PRACTICAL 1

Creating window and initializing direct 3d.

- ➔ For rendering or drawing something on screen first thing that we would need is a window.

  >> We could do this by using the WNDCLASSEX which is struct that helps us to describe the window properties.

  >> We first assign the memory block for our struct ,then we set its properties like size, style (that's horizontal and vertical redraw part),assign some event handler (WindowProc) that is indicator to handle any event that occurs related to window, give a copy of our application or handle to r application to the window, then set cursor type, next set the brush that will be used to color the window, finally provide the class that the window describes.

  >> Next we create the window using CreateWindowEx(), which takes the class of window, the title shown in the window, the property of window containing all the necessary components required, the start position of window ,the width and height and handle to our instance.

  >> Finally use the ShowWindow(), that helps to display our window.

  >> Now one thing we need is how to handle our window events that helps us keep running our application until it is stopped by the user, so what we would do is attach a loop that runs to infinity and then we could handle our the event messages(events like mouse movement , key pressing and so on), using the peek message functionality. It helps us to read and process an of the messages which is pass through the events and then further handled by the WindowProc() which has the logic of what to do with this messages. Now this Infinite Loop does another important thing when not handling the messages, that is work on our gaming logic or in our case here rendering the frame continuously.

➔ Now the next part is where we actually set up for everything we are going to do and that is rendering.

>> You would now need some important things, COM interfaces that helps us in initializing our direct 3d environment.

>>ID3D11Device : it is pointer to device which represent a virtual adapter that will be helpful to attach thing we need in our application.

>>ID3D11DeviceContext : this is the one which actually manages the GPU and the rendering pipeline , so all tasks that changes something needed to go through device context.

>>ID3D11RenderTargetView : it's just pointer to back buffer.

>>IDXGISwapChain: it is the part of Direct X Graphic Infrastructure that helps to create the core part that is swap chain. As we could recall it is the one way the rendering process can work smoothly and one can render without flickering problems or delays that could produce ugly images or objects.

Here first things we need is to describe our swap chain, this takes total number of buffers, the usage property indicates where to draw (Render target view), the multi sampling count and some more which describes our chain. Next we need to create it and then attach it to device.

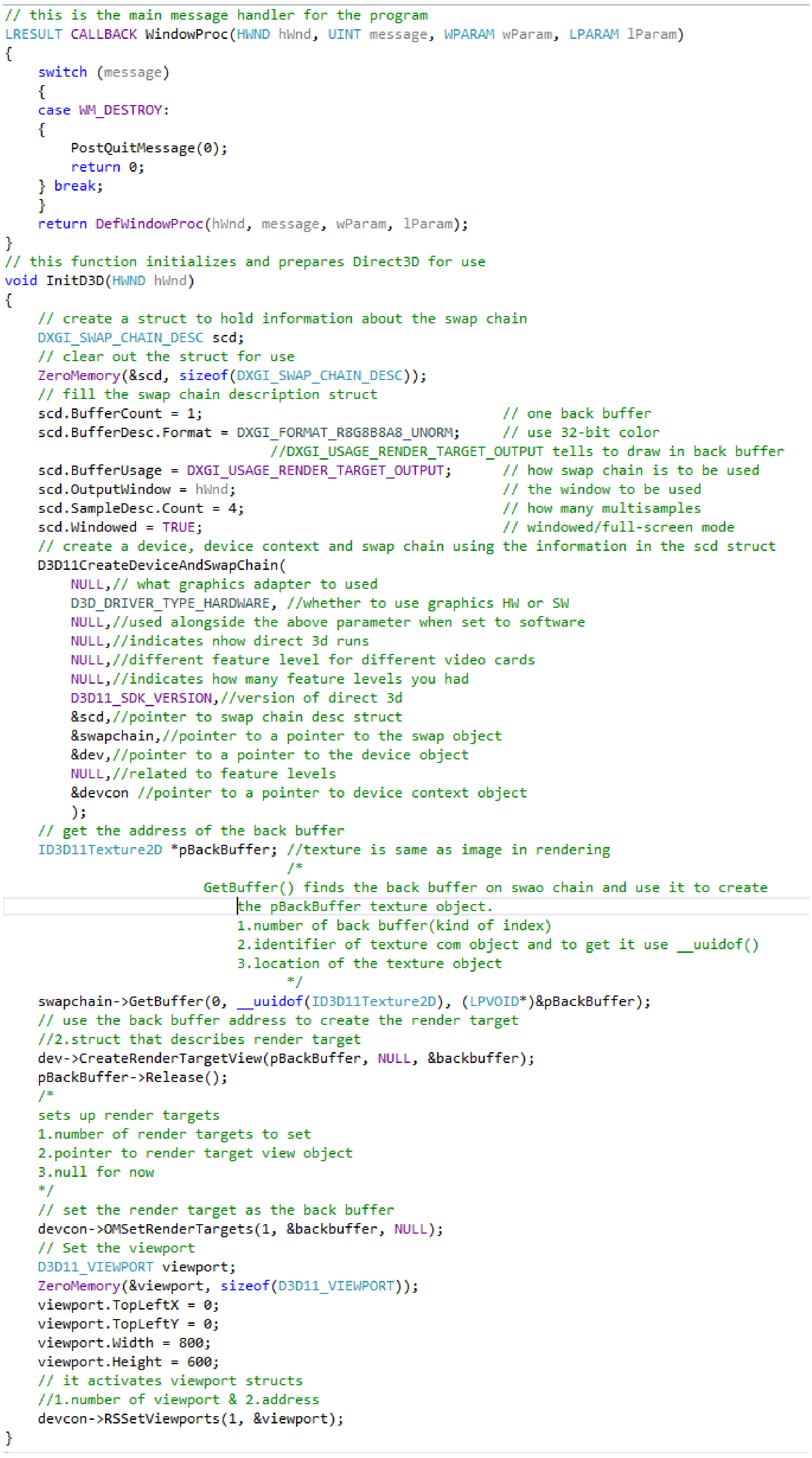
But before attaching to device we need to do one more thing that is important to swap chain that is give it an texture com object that hold our component we will draw , in other words it is the back buffer that needs to be attached to swap chain.

Finally we set output merger stage for rendering, where you provide number of render target you want to bind with(in our case that is one),then pass back buffer(Render target com object).

One other thing we need to do is set up our rasterizer stage that is the viewport and attach it to device context.

