffer still needs to be dynamic to enable adjusting the pyramid on runtime. Finally, the PyramidGenerator class is not needed anymore, since this logic will be moved to the geometry shaders.

The program already used a very basic effect file containing a vertex and pixel shader to enable drawing. This effect file is used from the BoxDemo application from Luna's book and has been extended incrementally to create the needed effect file. The result has multiple techniques and geometry shaders to enable drawing pyramids with a varying amount of sides.
For starters the structures for the input and output of the vertex shader, since this shader is executed before the geometry shader is executed and therefore will create the input for the geometry shader. The geometry shader expects a primitive as input; therefor the output from the vertex shader is defined as a point in the signature of the geometry shader. Secondly a structure for the output of the geometry shader had to be created. Note that this shader doesn't need its own input structure, since it is exactly the same as the output of the vertex shader. The structure for the output for the geometry shader contains homogenous and world coordinates. These coordinates used to be part of the output of the vertex shader and enable rendering the image on your screen. Now that a geometry shader is used, the vertex shader just passes the local coordinates and the geometry shader takes care of the translation to homogenous and world coordinates. Furthermore, output of the geometry shader contains a color to pass on to the pixel shader. This output structure is returned from the geometry shader as a triangle stream, so every three following indices will form a triangle.
Next, I programmed the first geometry shader that can generate a three sided pyramid. Since HLSL, the shader language, doesn't support lists, vectors, pointers or any other way to have some kind of dynamic list, arrays were used to store the calculated vertices and the output structure. In much the same way as the **PyramidGenerator** used to work, the geometry shader calculates the vertices for the pyramid:

- Declare an array with five 4D elements, to store the world and homogenous position (three vertices for the sides, one for the apex, one for the base of the center);
- Add the apex at position (0, height, 0, 1) to the array. The height variable is obtained from the input, which is the output structure of the vertex shader;
- Add every side vertex by calling the **GetSideVertex** function, which returns a position. This function needs a size, containing height and radius, the amount of sides, the index of the side (starting at 1) and a reference to PI as parameters. It calculates the position of the vertex in the same way the **PyramidGenerator** did, see section 2.2;
- Add the center of the base at position (0, 0, 0, 1).

Some code now has to be moved from the vertex shader to the geometry shader, namely the code to calculate the homogenous and world coordinates and to pass on the color variable. Lastly, the triangle stream has to be filled with the vertices in the right order to form the triangles. Since a triangle stream interprets the vertices as a triangle strip, for every side of the pyramid the side and connected bottom can be added to the triangle strip and will be drawn as a strip. After these are added the strip is restarted and the next side plus bottom is added. The resulting output is still the input for the pixel shader. See code snippet 04 for the implementation of these steps.

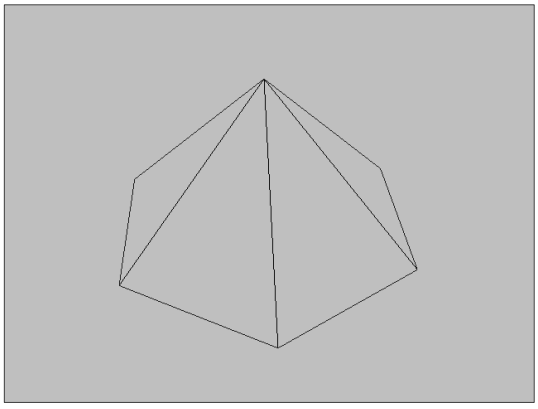
```
[maxvertexcount(12)]
void GSThreeSides(point VertexOut gin[1], inout TriangleStream<GeoOut> triStream)
{
    float PI = 3.14;

    // create a new array for the vertices with that
    // may not be declared in for loop since I get warnings if I do..
    float4 v[5];
    v[0] = float4(0.0f, gin[0].Size.x, 0.0f, 1.0f);
    v[1] = GetSideVertex(gin[0].Size, 3, 1, PI);
    v[2] = GetSideVertex(gin[0].Size, 3, 2, PI);
    v[3] = GetSideVertex(gin[0].Size, 3, 3, PI);
    v[4] = float4(+0.0f, 0.0f, +0.0f, 1.0f);

    float4 color = float4(0.0f, 0.0f, 0.0f, 1.0f);
    GeoOut gout[5];
```

```
    [unroll]
    for (int i = 0; i < 5; ++i)
    {
        // define side of pyramid
        gout[i].PosH = mul(v[i], gWorldViewProj);
        gout[i].PosW = v[i].xyz;
        gout[i].Color = color;
    }

    triStream.Append(gout[0]);
    triStream.Append(gout[2]);
    triStream.Append(gout[1]);
    triStream.Append(gout[4]);
    triStream.RestartStrip();

    triStream.Append(gout[0]);
    triStream.Append(gout[1]);
    triStream.Append(gout[3]);
    triStream.Append(gout[4]);
    triStream.RestartStrip();

    triStream.Append(gout[0]);
    triStream.Append(gout[3]);
    triStream.Append(gout[2]);
    triStream.Append(gout[4]);
    triStream.RestartStrip();
}
```

Code snippet 04 – Geometry shader for three sided pyramid generation.

To enable dynamically configuring the amount of sides of the pyramid, multiple geometry shader methods had to be made. This is caused by the fact that a geometry shader can only add a fixed amount of vertices to a primitive and the amount of sides of the pyramid influences the amount of vertices needed. Creating the other geometry shaders was a matter of copying the existing and discussed geometry shader and changing the size of the array to store the vertices and adding the additional vertices. Four geometry shaders were created in total: for three, four, five and six sided pyramids. More shaders could be created, but I would like to show that I am capable of switching effect techniques on runtime, not that I can copy paste some code over and over again. Therefore, the new maximum amount of sides is six.

