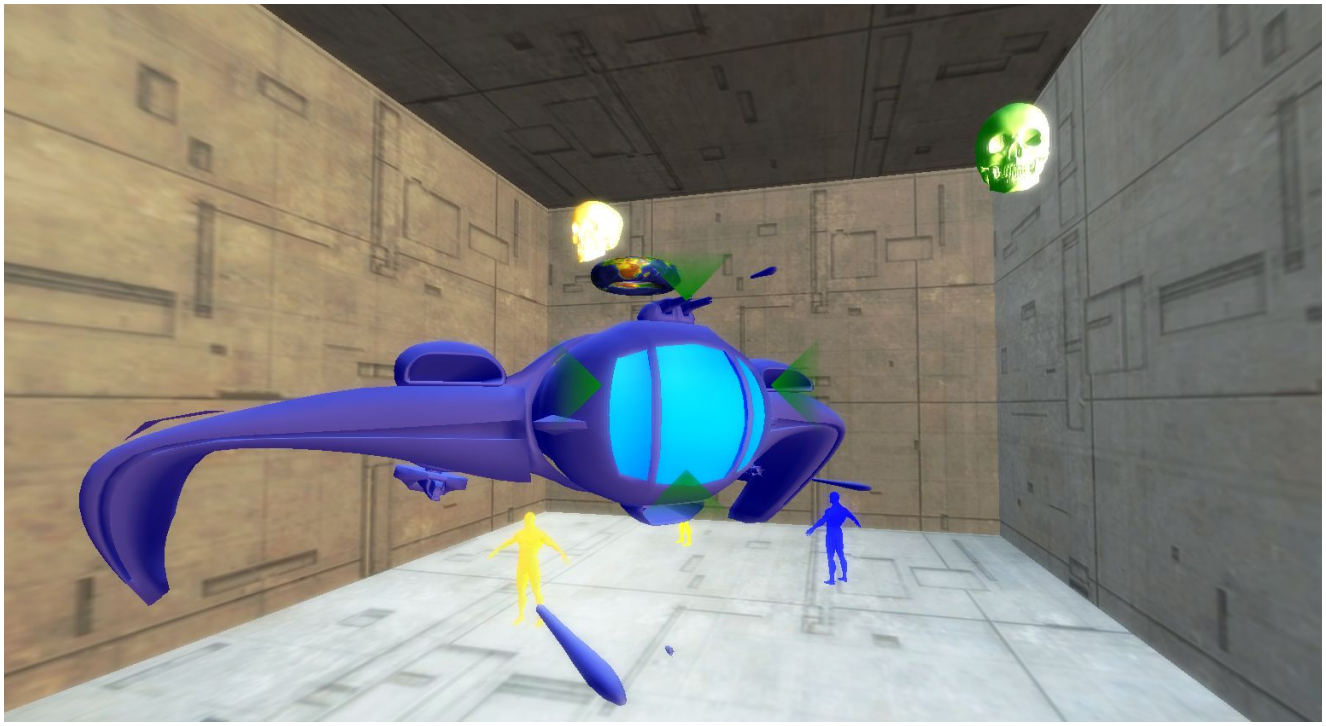
A DIVE INTO DIRECTX

Discussing the different possibilities for effects and shaders in DirectX



Rohan Menon

14/12/2019

INTRODUCTION

This paper is a discussion on how rendering can be used to create complex 3D worlds. The associated project has implementation of texturing, lighting, vertex geometry and sounds implemented in DirectX using a GPU to create fast render passes . It is also an exploration of PBREffects to render 4 different maps for height, roughness, metallic and normal mapping.

CAMERA CONTROL

This section talks about how the camera is viewing the world and is moved using keyboard inputs. The following controls define how the keyboard moves objects in the world space.

The controls are:

W: Move Forward

S: Move Backward

A, Left Arrow: Strafe Left

D, Right Arrow: Strafe Right

Up Arrow, Space: Move Upwards

Down Arrow: Move Downwards

Mouse Movement: Click and then move the mouse to change orientation

The keyboard applies a worldspace transformation every update to move the Camera in new positions.

A quaternion is used from the mouse input to define a rotation matrix for the code to move the camera in a particular direction. The key controls move the player relative to this quaternion expression using translation.

DirectX allows us to define a world transform, projection and a view using matrices. These matrices can be used to show any perspective on the screen. We can render objects into the worldspace by defining their positions and rotations through matrices.

These objects will be reduced to a projection that can be viewed by the camera screen.

The reduction to a view is important as it allows us to use Vertex and Pixel Shaders(which can be used to generate complex visual interactions, more on this later).