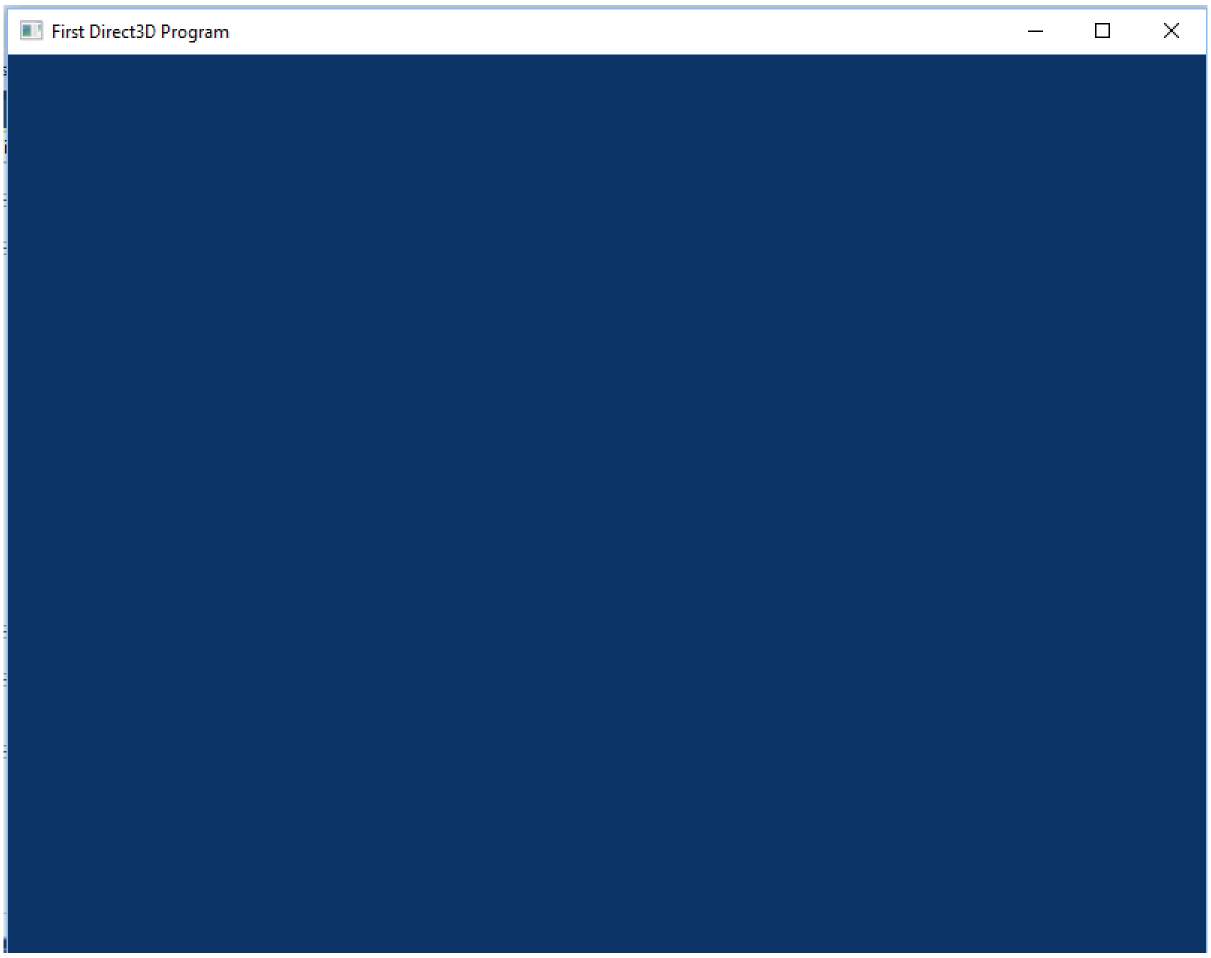
Lastly in the game logic part of our forever loop we tell it render frames continuously and after exiting make sure to realise all the com objects.

```
// include the basic windows header files and the Direct3D header files
#include <windows.h>
#include <windowsx.h>
#include <d3d11.h>
// include the Direct3D Library file
#pragma comment (lib, "d3d11.lib")
// global declarations
IDXGISwapChain *swapchain;             // the pointer to the swap chain interface
                                       /* device is an virtual representation
                                       of video adapter.
                                       */
ID3D11Device *dev;                     // the pointer to our Direct3D device interface
                                       /*
                                       device context is responsible for managing
                                       the gpu and the rendering pipeline, it used to
                                       render graphics and to determine how they will be rendered
                                       */
ID3D11DeviceContext *devcon;           // the pointer to our Direct3D device context
ID3D11RenderTargetView *backbuffer;    // the pointer to our back buffer
                                       // function prototypes
void InitD3D(HWND hWnd);    // sets up and initializes Direct3D
void CleanD3D(void);        // closes Direct3D and releases memory
void RenderFrame(void);     // renders a single frame
                            // the WindowProc function prototype
LRESULT CALLBACK WindowProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam);
// the entry point for any Windows program
int WINAPI WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance,LPSTR lpCmdLine,int nCmdShow){
    HWND hWnd;
    WNDCLASSEX wc;
    ZeroMemory(&wc, sizeof(WNDCLASSEX));
    wc.cbSize = sizeof(WNDCLASSEX);
    wc.style = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc = WindowProc;
    wc.hInstance = hInstance;
    wc.hCursor = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground = (HBRUSH)COLOR_WINDOW;
    wc.lpszClassName = "WindowClass";
    RegisterClassEx(&wc);
    RECT wr = { 0, 0, 800, 600 };
    AdjustWindowRect(&wr, WS_OVERLAPPEDWINDOW, FALSE);
    hWnd = CreateWindowEx(NULL,
        "WindowClass",
        "First Direct3D Program",
        WS_OVERLAPPEDWINDOW,
        50,
        50,
        wr.right - wr.left,
        wr.bottom - wr.top,
        NULL,
        NULL,
        hInstance,
        NULL);
    ShowWindow(hWnd, nCmdShow);
    // set up and initialize Direct3D
    InitD3D(hWnd);
    // enter the main loop:
    MSG msg;
    while (TRUE){
        if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE)){
            TranslateMessage(&msg);
            DispatchMessage(&msg);
            if (msg.message == WM_QUIT)
                break;                      }
        else{
            // Run game code here
            RenderFrame();
            }
    }
    // clean up DirectX and COM
    CleanD3D();
    return msg.wParam;
}
```



```
// this is the main message handler for the program
LRESULT CALLBACK WindowProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {
    case WM_DESTROY:
    {
        PostQuitMessage(0);
        return 0;
    } break;
    }
    return DefWindowProc(hWnd, message, wParam, lParam);
}
// this function initializes and prepares Direct3D for use
void InitD3D(HWND hWnd)
{
    // create a struct to hold information about the swap chain
    DXGI_SWAP_CHAIN_DESC scd;
    // clear out the struct for use
    ZeroMemory(&scd, sizeof(DXGI_SWAP_CHAIN_DESC));
    // fill the swap chain description struct
    scd.BufferCount = 1;                                    // one back buffer
    scd.BufferDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;     // use 32-bit color
                                //DXGI_USAGE_RENDER_TARGET_OUTPUT tells to draw in back buffer
    scd.BufferUsage = DXGI_USAGE_RENDER_TARGET_OUTPUT;      // how swap chain is to be used
    scd.OutputWindow = hWnd;                                // the window to be used
    scd.SampleDesc.Count = 4;                               // how many multisamples
    scd.Windowed = TRUE;                                    // windowed/full-screen mode
    // create a device, device context and swap chain using the information in the scd struct
    D3D11CreateDeviceAndSwapChain(
        NULL,// what graphics adapter to used
        D3D_DRIVER_TYPE_HARDWARE, //whether to use graphics HW or SW
        NULL,//used alongside the above parameter when set to software
        NULL,//indicates nhow direct 3d runs
        NULL,//different feature level for different video cards
        NULL,//indicates how many feature levels you had
        D3D11_SDK_VERSION,//version of direct 3d
        &scd,//pointer to swap chain desc struct
        &swapchain,//pointer to a pointer to the swap object
        &dev,//pointer to a pointer to the device object
        NULL,//related to feature levels
        &devcon //pointer to a pointer to device context object
        );
    // get the address of the back buffer
    ID3D11Texture2D *pBackBuffer; //texture is same as image in rendering
                              /*
                    GetBuffer() finds the back buffer on swao chain and use it to create
                        the pBackBuffer texture object.
                        1.number of back buffer(kind of index)
                        2.identifier of texture com object and to get it use __uuidof()
                        3.location of the texture object
                              */
    swapchain->GetBuffer(0, __uuidof(ID3D11Texture2D), (LPVOID*)&pBackBuffer);
    // use the back buffer address to create the render target
    //2.struct that describes render target
    dev->CreateRenderTargetView(pBackBuffer, NULL, &backbuffer);
    pBackBuffer->Release();
    /*
    sets up render targets
    1.number of render targets to set
    2.pointer to render target view object
    3.null for now
    */
    // set the render target as the back buffer
    devcon->OMSetRenderTargets(1, &backbuffer, NULL);
    // Set the viewport
    D3D11_VIEWPORT viewport;
    ZeroMemory(&viewport, sizeof(D3D11_VIEWPORT));
    viewport.TopLeftX = 0;
    viewport.TopLeftY = 0;
    viewport.Width = 800;
    viewport.Height = 600;
    // it activates viewport structs
    //1.number of viewport & 2.address
    devcon->RSSetViewports(1, &viewport);
}
```

```
// this is the function used to render a single frame
void RenderFrame(void)
{
    float red = (float)(rand() % 255) / 255.0f;
    float color[4] = { red, 0.2f, 0.4f, 1.0f };
    //float color[4] = { 0.0f, 0.2f, 0.4f, 1.0f };
    // clear the back buffer to a deep blue
    devcon->ClearRenderTargetView(backbuffer, color);
    // do 3D rendering on the back buffer here
    // switch the back buffer and the front buffer
    swapchain->Present(0, 0);
}
// this is the function that cleans up Direct3D and COM
void CleanD3D(void)
{
    // close and release all existing COM objects
    swapchain->Release();
    dev->Release();
    backbuffer->Release();
    devcon->Release();
}
```



PRACTICAL 2

With use of buffers, HLSL and create a Triangle.

This part will require all previous parts we needed to initialize direct 3d and some new com interfaces, that will help us setting up our layout primitive , the vertex buffer, the two shaders that is vertex and pixel shader.

First thing we need to do is set up our shaders , we do this by attaching our vertex shader and pixel shader to the device and device context. This shaders works through the HLSL (High Level Shader Language) part which computes per vertex values and uses interpolation between them , also it computes per pixel color values and uses interpolation between them.

Next we describe the input element (vertexes) the way we are about to structure it and then finally pass it device and device context.

Now we need to setup are vertices and their color values, these are the exact way we describe in input element description part.

Then we need a vertex buffer to work well with the graphic memory, so we describe a vertex buffer and attach it to device.

As we are aware we are not allowed to access the memory directly, we need a map resource to map our render target and then we can do whatever we need to and finally unmap it from our device context.

In render frame part we need to set our vertex buffer to device context , set the primitive type and pass on the information how many vertices we describe to draw on screen. Then we finally present it using our swap chain mechanism to render graphics on screen.