Multiple effect techniques are instantiated in the **BuildFX** function, one for each described shader. These techniques are defined in the effect file and all use the same vertex and pixel shader but a different geometry shader. Based on the variable that holds the amount of sides and that can (still) be adjusted by the user, a switch case checks for the effect technique that should be used in the **DrawScene** function. The determined technique will be used for rendering the pyramid dynamically on the GPU.



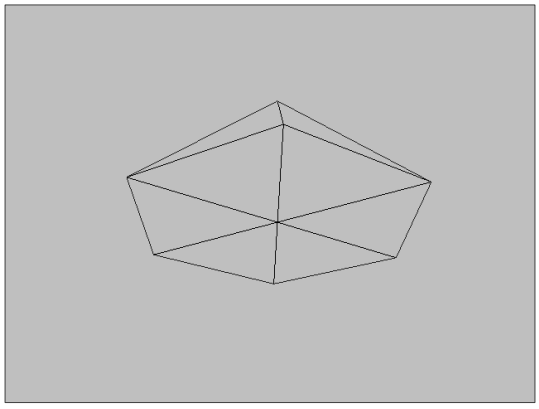


Image 01 and 02 – Pyrimad top and bottom view.

3. TEXTURES

Textures are arrays of data, often representing an image. This way, we can map image data onto a mesh to make it look more realistic. But textures can be used to store data for more advanced techniques, such as normal mapping.

For this assignment, my own smartphone had to be created virtually and rendered. The requirements are as following:

- Basic: create a mesh that resembles your phone. Then take pictures of all sides to create a single texture image to be applied with the unwrapping principle. Render the results;
- Intermediate: take a picture with your physical phone and store it in an additional texture file. Apply this additional texture on the screen space of the virtual phone without adding new geometry.
- Advanced: instead of taking a picture with your physical phone, render an image on the screen of your virtual phone as if taken from the virtual phone. Note that you need objects in your scene to actually prove that it works.

Again, incrementally was worked on this assignment. The following sections discuss how the project progressed and what choices had to be made.

3.1 MODEL AND TEXTURE MY PHONE

For starters I needed the mesh and texture of my phone, so that I could load and render them in DirectX 11. Since simple primitives (like the pyramid from last chapter) can be easily defined by hand, but meshes are often made by 3D artists, I used 3DS Max to create my mesh and uv mapping for the texture and Photoshop to create the texture itself. In Photoshop, the different pictures I took of my phone had to be rescaled and put side to side to enable unwrapping. Since I am but a novice 3D modeler, Jasper Meier assisted me in creating these assets.

Next, I set up a basic application just like in section 2.1. In this application, the mesh and texture had to be loaded and rendered. To achieve this, the **Vertex** structure had to be expanded: it now also needed a normal, as 3D vector, and a texture coordinate, as 2D texture, to enable mapping the texture onto the vertices of the mesh. I struggled with loading my model into the application, since DirectX can't load files like .obj or .fbx. I found a converter that takes an .obj file and converts it to a simple .txt file (http://www.rastertek.com/dx11tut08.html). It also contains a method to read the input from the file and store it in lists containing the vertices and indices. I placed this code in the **BuildGeometryBuffers** method to create the immutable vertex and index buffers.

To make use of a texture, I used the effect file from Luna's Crate demo, in which he textures a box. This effect file contains a Texture2D variable called **gDiffuseMap**. I created a shader resource view of my texture and set it to be the value for **gDiffuseMap** every frame before the phone is drawn. In the existing effect file, this variable is used in the pixel shader to sample the correct color for the given pixel. These two steps – loading the model and binding the texture – enables drawing my phone on the screen.

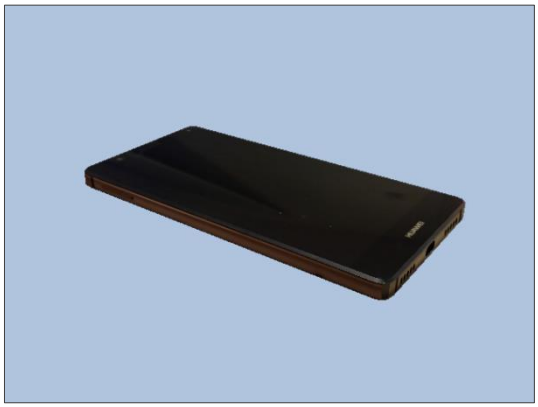


Image 03 – Textures basics

3.2 SCREEN ON

To complete the intermediate assignment, I took a picture with my physical phone, stored it in a .png file (just like I did with the phone texture) and used it as a second texture to draw a screen on my phone.

I thought a long time about how to complete this assignment. I decided that I had to define a second texture coordinate for each vertex and set a second texture to the effect. But since I had a relatively complex model, I couldn't define a second texture coordinate easily by hand: it was hard to image which vertex was which and how offsets should be considered. After the consultant of my teacher, I decided to lose the complex model and define my phone as a simple rectangular box.

Image 04 – Texture intermediate

To enable defining a second texture coordinate, I expanded the vertex structure to accept a second texture coordinate as a 2D vector. The vertices for the phone are defined by hand in an array structure just like Luna did in his Box demo. I used 3DS Max to help me figure out and visualize the texture coordinates for both the phone and the screen texture. Indices are also defined by hand. All this code is placed in **BuildGeometryBuffers** instead of reading the contents of the .txt file. Afterwards the buffers are created with the given vertex and index data.
The second texture coordinate didn't do anything extra up to this point: now I had to actually do something with it. First, I defined both the first and second shader resource view, by using Luna's method **CreateTexture2DArraySRV**. This method takes a vector of filenames and results in a shader resource view pointer to access the different textures. Second, I altered the effect file, used in section 3.1:

- It now has a Texture2DArray variable instead of a Texture2D, so I can send multiple textures;
- A new SamplerState has been defined with border mode and a transparent border color. This state will be used when sampling the second texture;
- The input and output structures for the vertex shader are expanded with a second texture coordinate;
- The pixel shader first samples from the first texture and then from the second texture with the new SamplerState. Afterwards the second color is multiplied (to make the screen look brighter) and added to the first. See code snippet 05 for the code that samples from the textures and blends the colors.