MESH RENDERING

Meshes are a collection of vertices that are interconnected (hence the term mesh) as either triangles or polygons. Most 3D rendering softwares uses a variety of different formats to create these meshes. This mesh data can then be fed into a rendering software to define points in space, but it also allows us to define how these vertices behave when light interacts with them.

In DirectX, the meshes can be loaded in as a .cmo or .sdkmesh file. These objects can also be used along with texture mapping and material data that allows us to define different ways that not only shows how the object is textured(coloured) but it also allows us to define different ways to show material rendering, for example something like metal will be very reflective but rubber will reflect less light and will have to look duller.

A lot of 3D rendering software have graduated to PBR(Physically Based Rendering) Effects.

“PBR rendering techniques are based on the microfacet theory, which says that the surface of each object at a microscopic level can be shown as tiny reflective mirrors called microfacets, the alignment of these mirrors is varied depending u on the roughness of the objects. Based on the roughness of a surface we can calculate the ratio of microfacets roughly aligned to the halfway vector. The more the microfacets are aligned to the halfway vector, the sharper and stronger the specular reflection.”

This can be fed into our mesh data and a 2D Texture can be applied to map out this data for different effects like metalness, roughness and emissivity.

MODEL CLASS

DirectX allows the use of a Model class that can be used to store mesh data loaded from a .cmo or .sdkmesh file. The properties of this class can be found on the DirectXTK website and allows us to load mesh data into an object along with mapped textures and other material properties. The tool that is used for this is mesh convert. But it does need the material files to be defined in a very precise way. An alternative importer to get vertex

or mesh data is assimp.

Multiple DirectX::Effects can be defined on this object to generate complex lighting, geometry and other interesting behaviours for the objects. In the code example, environmental and basic effects are being applied to render colour data on models that can then be modified not only on a single model class but as a FactoryEffect to do fast and efficient lighting and rendering calculations on a wide variety of models.

EFFECTS

The Effects class in DirectX allows one to render a variety of different render passes and also combine them into a single result that is then used when rendering the objects. There are 2 special things about effects in particular,

1. They are able to be updated while rendering is going on
2. They can be copied from on mesh to another and multiple instances can be run together on the GPU with relative ease

An effect encapsulates all of the render states required by a texture output(Normal mapping, albedo, etc) into a single rendering function called a technique. Multiple passes of the effect can be used to generate a combination of both by applying it one after the other in the render loop.

Effects use HLSL shaders and can be linked to them for a wide variety of uses. Effects also allow control for how different elements in the class will be used for different stages of hardware rendering, so we can define changes for how a normal map is applied to create an effect of water on it but this can also be expanded across multiple objects with every object having its own definition on how reflective these patches are. This allows for much greater variety as well as control over how to make Effects mapped over multiple objects a lot more manageable.

LIGHTING

Lighting in DirectX is achieved through the use of Blinn-Phong lighting in the most generic case. There are heavier and faster lighting methods available depending on whether one would prefer to have higher detail or better running performance(For example PBREffects).

The DirectX Model and Effects classes can be used to make a multitude of different

Lighting parameters, like defining lights that can be controlled from outside the mesh to allow for light mapping, generation of reflectivity data or even regular colour mapping with a Blinn-Phong light interaction.

Since .sdkmesh and .cmo objects usually have all their materials stored within themselves, this data can be used for even more complex mapping algorithms that usually needed a separate infrastructure to implement. A lot of model data can be useful in determining how an object is lit, including normal map information and roughness/metalness information.

This is not only about how light interacts when it hits an object but also about how light can be made to emit out of an object using an emissive texture allowing the representation of scattering of light on shiny metal objects for example. There are many

TEXTURES AND MATERIALS

Textures can be applied on objects to map out different colour data. But textures in themselves don't actually "texture" an object. They only provide basic colour map data to the object, usually through the use of a UV map.

But using the power of HLSL shaders on GPUs, we can store a lot more information for a mesh in the form of different texture maps that also define light bouncing properties, smoothness, reflections, etc.

This way to define the essence of how an object reacts to light is called a material. Most material definitions for mesh data are stored on .mtl files that need to be integrated into mesh data as a .sdkmesh or .sdkmesh2 file.

Materials are very useful in the sense that when rendering a game engine we can simply define most of its material data into different UV maps and then when the mesh is in the scene, we can use different shaders like IBR, material definition, PBR etc to create a more realistic representation of how light will bounce off the object and also use it for rendering the objects in a believable fashion.