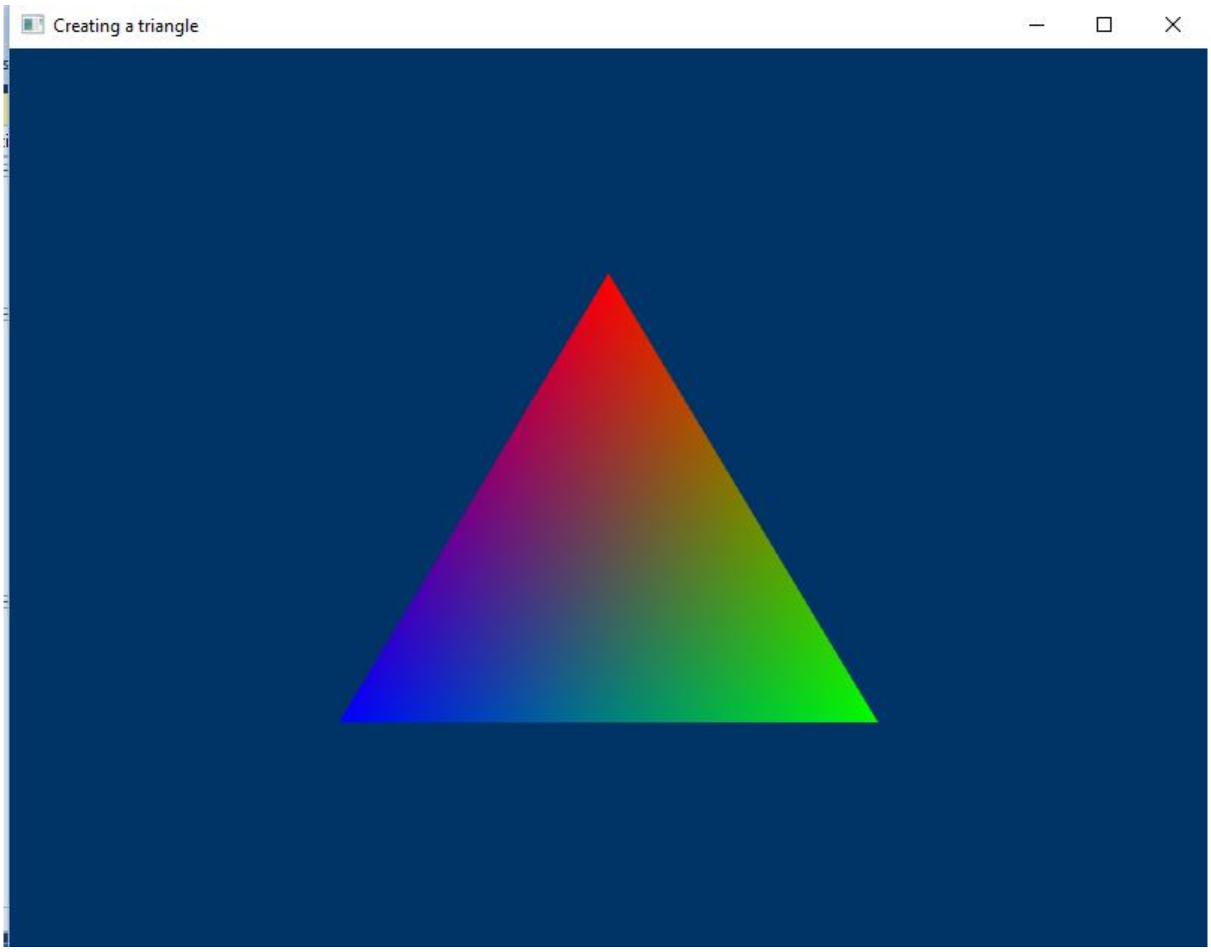
Remember till now except tessellation and stream output all the stage of graphic pipeline has been used in our example. The tessellation is the part where you focus more on the graphic part or adding lights and textures to our finally graphics that is what we’ll do next.

```
// include the basic windows header files and the Direct3D header files
#include <windows.h>
#include <windowsx.h>
#include <d3d11.h>
#include<d3dcompiler.h>
// include the Direct3D Library file
#pragma comment (lib, "d3d11.lib")
// define the screen resolution
#define SCREEN_WIDTH  800
#define SCREEN_HEIGHT 600
// global declarations
IDXGISwapChain *swapchain;             // the pointer to the swap chain interface
ID3D11Device *dev;                     // the pointer to our Direct3D device interface
ID3D11DeviceContext *devcon;           // the pointer to our Direct3D device context
ID3D11RenderTargetView *backbuffer;    // the pointer to our back buffer
ID3D11InputLayout *pLayout;            // the pointer to the input layout
ID3D11VertexShader *pVS;               // the pointer to the vertex shader
ID3D11PixelShader *pPS;                // the pointer to the pixel shader
ID3D11Buffer *pVBuffer;                // the pointer to the vertex buffer
ID3DBlob *VS, *PS;
// a struct to define a single vertex
struct VERTEX { FLOAT X, Y, Z; float Color[4]; };
// function prototypes
void InitD3D(HWND hWnd);    // sets up and initializes Direct3D
void RenderFrame(void);     // renders a single frame
void CleanD3D(void);        // closes Direct3D and releases memory
void InitGraphics(void);    // creates the shape to render
void InitPipeline(void);    // loads and prepares the shaders
                            // the WindowProc function prototype
LRESULT CALLBACK WindowProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam);
// this is the main message handler for the program
LRESULT CALLBACK WindowProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {
    case WM_DESTROY:
    {
        PostQuitMessage(0);
        return 0;
    } break;
    }

    return DefWindowProc(hWnd, message, wParam, lParam);
}
// this function initializes and prepares Direct3D for use
void InitD3D(HWND hWnd)
{
    // create a struct to hold information about the swap chain
    DXGI_SWAP_CHAIN_DESC scd;
    // clear out the struct for use
    ZeroMemory(&scd, sizeof(DXGI_SWAP_CHAIN_DESC));
    // fill the swap chain description struct
    scd.BufferCount = 1;                                    // one back buffer
    scd.BufferDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;     // use 32-bit color
    scd.BufferDesc.Width = SCREEN_WIDTH;                    // set the back buffer width
    scd.BufferDesc.Height = SCREEN_HEIGHT;                  // set the back buffer heigh
    scd.BufferUsage = DXGI_USAGE_RENDER_TARGET_OUTPUT;   // how swap chain is to be used
    scd.OutputWindow = hWnd;                                // the window to be used
    scd.SampleDesc.Count = 4;                               // how many multisamples
    scd.Windowed = TRUE;                                    // windowed/full-screen mode
    scd.Flags = DXGI_SWAP_CHAIN_FLAG_ALLOW_MODE_SWITCH;  // allow full-screen switching
```

```cpp
//create a device,device context and swap chain using the information in the scd struct
    D3D11CreateDeviceAndSwapChain(NULL,
        D3D_DRIVER_TYPE_HARDWARE,
        NULL,
        NULL,
        NULL,
        NULL,
        D3D11_SDK_VERSION,
        &scd,
        &swapchain,
        &dev,
        NULL,
        &devcon);
    // get the address of the back buffer
    ID3D11Texture2D *pBackBuffer;
    swapchain->GetBuffer(0, __uuidof(ID3D11Texture2D), (LPVOID*)&pBackBuffer);
    // use the back buffer address to create the render target
    dev->CreateRenderTargetView(pBackBuffer, NULL, &backbuffer);
    pBackBuffer->Release();
    // set the render target as the back buffer
    devcon->OMSetRenderTargets(1, &backbuffer, NULL);
    // Set the viewport
    D3D11_VIEWPORT viewport;
    ZeroMemory(&viewport, sizeof(D3D11_VIEWPORT));
    viewport.TopLeftX = 0;
    viewport.TopLeftY = 0;
    viewport.Width = SCREEN_WIDTH;
    viewport.Height = SCREEN_HEIGHT;
    devcon->RSSetViewports(1, &viewport);
    InitPipeline();
    InitGraphics();
}
// this is the function used to render a single frame
void RenderFrame(void)
{
    // clear the back buffer to a deep blue
    float c[4] = { 0.0f, 0.2f, 0.4f, 1.0f };
    devcon->ClearRenderTargetView(backbuffer, c);
    // select which vertex buffer to display
    UINT stride = sizeof(VERTEX);
    UINT offset = 0;
    devcon->IASetVertexBuffers(0, 1, &pVBuffer, &stride, &offset);
    // select which primtive type we are using
    devcon->IASetPrimitiveTopology(D3D10_PRIMITIVE_TOPOLOGY_TRIANGLELIST);
    // draw the vertex buffer to the back buffer
    devcon->Draw(3, 0);
    // switch the back buffer and the front buffer
    swapchain->Present(0, 0);
}
// this is the function that cleans up Direct3D and COM
void CleanD3D(void)
{
    swapchain->SetFullscreenState(FALSE, NULL);    // switch to windowed mode
    // close and release all existing COM objects
    pLayout->Release();
    pVS->Release();
    pPS->Release();
    pVBuffer->Release();
    swapchain->Release();
    backbuffer->Release();
    dev->Release();
    devcon->Release();
}
```

```
// this is the function that creates the shape to render
void InitGraphics()
{
    // create a triangle using the VERTEX struct
    VERTEX OurVertices[] =
    {
        { 0.0f, 0.5f, 0.0f,{ 1.0f, 0.0f, 0.0f, 1.0f } },
        { 0.45f, -0.5, 0.0f,{ 0.0f, 1.0f, 0.0f, 1.0f } },
        { -0.45f, -0.5f, 0.0f,{ 0.0f, 0.0f, 1.0f, 1.0f } }
    };
    // create the vertex buffer
    D3D11_BUFFER_DESC bd;
    ZeroMemory(&bd, sizeof(bd));
    bd.Usage = D3D11_USAGE_DYNAMIC;                // write access access by CPU and GPU
    bd.ByteWidth = sizeof(VERTEX) * 3;             // size is the VERTEX struct * 3
    bd.BindFlags = D3D11_BIND_VERTEX_BUFFER;       // use as a vertex buffer
    bd.CPUAccessFlags = D3D11_CPU_ACCESS_WRITE;    // allow CPU to write in buffer
    dev->CreateBuffer(&bd, NULL, &pVBuffer);       // create the buffer
                                                   // copy the vertices into the buffer
    D3D11_MAPPED_SUBRESOURCE ms;
    devcon->Map(pVBuffer, NULL, D3D11_MAP_WRITE_DISCARD, NULL, &ms); // map the buffer
    memcpy(ms.pData, OurVertices, sizeof(OurVertices));              // copy the data
    devcon->Unmap(pVBuffer, NULL);                                  // unmap the buffer
}

// this function loads and prepares the shaders
void InitPipeline()
{
    // load and compile the two shaders
    //DWORD dwShaderFlags = D3DCOMPILE_ENABLE_STRICTNESS;
    D3DCompileFromFile(L"Effect.fx", nullptr, nullptr, "VShader", "vs_4_0", 0, 0, &VS, nullptr);
    D3DCompileFromFile(L"Effect.fx", nullptr, nullptr, "PShader", "ps_4_0", 0, 0, &PS, nullptr);
    // encapsulate both shaders into shader objects
    dev->CreateVertexShader(VS->GetBufferPointer(), VS->GetBufferSize(), NULL, &pVS);
    dev->CreatePixelShader(PS->GetBufferPointer(), PS->GetBufferSize(), NULL, &pPS);
    // set the shader objects
    devcon->VSSetShader(pVS, 0, 0);
    devcon->PSSetShader(pPS, 0, 0);
    // create the input layout object
    D3D11_INPUT_ELEMENT_DESC ied[] =
    {
        { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0, D3D11_INPUT_PER_VERTEX_DATA, 0 },
        { "COLOR", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0, 12, D3D11_INPUT_PER_VERTEX_DATA, 0 },
    };
    dev->CreateInputLayout(ied, 2, VS->GetBufferPointer(), VS->GetBufferSize(), &pLayout);
    devcon->IASetInputLayout(pLayout);
}
```

```
// the entry point for any Windows program
int WINAPI WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance,LPSTR lpCmdLine,
    int nCmdShow)
{
    HWND hWnd;
    WNDCLASSEX wc;
    ZeroMemory(&wc, sizeof(WNDCLASSEX));
    wc.cbSize = sizeof(WNDCLASSEX);
    wc.style = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc = WindowProc;
    wc.hInstance = hInstance;
    wc.hCursor = LoadCursor(NULL, IDC_ARROW);
    wc.lpszClassName = "WindowClass";
    RegisterClassEx(&wc);
    RECT wr = { 0, 0, SCREEN_WIDTH, SCREEN_HEIGHT };
    AdjustWindowRect(&wr, WS_OVERLAPPEDWINDOW, FALSE);
    hWnd = CreateWindowEx(NULL,
        "WindowClass",
        "Our First Direct3D Program",
        WS_OVERLAPPEDWINDOW,
        50,
        50,
        wr.right - wr.left,
        wr.bottom - wr.top,
        NULL,
        NULL,
        hInstance,
        NULL);
    ShowWindow(hWnd, nCmdShow);
    // set up and initialize Direct3D
    InitD3D(hWnd);
    // enter the main loop:
    MSG msg;
    while (TRUE)
    {
        if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);

            if (msg.message == WM_QUIT)
                break;
        }
        RenderFrame();
    }
    // clean up DirectX and COM
    CleanD3D();
    return msg.wParam;
}
```


Creating a triangle

PRACTICAL 3

Use of textures in rendering.

Here we will work on our tessellation stage, as you may recall it is for texturing and adding more meshes(triangles) to our graphic that we need to render on screen, so this what we are going to do.