Instead of setting the value of the 2D texture every frame, now the value of the 2D texture array is set before the phone is drawn.

```
float3 uvw = float3(pin.Tex, 0);
texColor = gMapArray.Sample(samAnisotropic, uvw);
uvw = float3(pin.Tex02, 1);
texColor02 = gMapArray.Sample(samAnisotropicWBorderColor, uvw);
texColor = texColor + (texColor02 * 1.9f);
```

Code snippet 05 – Sampling and blending the textures.

3.3 DYNAMIC SCREEN

To finish the textures assignment, the advanced requirements state that the screen of the phone has to be dynamic, as is you are taking a picture with the virtual phone. To achieve this, the following steps had to be taken:

- Objects needed to be placed in the scene, so that the user can actually see the screen texture change;
- Instead of only having a shader resource view, I also need a texture2D to render to and a second render target view;
- A method **BuildOffscreenViews** has been defined. This method creates the descriptions for and initializes the variables named in the last bullet point;
- The two textures can't be loaded on initialization: since the screen texture will be drawn on runtime, only the first texture is defined as in section 3.1;
- Instead of expecting a Texture2DArray, the effect file expects two Texture2D variables;
- Drawing the scene is split up in multiple methods. This will be explained later in this section.

Defining the vertex and index buffers didn't change from last section's project, just as the pixel shader and input and output structures for the vertex shader.
The remaining of this section will discuss the applied changes. The results of these changes is that the at first static texture representing the screen, will now show the scene, as if the phone runs the camera application.

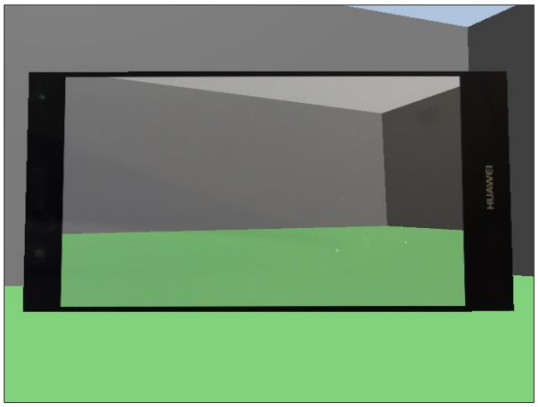


Image 05 – Textures advanced

Firstly, the scene needed to be filled with a few simple objects, so that the effects of the dynamic texture are actually visible. I used Luna's **GeometryGenerator** class to create a box, representing a wall, a grid, representing the floor. I calculated the correct offsets in the buffers and stored the vertices and indices of these objects alongside the vertices and indices of the phone. These offsets are used to draw the correct indices for each object in the **DrawScene** method.

With translation and rotation matrices I positioned four walls and a floor. With two simple materials I gave them some color to make the scene more interesting.

A few extra variables are needed to create the screen texture. The idea is as following: render the complete scene just like you would normally do, but don't render the phone and don't render it to the back buffer, but to a different buffer. Then, this buffer can be used to create a texture of the rendered image. This texture is used as screen texture.
To enable this, I defined a shader resource view, a 2D texture and a render target view. Instead of rendering to the back buffer, I will render to the new render target view. The shader resource view is used to set the second texture and to sample from in the effect file. The texture is used to create both the shader resource view and the render target.
In the method **BuildOffscreenViews** all these variables are created and coupled. The 2D texture is created based on the screen size and is bound to the pipeline as a render target. This texture is used in the creation of both other variables, as mentioned. This method is mostly based on Luna's Blur demo, in which he renders the complete scene to a texture, blurs the texture and draws it on a quad. A few changes had to be made, since Luna uses the compute shader to blend his texture and I have no use for the compute shader in this project.

The method used to load multiple textures in the last section, couldn't be used for this assignment, since only one texture is actually stored in a file; the other is created in the **BuildOffscreenViews** method. I decided to load only the first texture in the same way as I loaded it in section 3.1.
This had as consequence that I couldn't send both textures in an array to the GPU. I decided to define two Texture2D variables in the effect file and set them individually. This doesn't change much in sampling from the textures in the pixel shader: instead of referring to the first or second element in the texture array, just refer to the correct texture variable.

Finally the draw method had to be split up in three methods. See code snippet 06 for the DrawScene method:

- **DrawScene**, called every frame as usual. This method now:
  - Starts by setting the render target for the texture and clearing it;
  - Second it calls the **DrawStart** method;
  - Third it sets the default render target to render to the back buffer and clears it;
  - Then it calls the **DrawFinish** method;
  - Finally swap the back and front buffer.
- **DrawStart**, draws all objects except the phone.
- **DrawFinish**, draws all objects including the phone.

So the scene is drawn two times, once to the screen texture and once to the back buffer as usual.

```
void TexturesApp::DrawScene()
{
   // set the SRV for the screen so that I can render to it
   ID3D11RenderTargetView* renderTargets[1] = { _offscreenRTV };
   md3dImmediateContext->OMSetRenderTargets(1, renderTargets, mDepthStencilView);

   // clear views
   md3dImmediateContext->ClearRenderTargetView(_offscreenRTV, reinterpret_cast<const
     float*>(&Colors::Silver));
   md3dImmediateContext->ClearDepthStencilView(mDepthStencilView, D3D11_CLEAR_DEPTH |
     D3D11_CLEAR_STENCIL, 1.0f, 0);

   // draw everything in the scene except from the phone
   DrawStart();

   // set SRV to the default render target, the back buffer
   renderTargets[0] = mRenderTargetView;
   md3dImmediateContext->OMSetRenderTargets(1, renderTargets, mDepthStencilView);

   // clear views
   md3dImmediateContext->ClearRenderTargetView(mRenderTargetView, reinterpret_cast<const
     float*>(&Colors::LightSteelBlue));
   md3dImmediateContext->ClearDepthStencilView(mDepthStencilView, D3D11_CLEAR_DEPTH |
     D3D11_CLEAR_STENCIL, 1.0f, 0);
```

```
    // draw everything, including phone with screen texture
    DrawFinish();

    HR(mSwapChain->Present(0, 0));
}
```

Code snippet 06 – Drawing the scene to a texture before drawing it to the screen.