SHADERS

The Vertex and Pixel shaders are used to display observations on a 3D space and 2D space respectively. The vertex shader is applied on vertices in space and can be used to define modifications in geometry. The Pixel Shader is used to create changes in views

and allow for cheaper calculations to render advanced effects like lighting etc.

Shaders originally were used in computer games programming purely for rendering complex pieces of geometry. But the concept of a graphics shader has evolved into multiple uses from physics, AI to procedural generation through the use of compute shaders. Hence the new definition of a shader should now be any piece of code that can be more efficiently run using the significant calculation performance of a GPGPU without the need for cached memory information.

SOUNDS

DirectX also allows the use of XAudio 2.8 as a sound rendering solution. It has multiple ways to not only play a sound but allow it to be used with 3D positioning, pitch blending and wavebanks for storing a wide variety of sounds together.

AN EXAMPLE LEARNING APPLICATION

The application that was made alongside this report was initially meant to have PBREffects processing using the DirectX engine. A model that has different properties to use this integration has been included in the source code but due to a lack of time it was not implemented as the PBREffects documentation on DirectXTK is particularly lacking. And getting an example .sdkmesh that was able to have the correct formatting for use with PBREffects and rendered properly proved to be quite difficult. Especially being able to create the correct lighting interaction that was required from the material definitions and the fact that DirectX::PBREffect did not allow the usage of a metallic or roughness texture map and instead used constant values to define them.

The associated project has the implementation for a simple Pixel shader using bloom to spread out the output of the view matrix and applying a shine alpha on it that allows it to glow slightly brighter in white spaced areas and gives a nice ‘pseudo lighting’ effect in the corners of the generated room. This is done by storing an extra texture in the swapchain and rerendering it when we redraw our frame.

A basic effect is being used to render a set of 4 triangles as a reticle in the center of the screen. An animation is being rendered on them that allows the triangle to scale and warp in different ways using a value set in the Update method.

DirectX::Effects have been used instead to showcase how EffectsFactory can be used on

multiple objects of the same type when rendering pixel data to a shader. There are 3 different people being rendered below the large ship that have ambient lighting, ambient and 1 diffuse light and ambient lighting and 2 diffuse lights to show how the same effect factory can be used on multiple objects but also show that an effects factory instance needs to be very careful when updating effects as well as the changes should not carry forward to the next object using the effects factory. Another thing to notice is how easy it is to do lighting manipulation using the inbuilt effects class to create interesting outcomes with respect to lighting. Fog has also been implemented a certain distance away as a highlighting tool because it seemed like the people under the ship were being lost in the surroundings.

Animations are being done on multiple objects to show knowledge about world space transformations. The lights associated with various effects being used by the models are also animated based on a rotationFactor that is being changed in update.

Sounds have been added to the scene with an implementation showcasing positional audio(the sci fi hum that is heard in the ring in the centre of the scene) and pitch blending that is being done on the ambient bird noises to give it a slightly eerie tone.

The code in itself lacks structure but is a lot cleaner to explain how different objects are being rendered as compared to most example code online. This cleanup was done more for the programmer who wrote the code as the different parts of DirectX graphics programming can be quite confusing and it is a cautionary tale on how one should be careful towards structure when writing it.

REFERENCES

DirectX Tool Kit Getting Started (2019) *API Documentation*. Available at: https://github.com/Microsoft/DirectXTK/wiki/Getting-Started (Accessed: 14/11/2019)

Learn OpenGL: PBR Theory
Available at: https://learnopengl.com/PBR/Theory (Accessed: 28/11/2019)

DirectX Effects guide
Available at:
https://docs.microsoft.com/en-us/windows/win32/direct3d11/d3d11-graphics-programming-guide-effects

MeshConvert Readme
Available at:
https://github.com/Microsoft/DirectXMesh/wiki/meshconvert

How to use PBR textures in Blender
Available at:
https://www.cgbookcase.com/textures/how-to-use-pbr-textures-in-blender

Real-Time Rendering, Fourth Edition
Available at:
https://www.amazon.co.uk/Real-Time-Rendering-Fourth-Tomas-Akenine-M%C3%B6ller/dp/1138627003
By Tomas Akenine-Mo ¨ller, Eric Haines, Naty Hoffman

Adding DirectX toolkit for audio
Available at:
https://github.com/microsoft/DirectXTK/wiki/Adding-the-DirectX-Tool-Kit-for-Audio

Physically Based Rendering in DirectX11
Available at:
https://himanshupaul.com/2018/03/13/physically-based-rendering-directx-11/
By Himanshu Paul