As you know it is far more easier to have preloaded set of textures and use it in our graphic rather than drawing it all by yourself. This is what exactly we are going to do in this part. We will load texture from an given image and simply apply on the graphic we draw on screen.

We will need some new com interfaces that will do our job so we start by describing each of them.

Loading Texture from a file

ID3D11ShaderResourceView : this will hold our texture that we load from file.

We load the texture using D3DX11CreateShaderResourceViewFromFile().

It needs a pointer to device, then a the resource file(or image we are going to use), the pointer to shader resource we created. The null parameter are helpful in describing the specifics of image or texture structure and if you are going to use multithreaded model we don't require them here.

Describing the Sample State

ID3D11SamplerState : this will hold our sampler state information.

Here we describe the sampler state or how the shader will render the texture so we create D3D11_SAMPLER_DESC object. This object takes few properties as follows,

1. The filter which describes the filtering method to use
2. It is address of U which tells what to do if U value is larger than 1.
3. It is address of V which tells what to do if V value is larger than 1.

Both of them are told to wrap around so if you scale an 256*256 image it will stretch around whatever the dimension you will have of you primitive.

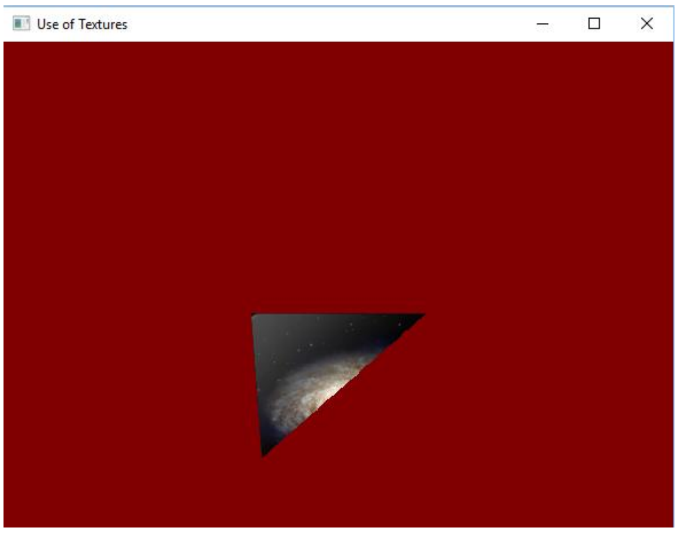
4. There is W for depth but we won't require it as we focus on the triangle.

5. Comparison function tells whether to compare with other samples.

MinLOD and MaxLOD : the mipmap level to use 0 being the most detailed to draw our texture.

After we described our sampler state, we create it by using the CreateSamplerState() method, where the first parameter is our described sampler state and the second is an interface to put the sampler state into.

```
ID3D11ShaderResourceView* CubesTexture;
ID3D11SamplerState* CubesTexSamplerState;

hr = D3DX11CreateShaderResourceViewFromFile(d3d11Device, L"Galaxy.png",
    NULL, NULL, &CubesTexture, NULL);

// Describe the Sample State
D3D11_SAMPLER_DESC sampDesc;
ZeroMemory(&sampDesc, sizeof(sampDesc));
sampDesc.Filter = D3D11_FILTER_MIN_MAG_MIP_LINEAR;
sampDesc.AddressU = D3D11_TEXTURE_ADDRESS_WRAP;
sampDesc.AddressV = D3D11_TEXTURE_ADDRESS_WRAP;
sampDesc.AddressW = D3D11_TEXTURE_ADDRESS_WRAP;
sampDesc.ComparisonFunc = D3D11_COMPARISON_NEVER;
sampDesc.MinLOD = 0;
sampDesc.MaxLOD = D3D11_FLOAT32_MAX;

//Create the Sample State
hr = d3d11Device->CreateSamplerState(&sampDesc, &CubesTexSamplerState);
```



PRACTICAL 4

Use of direction lights using diffuse type.

Diffuse light (Light type)

A simple working of light when it strikes on the surface, it is reflected off the surface at all angle equally. This way, we do not need to take the position of our camera into account. When we look at an object being hit by this kind of light, it doesn’t matter what angle we look at it, it will still have the same amount of light bouncing off and entering our eye.

Direction Lights (Light source type)

They are also known as parallel lights, they are simplest implementation of light. They have only direction and color.

Normals

Normals are used to define which direction a surface is facing. Here we will use them to determine if the surface is in the light from a light source, and how much light it should receive. Translating the vertices in world space makes the normal vector not of unit length, meaning the values in the normal vector are greater than 1 or less than zero. We fix in our HLSL part and normalize them.

Vertex normal : we define a normal when defining a vertex. Then we use this normal for each pixel on the surface to determine how much light that pixel will receive.

Light Structure:

We need to create a light structure, which basically matches the light structure we will put in our effect file. We will send this structure to the pixel shader to fill in the light structure in the effect file to implement the light. First we clear the memory when we create a new light structure. Then we have a direction , a pad, an ambient, and a diffuse member. The pad is extra value that needs to go along with the direction as it only 3d vector(x,y,z) and the HLSL needs 4d vector for computation.

Vertex Structure and Vertex Layout:

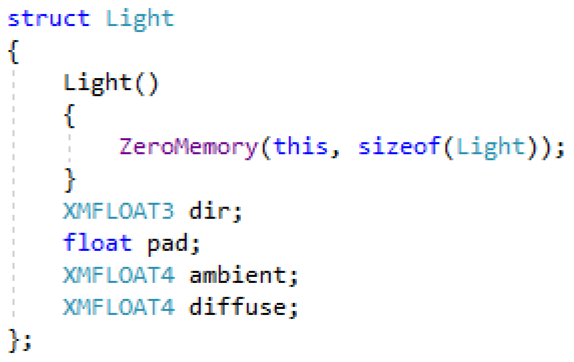
Now since we are implementing light, we need to include new member to our vertex structure. This member will hold normal data. So we can decide if and how much light is striking the surface.

Define the light:

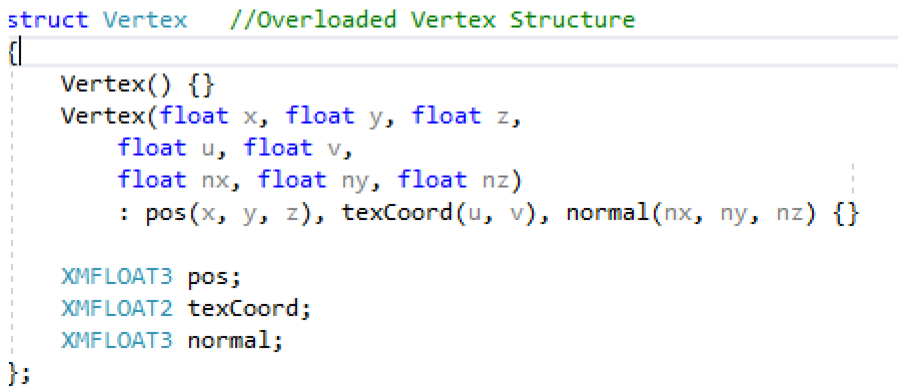
Here we will define the direction, ambient and diffuse members of our light. We will give it a white light(diffuse), and a dark ambience(so that when light is not striking the surface, it appears to be much darker than when the light is striking the surface).

HLSL Pixel Part:

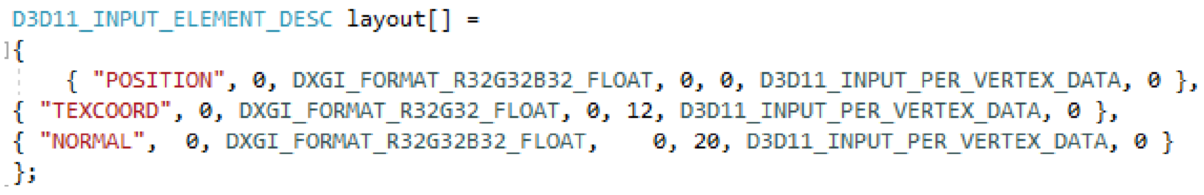
For final calculation of light in our scene, first we normalize the normal, as it may not be of unit length. Then diffuse color from our texture, after that create a new variable which hold our final color after the lighting calculations are done. We then calculate the color of the pixel with the ambient light of our lightning model. This will ensure that we can still see the parts of our scene that are not directly in light. We then calculate the final color of the pixel after we have checked how much light is hitting it. We do this by taking the dot product of direction and the normal of the surface, then multiply that with the diffuse color of our light and finally the diffuse color of our pixels texture.We saturate this to make sure no values are higher than 1.0f and then add it to finalColor, which is already holding the color of the pixel with ambient lighting.



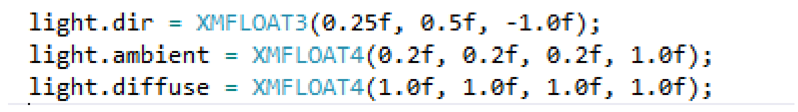
```
struct Light
{
    Light()
    {
        ZeroMemory(this, sizeof(Light));
    }
    XMFLOAT3 dir;
    float pad;
    XMFLOAT4 ambient;
    XMFLOAT4 diffuse;
};
```



```
struct Vertex   //Overloaded Vertex Structure
{
    Vertex() {}
    Vertex(float x, float y, float z,
        float u, float v,
        float nx, float ny, float nz)
        : pos(x, y, z), texCoord(u, v), normal(nx, ny, nz) {}

    XMFLOAT3 pos;
    XMFLOAT2 texCoord;
    XMFLOAT3 normal;
};
```