The phone rotates with the camera, so that it looks like the user holds the phone in front of him and can rotate around with a camera application on.

4. LIGHTING

The assignments regarding lighting explored the different kinds of lights and for advanced a sophisticated lighting technique. The requirements are as follows:

- Basic: implement a magic wand with a point light;
- Intermediate: implement a red laser beam with a variable cone;
- Advanced: implement a light effect hovering over water.

Since the different requirements don't follow each other up directly, I started with a new project for each requirement and used the Lighting.fx effect file from Luna. This effect file has a constant buffer for the three different types of lighting: directional, point and spot light. The constant buffer will be set every frame. In the pixel shader of this file, the color of the pixel is calculated based on the normal and the light.

4.1 MAGIC WAND

For the basic assignment, I implemented a wand with a point light at the end of the wand. The point light can be turned on and off with the right mouse button. With the left mouse button the user can make the wand follow the mouse. Since light can only be viewed when it falls on an object, I also placed a wall and floor in the scene.

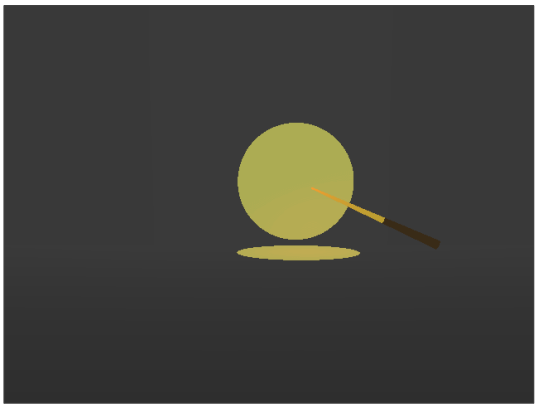


Image 06 – Lighting basics

First I created the objects in the scene in the **BuildGeometry** method in much the same way as before. I needed a cylinder for the wand, a box for the wall and a grid for the floor. These objects are positioned with translation and rotation matrices so that the light from the wand can fall on the wall and floor and so that the wand itself is rotated on the z axis. For each object a material was created with different tints of brown.

Second I create the lights. A directional light makes sure that the scene isn't pitch-black without other light sources. A yellow point light is positioned at one end of the wand as if it emerges from the wand. See code snippet 07 for the initialization of the point light. By passing both lights to the GPU in the draw method they will be processed by the effect file code.

Finally I wanted to be able to move the wand and make the point light follow it. Also, I wanted to turn the point light on and off on commando. To achieve this, I did the following:

- If the user presses the right mouse button, I set the range of the point light to 0 is the light is on. If the light is off, I set the range back to its starting value of 2;
- If the user presses the left mouse button and moves the wand, I calculate the difference in mouse position and clamp it so the wand can't move out of the screen. This difference is used to calculate the new point light position vector and the new matrix for the wand.
- Each frame the position of the point light is set and the matrix for the wand is passed to the GPU right before drawing the wand.

The result is a pretty dark scene with a bright yellow light following the wand.

```
// Point light
_lightPos = XMFLOAT3(0.0f, 3.0f, 0.0f);
_pointLight.Ambient = XMFLOAT4(0.9f, 0.9f, 0.2f, 1.0f);
```

```
_pointLight.Diffuse = XMFLOAT4(1.0f, 0.0f, 0.0f, 1.0f);
_pointLight.Specular = XMFLOAT4(0.3f, 0.3f, 0.3f, 1.0f);
_pointLight.Att = XMFLOAT3(1.0f, 1.0f, 1.0f);
_pointLight.Range = 2.0f;
_pointLight.Position = XMFLOAT3(0, 1.8f, 0);

// set the position of the point light to match the wand
XMVECTOR poslight = XMLoadFloat3(&_pointLight.Position);
XMStoreFloat3(&_pointLight.Position, XMVector3TransformCoord(poslight, XMLoadFloat4x4(&_wandWorld)));
_lightPos = _pointLight.Position;
```

Code snippet 07 – Point light

4.2 LASER BEAM

To complete the intermediate assignment, I implemented a spot light that defaults to a very small size. This is actually a large number for the spot size, since large numbers result in a small spot size and vice versa. The spot light is positioned at the same position as the camera and is directed in the same orientation as the camera as if the user points the light.

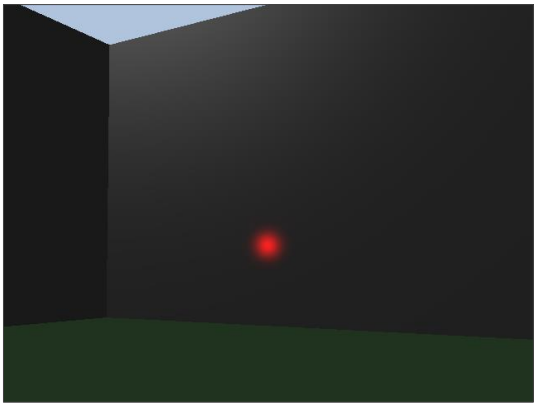


Image 07 – Lighting intermediate

Just as in the last section, I needed a few objects in the scene to enable seeing the light and a directional light. I started by defining the objects, their matrices and materials in the very same way as before. In the draw call I draw four walls and a floor. I wanted four walls so the user could look 360 degrees around him and see the effects of the spot light. The directional light is also defined as in the last section.

The spot light is set to red ambient and diffuse properties so it will always have a strong red color, no matter the color of the material it reflects upon. Both spot and range variables are initialized with large values, so the light will have a small spot and be intense, even at a distance. The size of the spot is captured in a float variable **spotSize**, so it can be adjusted based on user input. For the initialization of the spot light, see code snippet 08.

To adjust the spot size, I capture the key input each frame. If the user presses the left or right key, the value of **spotSize** decreases or increases respectively. The value is clamped between a minimum and maximum. Each frame the spot size of the spot light is set to be equal to **spotSize**.

The result is, again, a pretty dark scene with a bright red light following the camera orientation. The spot size can be adjusted by the user.

```
_spotLight.Ambient  = XMFLOAT4(1.0f, 0.0f, 0.0f, 1.0f);
_spotLight.Diffuse  = XMFLOAT4(1.0f, 0.0f, 0.0f, 1.0f);
_spotLight.Specular = XMFLOAT4(0.4f, 0.4f, 0.4f, 5.0f);
_spotLight.Att      = XMFLOAT3(1.0f, 0.0f, 0.0f);
_spotLight.Spot     = 5000.0f;
_spotLight.Range    = 10000.0f;
```

Code snippet 08 – Spot light

4.3 UNDERWATER LIGHT

The final assignment for lighting required to create a light effect hovering over water. I created an underwater scene for this assignment with a point light that projects a light map. It creates an effect that you see when light shines through the water on the bottom of a water body. This way it seems like the light shines through the water on the ground, even though there is no actual water.

I started setting up a dark scene, to give an underwater feeling. I used the following parts in the scene:

- Three very soft directional lights, to light the scene up just a little;
- A big grid with a tiled sand texture;
- Dark blue-green fog effect;
- Dark blue-green background.

In this resulting scene I placed a point light, which will be used to project the light map. The light map enables me to only project a light partly. It is a black-white colored texture. A pure white pixel indicates that all the light

will shine through and a black pixel indicates that no light will shine through. To project a light map I wrote additional code for the pixel shader Luna wrote for handling light, fog and textures and wrote the code that is needed to provide the pixel shader with the correct variables.

- A second texture for the light map is loaded and is send to the GPU. This is a black-white texture to calculate the amount of light that shines through;
- View and projection matrixes are built from the point light position and project on the ground. These matrixes are used to project the texture on the floor in the pixel shader and therefore send to the GPU.

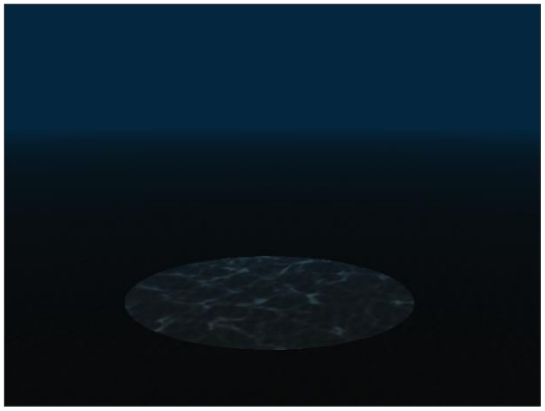


Image 08 – Lighting advanced

With these changes, the shader code has all needed information to project the texture. See code snippet 09 for the complete pixel shader. The code added to the existing pixel shader takes the following steps:

- Calculate, in the vertex shader, the homogenous coordinates from the view and projection matrix of the point light, **ViewPosW**, and the distance vector between the light and this vertex, **lightPos**;
- All further code is placed in the pixel shader. Calculate the magnitude of the **lightPos** vector. If this magnitude is longer than the range of the point light, set the float **lightIntensity** to 0. Else, calculate the dot product of the normal of this pixel and **lightPos** as light intensity;
- If **lightIntensity** is greater than 0, calculate the value of the float4 **color**, the color for the light based on the point light, **lightIntensity** and the brightness, which can be manually adjusted;
- Then **texColor**, the color based on the texture, and **litColor**, which is the color of **texColor** after light calculations, are calculated as usual in Luna's part of the shader. Finally the effect of the fog is calculated over the **litColor**;
- Next, I calculate two texture coordinates on the same water texture based on **ViewPosW**. Both texture coordinates give a different offset, so that they don't overlap. These texture will be used later on in this section to animate the water;
- The calculated texture coordinates need to be wrapped around 0 and 1, so that the texture loops around. The chosen texture is fit for tiling;
- Sample two texture colors from the same water texture, **projectionColor** and **projectionColor2**;
- Calculate the final projection color by multiplying **color** with **projectionColor**, **projectionColor2** and **litColor**. Calculate a light color, based on **litColor** and the point light, to add to the final projection color, so that some parts don't get pitch black and it looks more like a shadow effect;
- Return the resulting color.