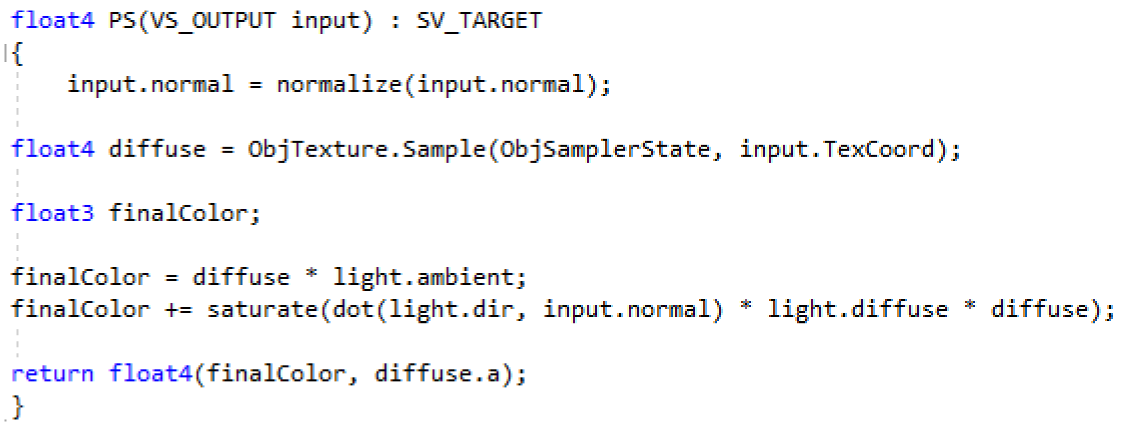


```
D3D11_INPUT_ELEMENT_DESC layout[] =
{
    { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0, D3D11_INPUT_PER_VERTEX_DATA, 0 },
{ "TEXCOORD", 0, DXGI_FORMAT_R32G32_FLOAT, 0, 12, D3D11_INPUT_PER_VERTEX_DATA, 0 },
{ "NORMAL",  0, DXGI_FORMAT_R32G32B32_FLOAT,    0, 20, D3D11_INPUT_PER_VERTEX_DATA, 0 }
};
```

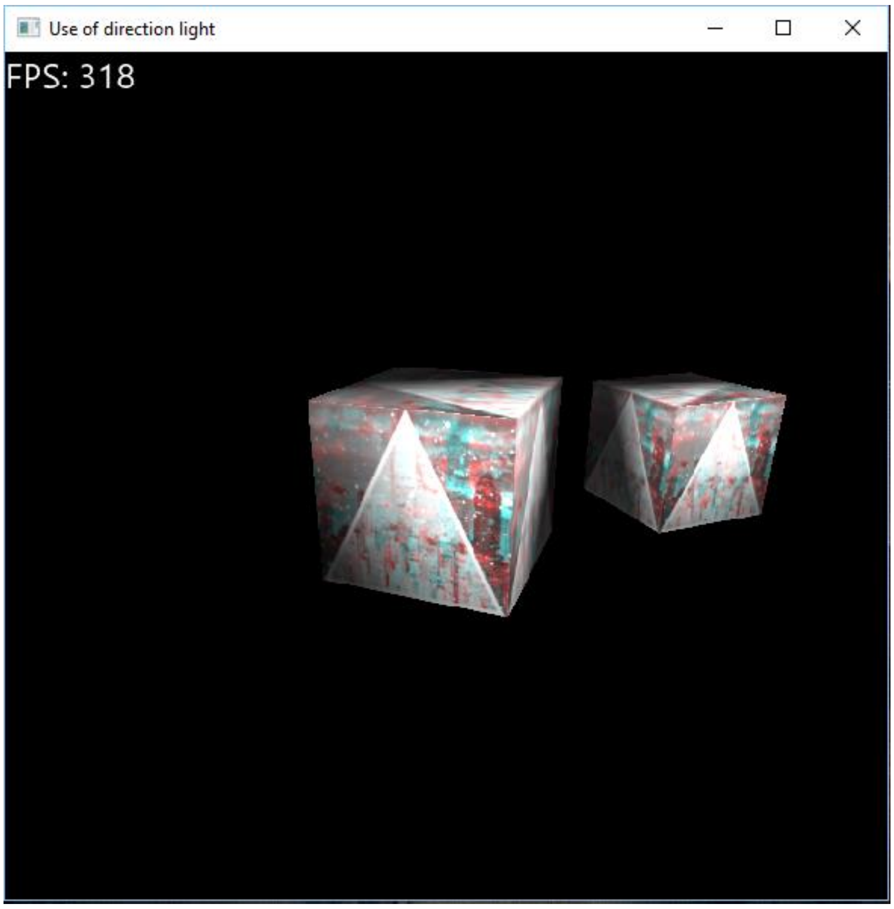


```
light.dir = XMFLOAT3(0.25f, 0.5f, -1.0f);
light.ambient = XMFLOAT4(0.2f, 0.2f, 0.2f, 1.0f);
light.diffuse = XMFLOAT4(1.0f, 1.0f, 1.0f, 1.0f);
```

HLSL PART(Pixel Shader)



```
float4 PS(VS_OUTPUT input) : SV_TARGET
{
    input.normal = normalize(input.normal);

float4 diffuse = ObjTexture.Sample(ObjSamplerState, input.TexCoord);

float3 finalColor;

finalColor = diffuse * light.ambient;
finalColor += saturate(dot(light.dir, input.normal) * light.diffuse * diffuse);

return float4(finalColor, diffuse.a);
}
```


Use of direction light
FPS: 318

PRACTICAL 5

Use of spotlights

Point lights are advances on the direction light that simply means it takes little more efforts to compute it and it adds lot more realism to scene. They require a position , its range and attenuation.

Position : Specifies where the lights are placed.

Range : The light does not continue forever like directional lights, hence we must give a range where any pixels or objects outside this range will not be lit up.

Attenuation : Is use to control how the intensity of light changes or decreased as the distance from the light source increases. We use a constant modifier, linear modifier and exponential modifier to control this factor.

Spotlights are basically just pointlight but with a direction.

Direction : This will be represented by 3d vector specifying our x,y,z axis.

Cone : This value will be the exponent in our spotlight equation. By changing this value we change the size of our spotlights cone. A lower value is wider cone, while a higher value is a smaller cone.

Spotlight equation: as spotlights are built from point lights, so we add one extra equation which will shine light in a specific direction through a cone instead of in all directions like a point light.

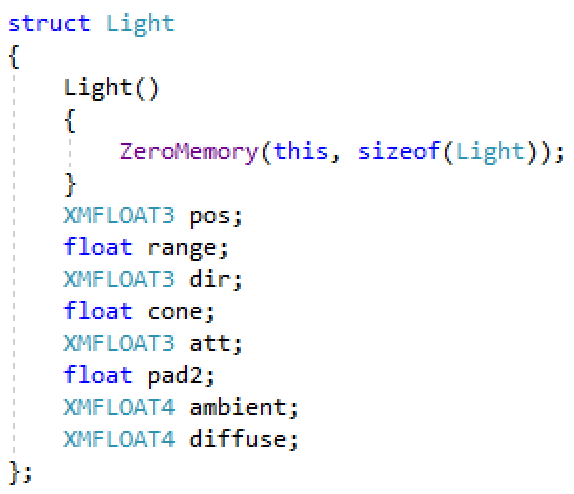
Describing the light

Here we describe our light. We set its position (0,1,0), its range to 1000.0f (so we don't see the light "cut off" in the distance), its attenuation attribute 0 to 0.4 (so the light does not get too bright when its close to camera), attribute 1 to 0.02f to give it constant fall of factor then set the attribute 3 cone value to 20.0f.

Defining the cone of light

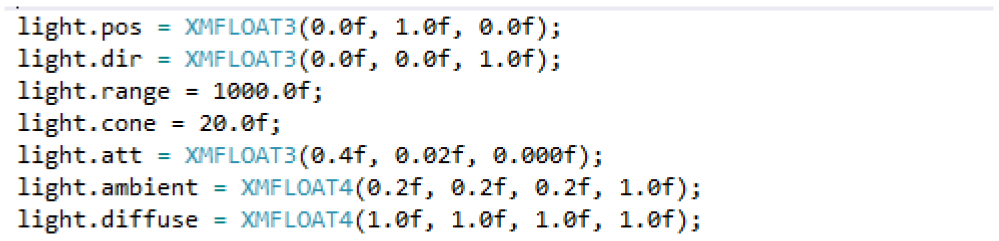
This will create a cone that our light will shine through. The first thing is find the angle between our lights direction and the direction from the light to the pixel. We use dot product to find this. Then we use the max function to make sure we do not get product less than 0.0f. A dot product less than 0.0f would be behind the position of the spotlight, therefore shining light not only in the front of the spotlight but also behind the spotlight. Finally we use the pow function with the light cone value as the exponent.

Structure of light



```
struct Light
{
    Light()
    {
        ZeroMemory(this, sizeof(Light));
    }
    XMFLOAT3 pos;
    float range;
    XMFLOAT3 dir;
    float cone;
    XMFLOAT3 att;
    float pad2;
    XMFLOAT4 ambient;
    XMFLOAT4 diffuse;
};
```

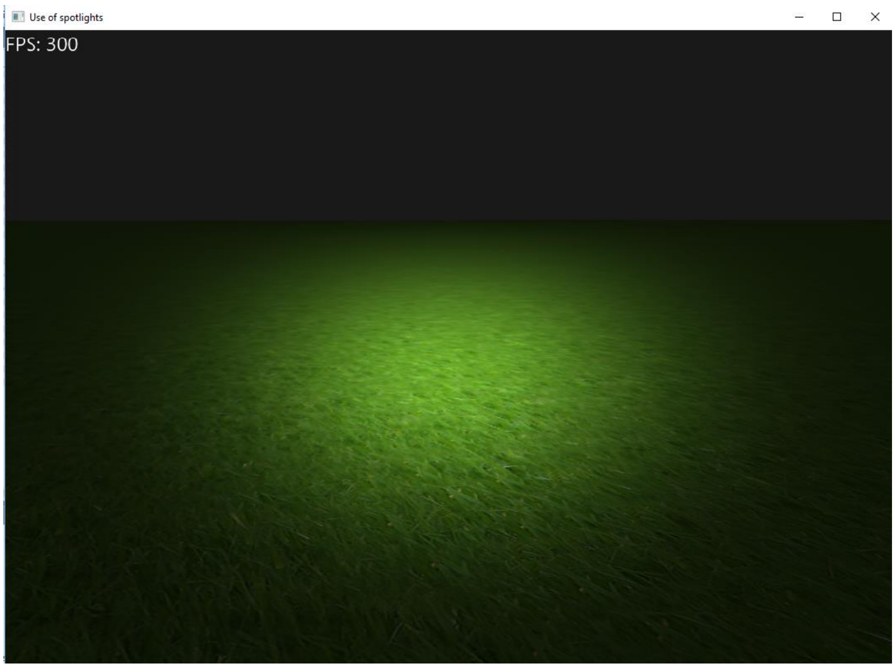
Defining the light



```
light.pos = XMFLOAT3(0.0f, 1.0f, 0.0f);
light.dir = XMFLOAT3(0.0f, 0.0f, 1.0f);
light.range = 1000.0f;
light.cone = 20.0f;
light.att = XMFLOAT3(0.4f, 0.02f, 0.000f);
light.ambient = XMFLOAT4(0.2f, 0.2f, 0.2f, 1.0f);
light.diffuse = XMFLOAT4(1.0f, 1.0f, 1.0f, 1.0f);
```