The result was a dark scene with the pattern of water projected on the ground. I was quite happy with the results, but wanted to finish it off by animating the light. I achieved this by passing a float to the GPU, which is increased slightly every frame. I use this float as an offset to sample from the water texture. Since I use two texture coordinates and offset them both on a different axis, the water effect seems more natural. The code to make sure that the texture coordinates can't get out of bounds was already in place. Concluding, the scene is dark and seems to be underwater, a light seems to shine on the ground from above water and it projects a texture on the ground as if the water flows slowly.

```
float4 PS(VertexOut pin, uniform int gLightCount, uniform bool gUseTexure) : SV_Target
{
    float brightness = 4.0f;
    float lightIntensity,
    float4 color;
    float4 texColor = float4(1, 1, 1, 1);
    float4 projectionColor;
    float4 projectionColor2;
    float2 projectTexCoord;
    float2 projectTexCoord2;

    pin.NormalW = normalize(pin.NormalW);

    float3 toEye = gEyePosW - pin.PosW;

    float distToEye = length(toEye);
    toEye /= distToEye;
```

```
    if ((pin.LightPos.x * pin.LightPos.x + pin.LightPos.z * pin.LightPos.z + pin.LightPos.y * pin.LightPos.y) >
(gPointLight.Range * gPointLight.Range))
        lightIntensity = 0.0f;
    else
        lightIntensity = saturate(dot(pin.NormalW, pin.LightPos));

    if (lightIntensity > 0.0f)
        color = (gPointLight.Diffuse * gPointLight.Ambient * lightIntensity) * brightness;

    /* ... Code for calculating texture, light and fog are left out ... */

    if (lightIntensity > 0.0f)
    {
        projectTexCoord.x = pin.ViewPosW.x / pin.ViewPosW.w / 2.0f + 0.5f + offset;
        projectTexCoord.y = -pin.ViewPosW.y / pin.ViewPosW.w / 2.0f + 0.5f;

        projectTexCoord2.x = pin.ViewPosW.x / pin.ViewPosW.w / 2.0f + 0.5f;
        projectTexCoord2.y = projectTexCoord.y - offset + 0.03;

        if (projectTexCoord.x < 0)
        {
            projectTexCoord.x = 1 + projectTexCoord.x;
        }
        else if (projectTexCoord.x > 1.0f)
        {
            projectTexCoord.x = projectTexCoord.x - 1;
        }

        if (projectTexCoord2.y < 0)
        {
            projectTexCoord2.y = 1 + projectTexCoord2.y;
        }
        else if (projectTexCoord2.y > 1.0f)
        {
            projectTexCoord2.y = projectTexCoord2.y - 1;
        }

        projectionColor = gProjectionTexture.Sample(samAnisotropic, projectTexCoord);
        projectionColor2 = gProjectionTexture.Sample(samAnisotropic, projectTexCoord2);

        litColor = saturate((color * projectionColor * projectionColor2 * litColor) + (gPointLight.Ambient *
                            litColor));
    }
    else
    {
        litColor = gPointLight.Ambient * litColor;
    }

    litColor.a = gMaterial.Diffuse.a * texColor.a;

    return litColor;
}
```

Code snippet 09 – Pixel shader light map projection

5. SHADING

For the shading assignments I had my first in-depth experience with HLSL, high level shading language. All effect files mentioned earlier are written in this programming language, so I already knew the basic syntax. For these assignments I learned programming these effect files myself. The requirements are as following:

- Basic: adjust the shader 'reflective chrome' made by Luna, so that the amount of reflectiveness can be changed by the user on runtime;
- Intermediate: implement an existing pixel shader from www.shadertoy.com;
- Advanced: implement a custom Fresnel shader.

Since requirements do not follow each other up, I started with a new project for each requirement.

5.1 REFLECTIVE CHROME BY LUNA

The reflective chrome material by Luna is implemented in his Dynamic cube mapping demo, in which he dynamically creates a cube mapping from the sky. The sphere in the middle of the scene uses the reflective chrome material and reflects the dynamic cube map. A normalized 4D vector, stored in the **Material** structure determines the amount of reflectiveness. The elements in this vector will be multiplied by a color value and can therefore be interpreted as R, G, B, A values. The pixel shader from Luna's effect file takes the following steps to apply the reflection:

- Take the opposite direction of the direction from this pixel to the eye position, call this incident;

- Reflect the incident through the normal to get the reflection vector;
- Sample the color to reflect by using the reflection vector as texture coordinate;
- Multiply the reflection color by the 4D reflectiveness vector from the material and add it to the color this pixel had to begin with.

By simply changing the R, G and B values in the 4D reflectiveness vector of the material of the sphere in Luna's demo, the amount of reflectiveness can be changed on runtime.

To achieve this, I started out with Luna's Dynamic cube mapping demo and made the following changes:

- Define floats to store the current reflectiveness and minimum and maximum values.
- Set the R, G and B values of the **reflect** variable in the **Material** variable **mCenterSphere** to the default reflectiveness value;
- Catch the input for left and right arrow keys each frame. If the left arrow key is pressed: decrease the reflectiveness value by 0.001. If the right arrow key is pressed: increase it by the same value.
- Clamp the reflectiveness value each frame between 0 and 1.
- Set R, G and B values of the **reflect** variable of **mCenterSphere** to the current reflectiveness value.

The result is the same scene as Luna created for the Dynamic cube mapping demo, but when the user presses the left or right arrow key, the amount of reflection of the center sphere will respectively decrease or increase.

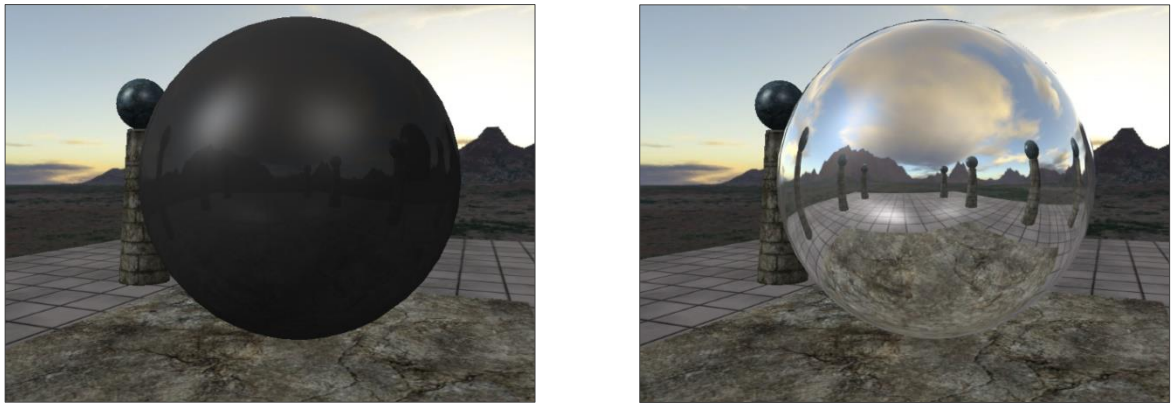


Image 09 and 10 – Shaders basic

5.2 COLORFULL NOISE

For the intermediate assignments I explored the beautiful shaders at www.shadertoy.com. I personally like abstract shapes or colors and I found a few shaders using noise in different ways, which I liked very much. I chose to implement a shader called **isovalues 2**, created with noise and contour lines (https://www.shadertoy.com/view/MdfcRS). There are different versions of this shader, with thick and thin lines or with a depth perception. I started a new project with the **Lighting.fx** file from Luna and took the following steps to set up my scene, after which I was ready to implement the shader:

- Create a sphere with Luna's **GeometryGenerator** and use its information to create the vertex and index buffers;
- Define 125 matrices for 125 spheres that translate in such a way that the spheres form a cube structure;
- Set a directional light and simple grey material to light the spheres up;
- Draw 125 spheres in the **DrawScene** method.

To implement the chosen shader, I started to read the code. When I got the basic idea I copied all the code from the **mainImage** method from www.shadertoy.com to my own method **PSNoise**, which will be used as pixel shader in a newly defined technique. Other methods defined by the developer of this shader were also copied, **noise**, **noise3** and **hash3**. To give a broad overview of the implementation of the shader in GLSL:

- The pixel shader takes a 2D coordinate and outputs a 4D color vector. It calls **noise** with the x and y coordinates of the pixel and the current game time and applies a few other functions based on the result of **noise** to calculate the final color;
- The **noise** method takes a 3D vector and applies a pseudo Perlin noise by calling the **noise3** method;

- The **noise3** method takes a 3D vector applies a function so that the borders are continuous and then applies a trilinear interpolation based on the given 3D vector and a **hash** function;
- The **hash3** method takes a 3D vector and returns a semi random number.

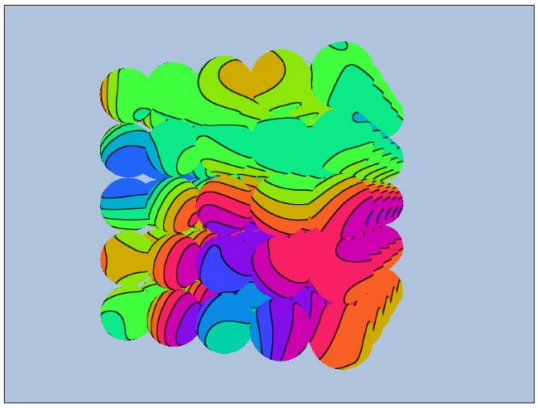


Image 11 – Shaders intermediate

To get the code to work, I mainly had to convert the GLSL code from www.shadertoy.com to HLSL code and make a few other alternations. See code snippet 10 for the resulting pixel shader:

- Every 'vector' in the GLSL code needs to be replaced by 'float' followed by the number;
- The **mix** method from GLSL is the **lerp** method in HLSL;
- The input parameters change from only 2D position to a 3D position for world coordinates, a 4D position for homogenous coordinates and normal vector;
- The output parameters didn't change since a pixel shader should only return a color;
- The shader needed a few variables from the CPU, namely the current resolution and the game time. I defined these two in the constant buffer that will be updated per frame, obtained the variables by name in my cpp file and set them every frame in the **DrawScene** method;
- Create a new technique called **NoiceTech** with the new pixel shader and get that technique in my cpp file.

These simple changes enabled drawing the spheres with the noise shader applied to them. Since the noise shader is 2D, the effect is less amazing than I hoped for: only the front and the back of the cube of spheres show the texture nicely, all other sides show a repeated pattern. The pattern is repeated since only the x and y positions of the pixel are taken into account when calculating the correct color. I tried to take also the z position into account, but I wasn't able to do it. Every adjustment I tried ended up in destroying the contour lines, disabling the smooth movement over time caused by passing the game time or creating a noise shader that created more and more contour lines if time passed. In the end I decided that it may not look as I wanted it to look, but I managed to successfully implement a pixel shader from www.shadertoy.com, as the assignment required.

```
float hash3(float3 n)
{
    return frac(sin(1000 * dot(n, float3(1, 57, -13.7))) * 4375.5453);
}

float noise3(float3 x)
{
    float3 p = floor(x);
    float3 f = frac(x);

    // make continuous smooth borders
    f = f * f * (3.0f - 2.0f * f);

   // trilinear interpolation
    return lerp(
        lerp(
            lerp(hash3(p + float3(0, 0, 0)), hash3(p + float3(1, 0, 0)), f.x),
            lerp(hash3(p + float3(0, 1, 0)), hash3(p + float3(1, 1, 0)), f.x), f.y
        ),
        lerp(
            lerp(hash3(p + float3(0, 0, 1)), hash3(p + float3(1, 0, 1)), f.x),
            lerp(hash3(p + float3(0, 1, 1)), hash3(p + float3(1, 1, 1)), f.x), f.y), f.z
        );
}

float noise(float3 x)
{
    return (noise3(x) + noise3(x + 11.5f)) / 2.0f;
}

float4 PSNoise(VertexOut pin) : SV_Target
{
    pin.PosW *= 8.0f / gResolution.y;
```

```
    float n = noise(float3(pin.PosW.xy, 0.1f * gGlobalTime));
    float v = sin(6.28f * 10.0f * n);
    v = smoothstep(0.0f, 1.0f, 0.7f * abs(v) / fwidth(v));
    n = floor(n * 20.0f) / 20.0f;

    return v * (0.5f + 0.5f * cos(12.0f * n + float4(0.0f, 2.1f, -2.1f, 0.0f)));
}
```