HLSL Pixel Shader

```
float4 PS(VS_OUTPUT input) : SV_TARGET
{
    input.normal = normalize(input.normal);
float4 diffuse = ObjTexture.Sample(ObjSamplerState, input.TexCoord);
float3 finalColor = float3(0.0f, 0.0f, 0.0f);
//Create the vector between light position and pixels position
float3 lightToPixelVec = light.pos - input.worldPos;
//Find the distance between the light pos and pixel pos
float d = length(lightToPixelVec);
//Add the ambient light
float3 finalAmbient = diffuse * light.ambient;
//If pixel is too far, return pixel color with ambient light
if (d > light.range)
return float4(finalAmbient, diffuse.a);
//Turn lightToPixelVec into a unit length vector describing
//the pixels direction from the lights position
lightToPixelVec /= d;
//Calculate how much light the pixel gets by the angle
//in which the light strikes the pixels surface
float howMuchLight = dot(lightToPixelVec, input.normal);
//If light is striking the front side of the pixel
if (howMuchLight > 0.0f)
{
    //Add light to the finalColor of the pixel
    finalColor += diffuse * light.diffuse;
    //Calculate Light's Distance Falloff factor
    finalColor /= (light.att[0] + (light.att[1] * d)) + (light.att[2] * (d*d));
    //Calculate falloff from center to edge of pointlight cone
    finalColor *= pow(max(dot(-lightToPixelVec, light.dir), 0.0f), light.cone);
}
//make sure the values are between 1 and 0, and add the ambient
finalColor = saturate(finalColor + finalAmbient);
//Return Final Color
return float4(finalColor, diffuse.a);
}
```



PRACTICAL 6

Creating a 2D UFO game.

1. We start by creating a 2d new project and make sure to import assets from 2D UFO tutorial.

2. Next we drag and drop sprites from the sprite folder in our scene view, first the game background and next the ufo object.

3. Then you add a Rigidbody2D component to ufo object. This has to be done to apply any type of physics logic to your component.

4. Now we want our object to move so add a new script ad edit it as follows:

5. Next you may observe that your ufo goes down when you start to play, so make gravity to zero from Rigidbody2d property.

6. Now your object should move properly. But still the object isn’t bound in space it moves everywhere around.

So here we require more physics component, we would add colliders to our ufo and the walls of our game background.

7. Add a Circle Collider 2d to ufo object, adjust the radius as appropriate.

8. Add a Box Collider 2d to background, now we need to adjust it in such a way that one side of the background is covered, it’s should look like a rectangle when we adjust. Do the same for all the sides and we should have our boundaries ready.

If you play now you should see that our ufo collides with each of the wall and does not escapes. Now to complete our game we need some asteroid like structure, so when the ufo collides with it then the asteroid disappears.

What we need is now to add the asteroid sprite named pickup from sprites folder to our scene view.

9. Since we need some collision to occur, we will add Circle Collider 2D Component to pickup object.

10. Now we need this object everywhere in our background, so simply right click and duplicate it , change the x and y position and place it all around the ufo.

Now if play it you may observe that our ufo collides with each one of the pickups or asteroids in surrounding, all we need is them to disappear when they are hit.

11. First thing we need to do is make our pickup object a prefab. So for this just drag the pickup from hierarchy and drop it the project assets folder. Now because of this you could adjust every other pickup behaviour using the prefab.

12. Select the prefab and in the inspector(properties) window add tag (Pick Up) if not set, this will allow us to use it in the script part for our player(ufo). Also in Circle Collider 2d part check the on trigger event as we would be using for collision check.

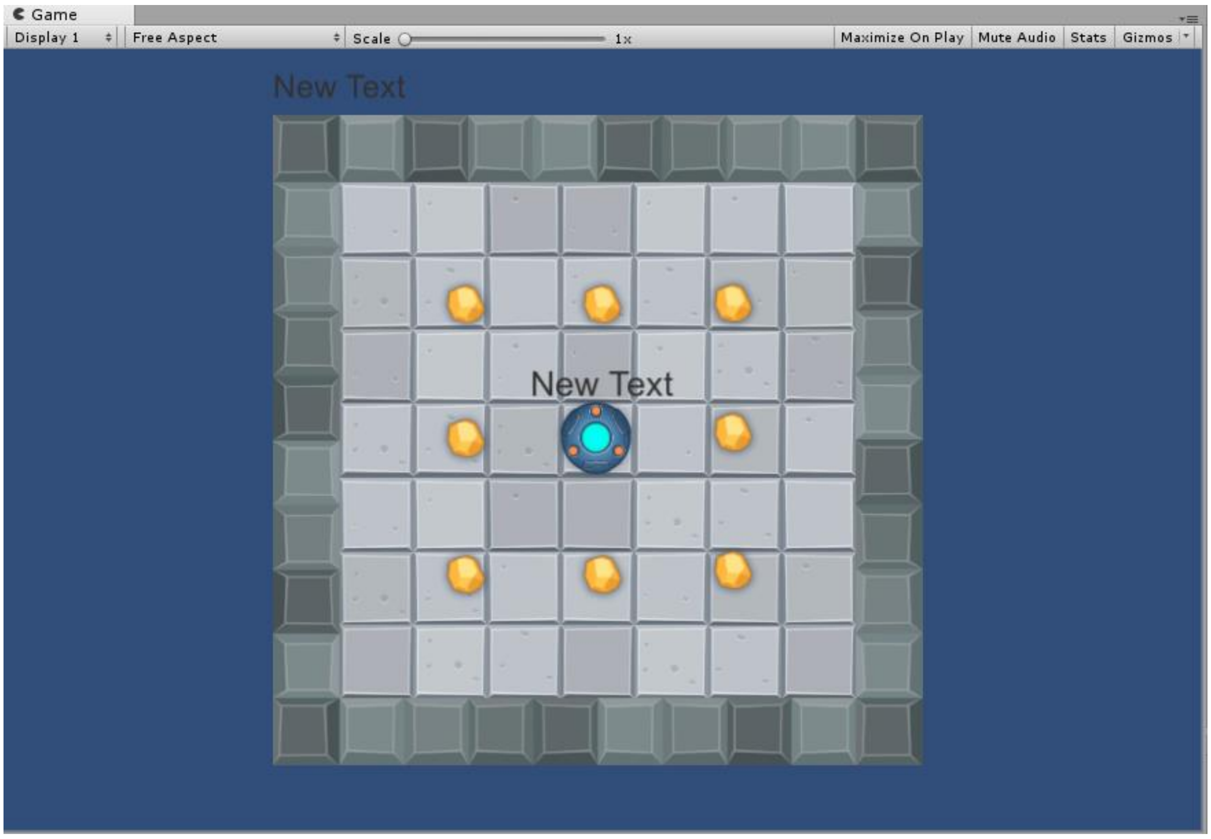
13. Now we edit our script create a new OnTriggerEnter2D (Collision), and edit as given.

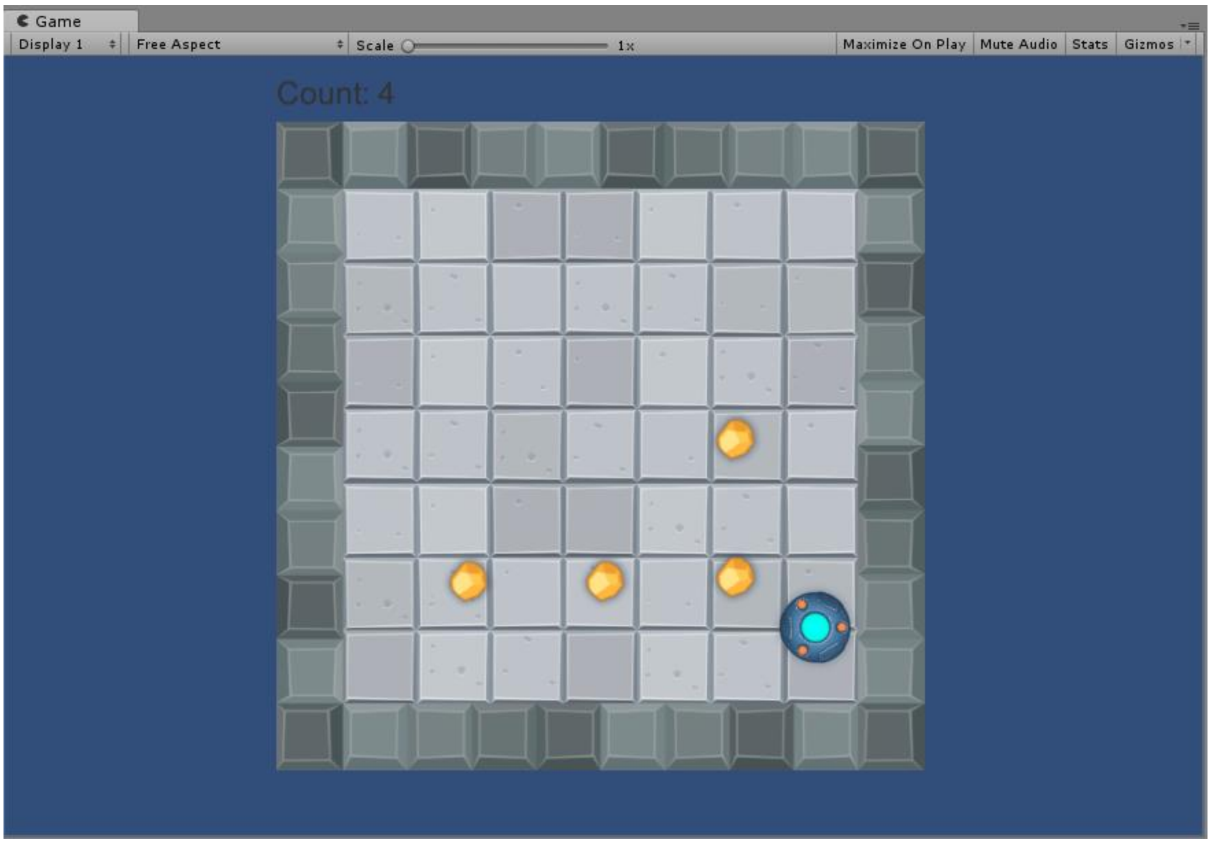
Now our game should work fine and the pickups must disappear once you collide with them.