Code snippet 10 – Perlin noise pixel shader

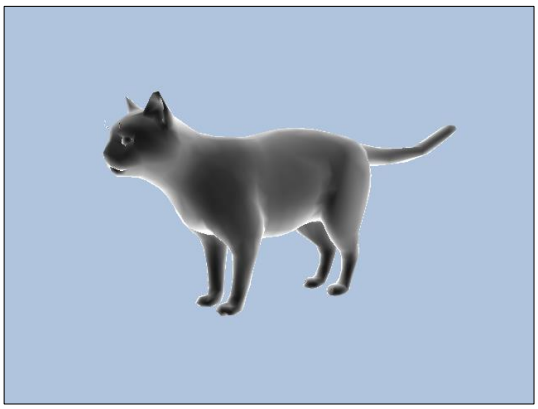
5.3 FRESNEL SHADER

The last shader assignment states I have to implement a custom Fresnel shader. A Fresnel shader is a pixel shader that calculates the color of a pixel based on the angle between the normal of the pixel and the camera's direction. To get started with this assignment I took the result from section 5.2. By taking that project as starting point, I only had to change the pixel shader to see the effects from my new Fresnel shader. Before I began programming, I worked out the steps the algorithm has to take:

1. Normalize the normal world vector of the pixel and the vector indicating the position of the camera;
2. Define the maximum and minimum color, e.g. black and white;
3. Calculate the dot product between the two normalized vectors and subtract the result from 1, so you get the inverse of the dot product;
4. Optionally do some other calculations on the result of the dot product, like to the power of some floating point number or adjusting by some value calculated over the minimum and maximum colors, to tweak the resulting color;
5. Interpolate between the minimum and maximum color with the result from the last step;
6. Set the alpha channel of the color to 1 and return.

The implementation of the algorithm went rather smoothly and without big obstacles. See code snippet 11 for the resulting pixel shader. I put most effort in tweaking the results, since the differences between some of them are huge and give interesting effects. I tried the following:

- Take the result from step 3 to some power. The higher the power the more intense the maximum color becomes and the less noticeable the minimum color becomes. I choose a value of 0.9;
- By subtracting the minimum color from the maximum and multiplying it by a small value I was able for generate one floating point number, but four. This way the interpolation can be done with four different values for R, G, B and A, which enables using a minimum and maximum color instead of only grey tints. The small value with which I multiplied the difference between the minimum and maximum color controls the intensity of the color. I choose a value of 0.09 so that the color only gives a vague gloss.



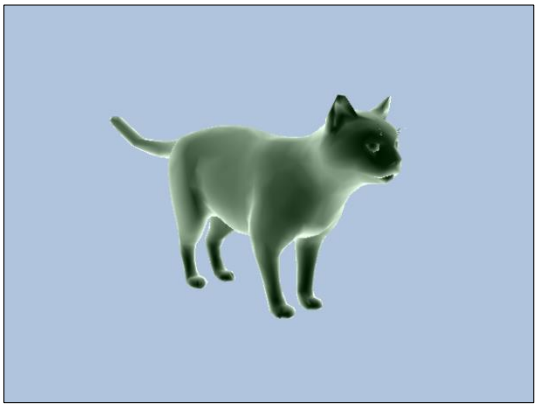


Image 12 and 13 – Shaders advanced

I really liked the effect of the last tweak, but in the original implementation I only used black and white. I decided that I wanted to change colors on runtime. I achieved this by declaring a new variable in the constant buffer per object and setting effect variable in the **DrawScene** method just before drawing the spheres. I also defined an array with all colors in the rainbow and an integer keeping track of the current color. This color is passed to the constant buffer. By using the left and right arrow keys the user can cycle through the array and change the color on runtime.

A final adjustment I made was changing the content in the scene. The cube formed by spheres looked nice, but didn't show the effect of the Fresnel shader nicely in my opinion: since all normal are ordered in a continuous fashion, the effect looks like it creates contours. I therefore looked for a more complex model with normals in different directions. I found a low poly model of a cat. In the same way as I converted and loaded my phone model in section 3.1 I loaded the cat model and stored its data in the vertex and index buffers. To finish off, the variables used to draw the spheres, e.g. matrices, buffers and their for-loop in the **DrawScene** method are discarded.
The result is a cat with the Fresnel shader applied. The user can look at the cat from all angles and adjust the color used in the shader himself. I am very happy with the result and I learned a lot from implementing my own shader.

```
float4 PS(VertexOut pin) : SV_Target
{
    pin.NormalW = normalize(pin.NormalW);
    float3 eyePos = normalize(gEyePosW);

    float4 refraction = float4(0.0f, 0.0f, 0.0f, 1.0f);
    float4 reflection = gColor;

    float4 fresnel = 0.09f * (reflection-refraction) + pow(abs(1.0f-dot(pin.NormalW, eyePos)), 0.9f);
    float4 result = lerp(refraction, reflection, fresnel);
    result.a = 1;
    return fresnel;
}
```

Code snippet 11 – Fresnel shader

CONCLUSION

I feel confident about my resulting projects. They may be very useful for my programming portfolio since they add variety to it. My favorite projects are 'Underwater light' and 'Fresnel shader'. The fun thing about programming light and shader effects is that the visual results are very appealing. The assignments on the topic primitives, for example, were also nice to make, but the visual results were far less appealing. Looking purely at the technical aspects I liked 'Dynamic screen' the best, since it gave a lot of technical challenges.

Concluding, I want to state that these assignments were very informative and I liked the overall degree of challenge. The assignments gave me a chance to get familiar with different graphic programming techniques. Since I didn't knew anything about graphics programming at the start of this semester, I really liked it that the lessons cover this variety of different topics. This overview of different topics and techniques gave me a good basis of knowledge as graphics programmer and I am glad with my new skills.