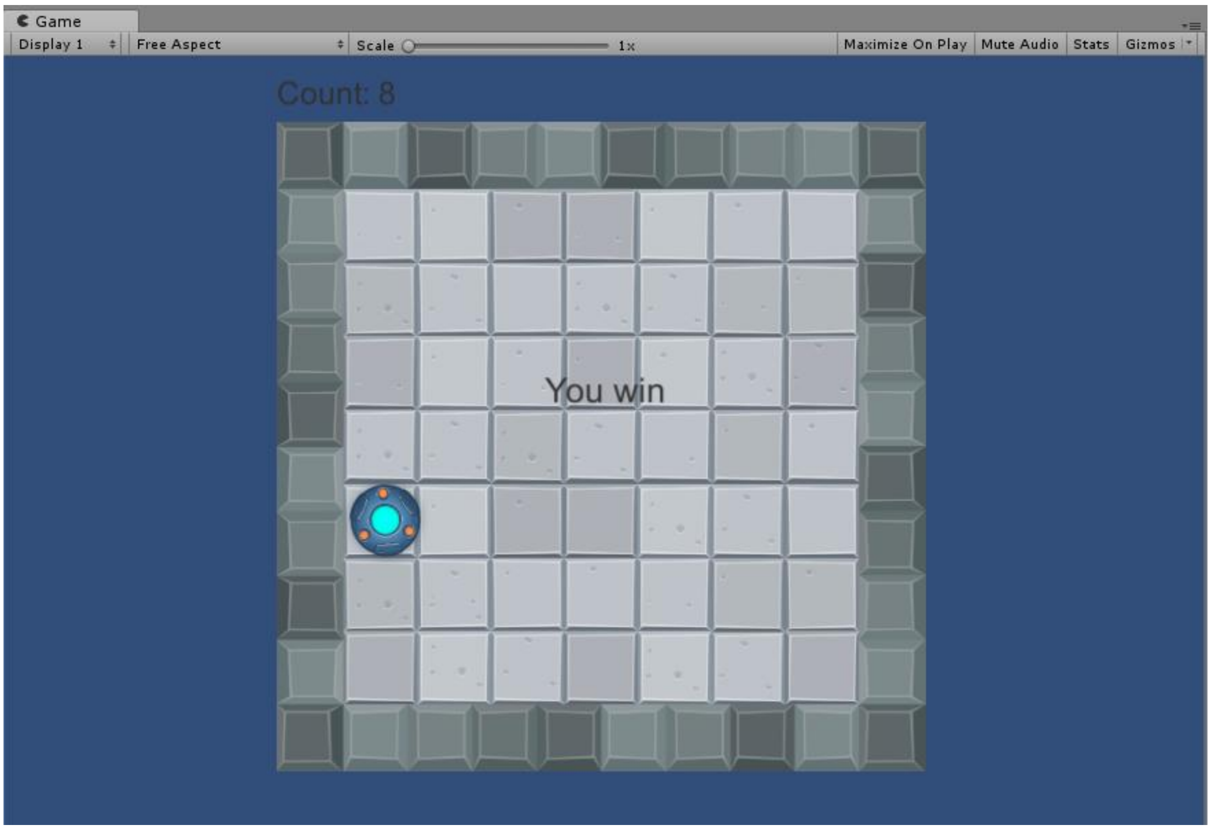
One final thing we would like to add, is some text to display our count of collecting pickups and to display win as we pick our last pickup.

15. So from hierarchy add text element and adjust it in top left of screen. An another text and place it above the ufo. Give them some appropriate tags as we will be using them in player (ufo) script. Make the appropriate changes as given in the script and then our game is ready.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
public class PlayerController : MonoBehaviour {


    public Text winT, countT;
    private int count;
    private Rigidbody2D rb2d;
    public float speed;

    void Start(){

        rb2d=GetComponent<Rigidbody2D>();
        count=0;
        winT.text="";
        setCount();
    }

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

        float mvh=Input.GetAxis("Horizontal");
        float mvv=Input.GetAxis("Vertical");
        Vector2 mov =new Vector2(mvh,mvv);


        rb2d.AddForce(mov*speed);
    }

    void OnTriggerEnter2D(Collider2D other){
            if(other.gameObject.CompareTag("PickUp")){

                other.gameObject.SetActive(false);
                count=count+1;
                setCount();
            }
    }

    void setCount(){

        countT.text="Count: "+count.ToString();
        if(count>=8){
            winT.text="You win";
        }
    }


}
```


Game
Display 1
Free Aspect
Scale
1x
Maximize On Play
Mute Audio
Stats
Gizmos
New Text
New Text


Game
Display 1
Free Aspect
Scale
1x
Maximize On Play
Mute Audio
Stats
Gizmos
Count: 4



Count: 8

You win

PRACTICAL 7

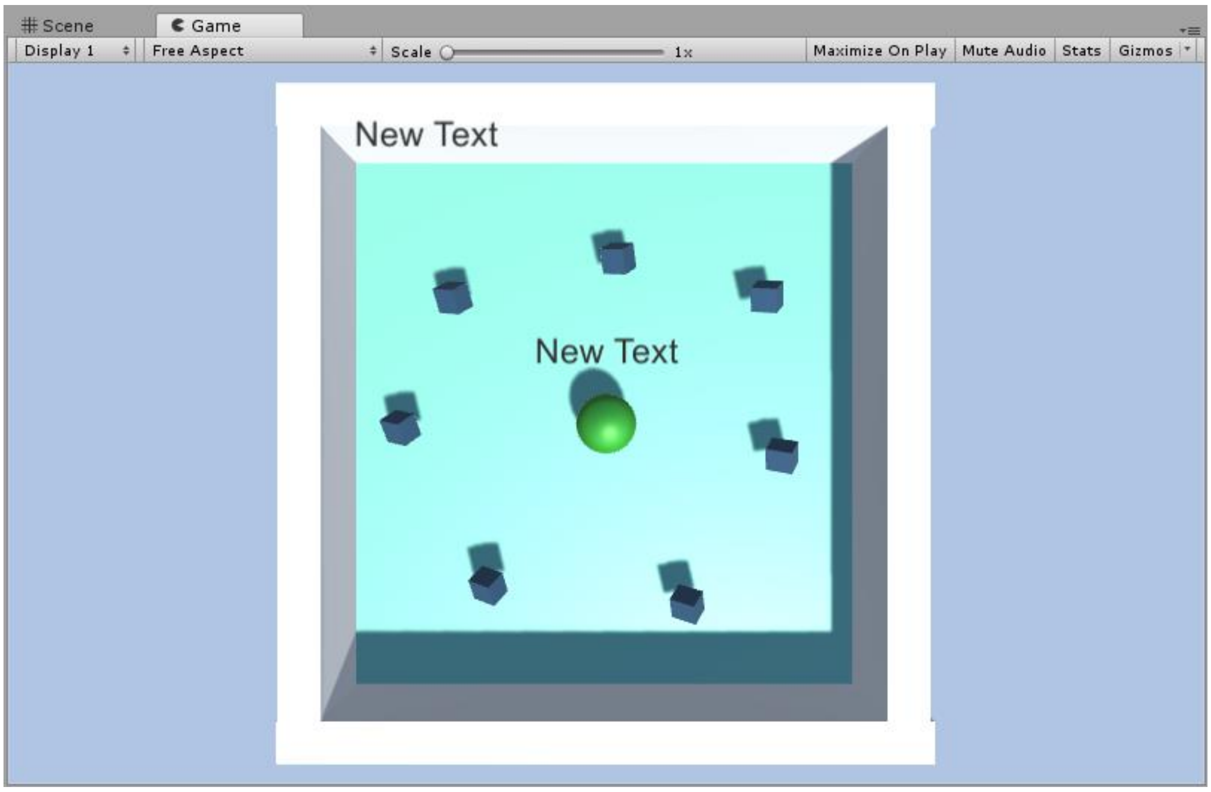
Creating a 3D game to Roll a ball.

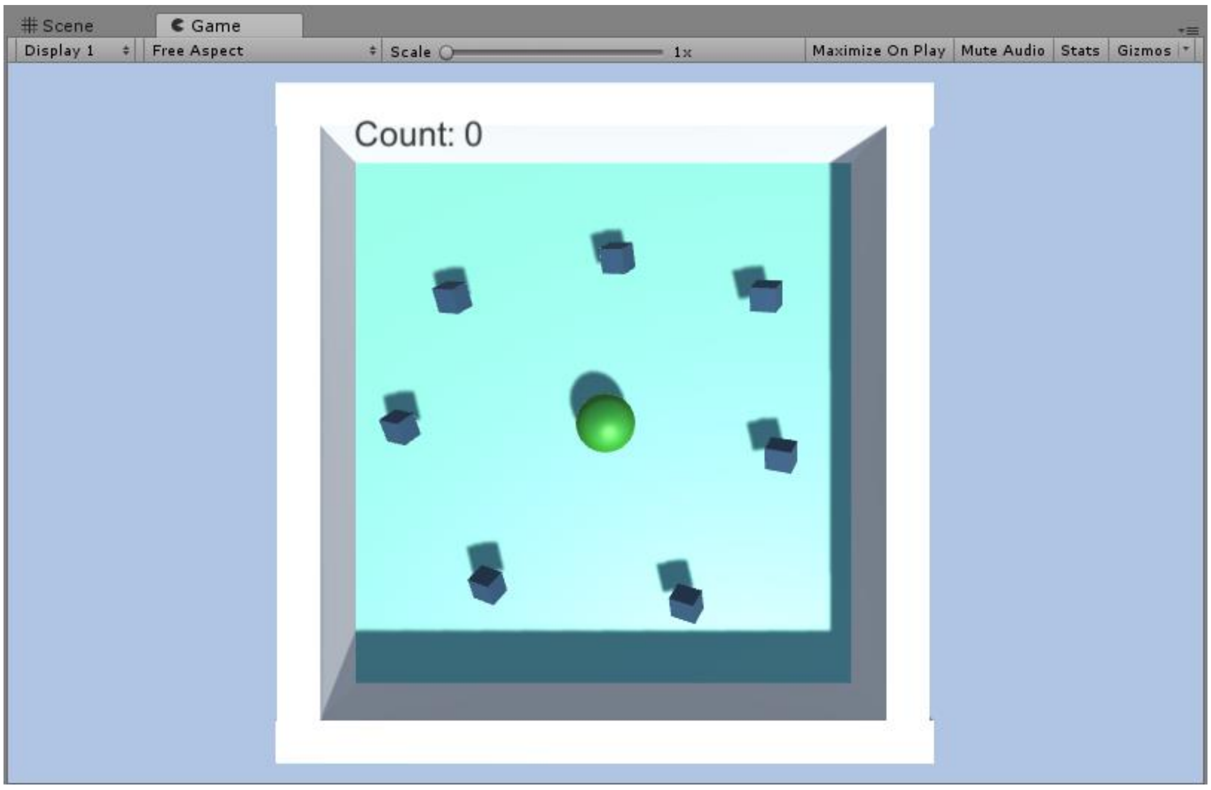
This is very similar to the previous 2D game we created, only the components now we will use and physics around them would be of 3d type.

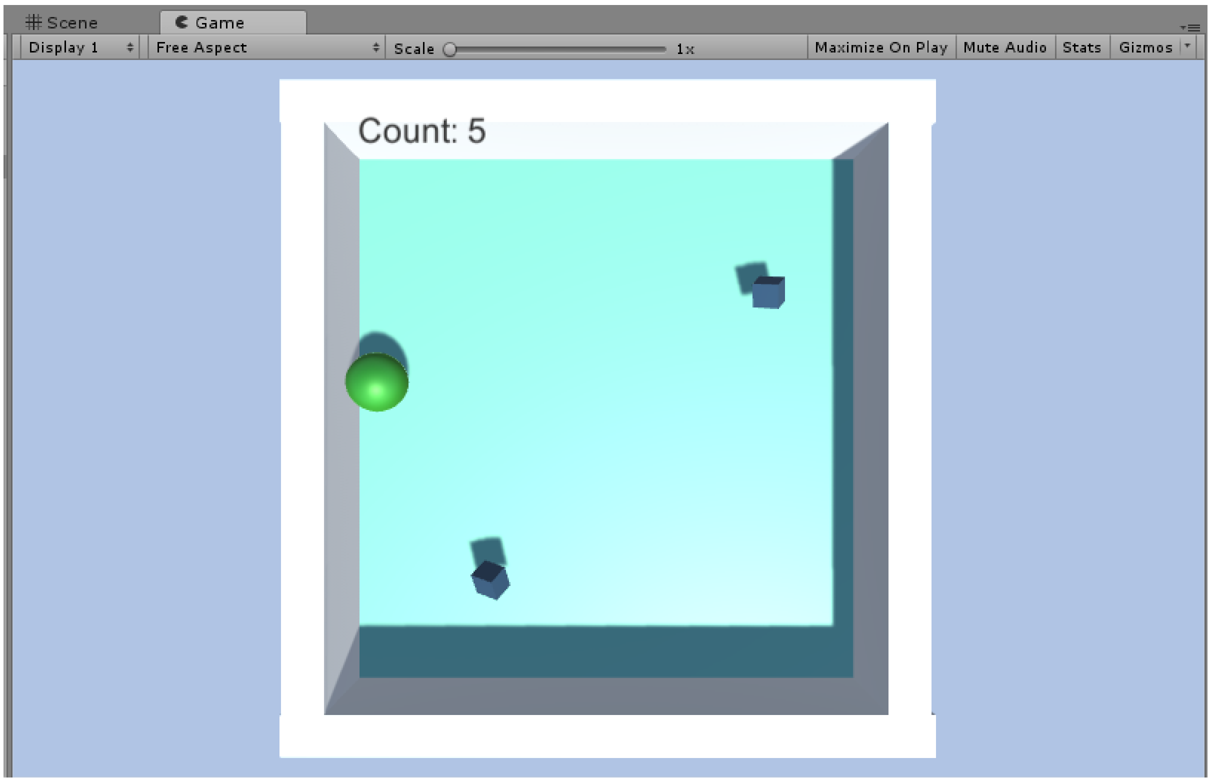
1. Create a new project, make sure it is of 3d type.
2. Next we will require a some surface in our scene, where we could place our 3d objects , as we are not going to use he sprites. So create a new plane from hierarchy view. You would require to adjust the camera , so reset it to default and rotate it about 90 degree on x axis and then move it towards the positive end of y axis.
3. Next create a sphere (our ball), that will be moving on our surface. So follow the same steps and now create a sphere. By default it is placed at origin so just move it up on y axis(0.5) and it should be perfect.
4. Now let's set boundaries for the ball otherwise it will move out of screen, so add cube the very same way from hierarchy view. Adjust each of them in rectangular component that will represent our walls on all the 4 sides.
5. As you should observe , unlike the previous 2d game we don't really need to add any kind of collider to our components in 3d as they come by default with colliders suited to their shapes. Though for our ball to move around we would need a Rigidbody attach to it, remember 2d and 3d physics system works differently and we should not use Rigidbody2D here.
6. After that attach a new script to sphere from add component in inspector (properties) window. Then edit to take speed as public variable adjusted by the user and use Rigidbody for giving it appropriate force in horizontal and vertical direction. This is very similar the way we moved our component las time around.
7. Now as everything looks white we would require to add some color to our components. So create a material now from you projects view, then in inspector window use he albedo(Color adjuster) to provide the color to material. Now all we would need is drag he material and place it on the component we want to see this color on , and we could do the same for surface and sphere.
8. Until now sphere must move smoothly all around the surface and the colliders should let the components collide. But like previously we would need pickups for our sphere to collect.

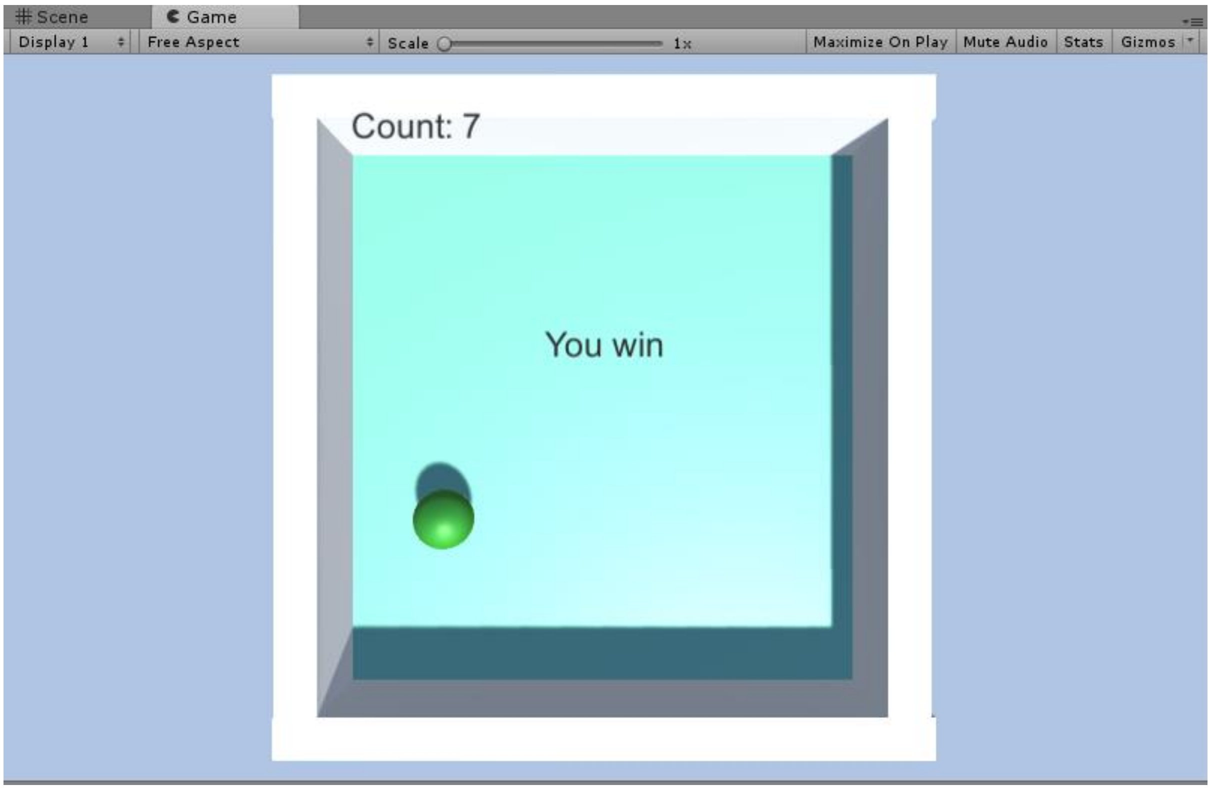
9. Add a cube rotate it 45 degrees on each of the axis and material to color it.
10. Now drag it from the hierarchy view and drop it in the projects (Asset part) view, so it could be a prefab. Now simply duplicate it all around the scene to for our ball to collide with it.
11. Once done we need to update our script as well as check the is trigger component for our pick ups, do this by selecting the prefab and then from inspector window select Is Trigger. Also remember to add a tag named Pick Up that we call in our program(use prefab for this too).
12. Next update the script as show here we use OnTriggerEnter() , you may remember we used same method just 2D at end in previous example and the rest remains same.
13. Finally add some text elements from hierarchy view (UI part) and use it in the program for the very similar logic we countered previously.

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

public class PlayerController : MonoBehaviour {

        // Use this for initialization

        public Text countT, winT;
        private int count;
        public float speed;
        private Rigidbody rb;

        void Start () {
                rb=GetComponent<Rigidbody>();
                count=0;
                winT.text="";
                setCount();
        }

        // Update is called once per frame
        void FixedUpdate () {
                float mvh=Input.GetAxis("Horizontal");
                float mvv=Input.GetAxis("Vertical");
                Vector3 mv=new Vector3(mvh,0.0f,mvv);

                rb.AddForce(mv*speed);
        }


        void OnTriggerEnter(Collider other){

                if(other.gameObject.CompareTag("Pick Up")){

                        other.gameObject.SetActive(false);
                        count++;
                        setCount();
                }

        }

        void setCount(){
                countT.text="Count: "+count.ToString();
                if(count>=7){
                        winT.text="You win";
                }
        }
}
```


Scene
Game
Display 1
Free Aspect
Scale
1x
Maximize On Play
Mute Audio
Stats
Gizmos
New Text
New Text


Scene
Game
Display 1
Free Aspect
Scale
1x
Maximize On Play
Mute Audio
Stats
Gizmos
Count: 0


Scene
Game
Display 1
Free Aspect
Scale
1x
Maximize On Play
Mute Audio
Stats
Gizmos
Count: 5


Scene
Game
Display 1
Free Aspect
Scale
1x
Maximize On Play
Mute Audio
Stats
Gizmos
Count: 7
You win

•