CMP301 Graphics Programming with Shaders

Dr Paul Robertson

p.robertson@abertay.ac.uk

Module Overview

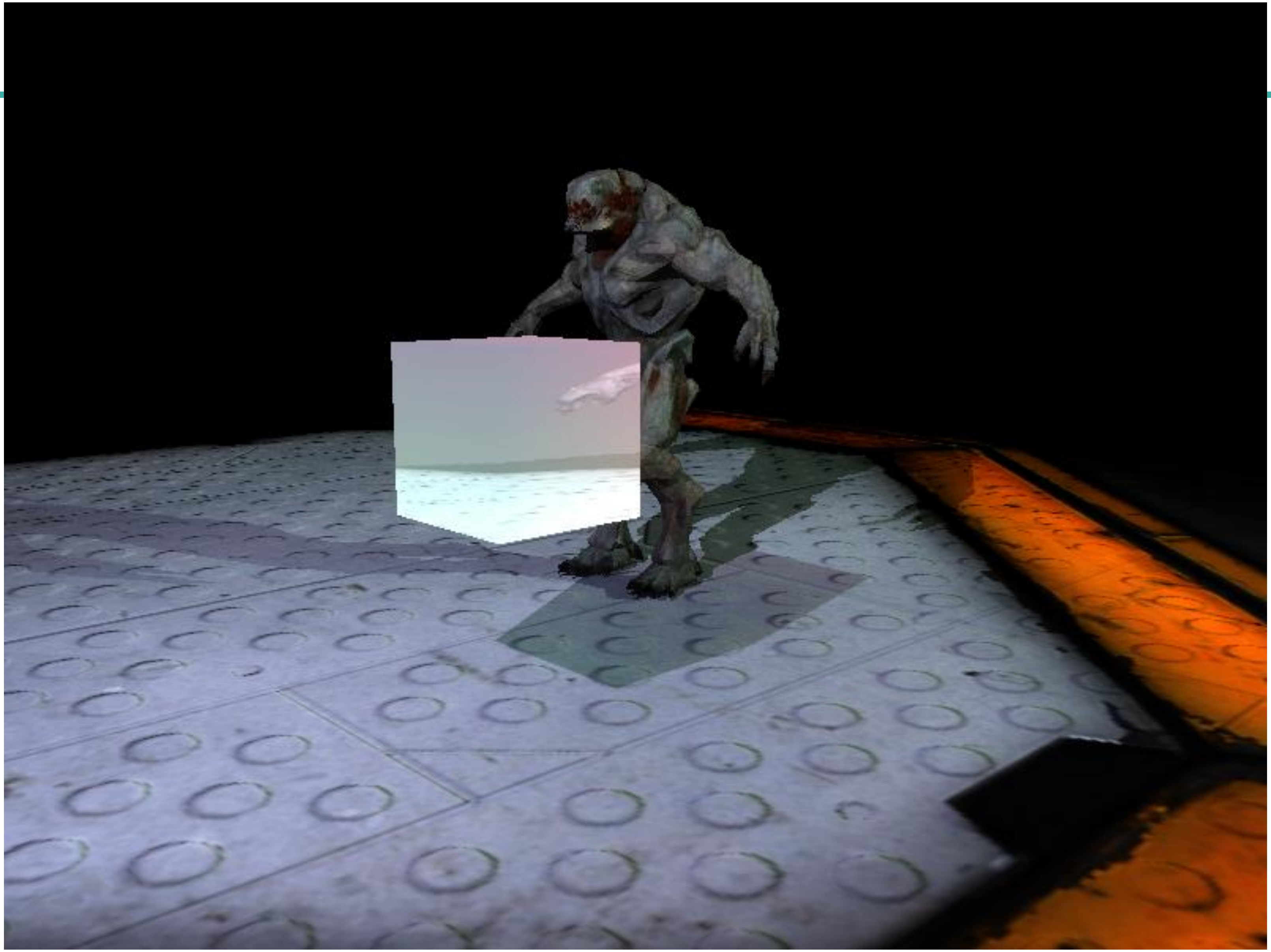
- What will you learn
  - What shaders are
  - What different shaders do
  - How to create and use shaders
  - This will include
    - Per pixel lighting
    - Vertex manipulation
    - Post processing
    - Tessellation
    - Geometry generation on the GPU
    - And a few other things

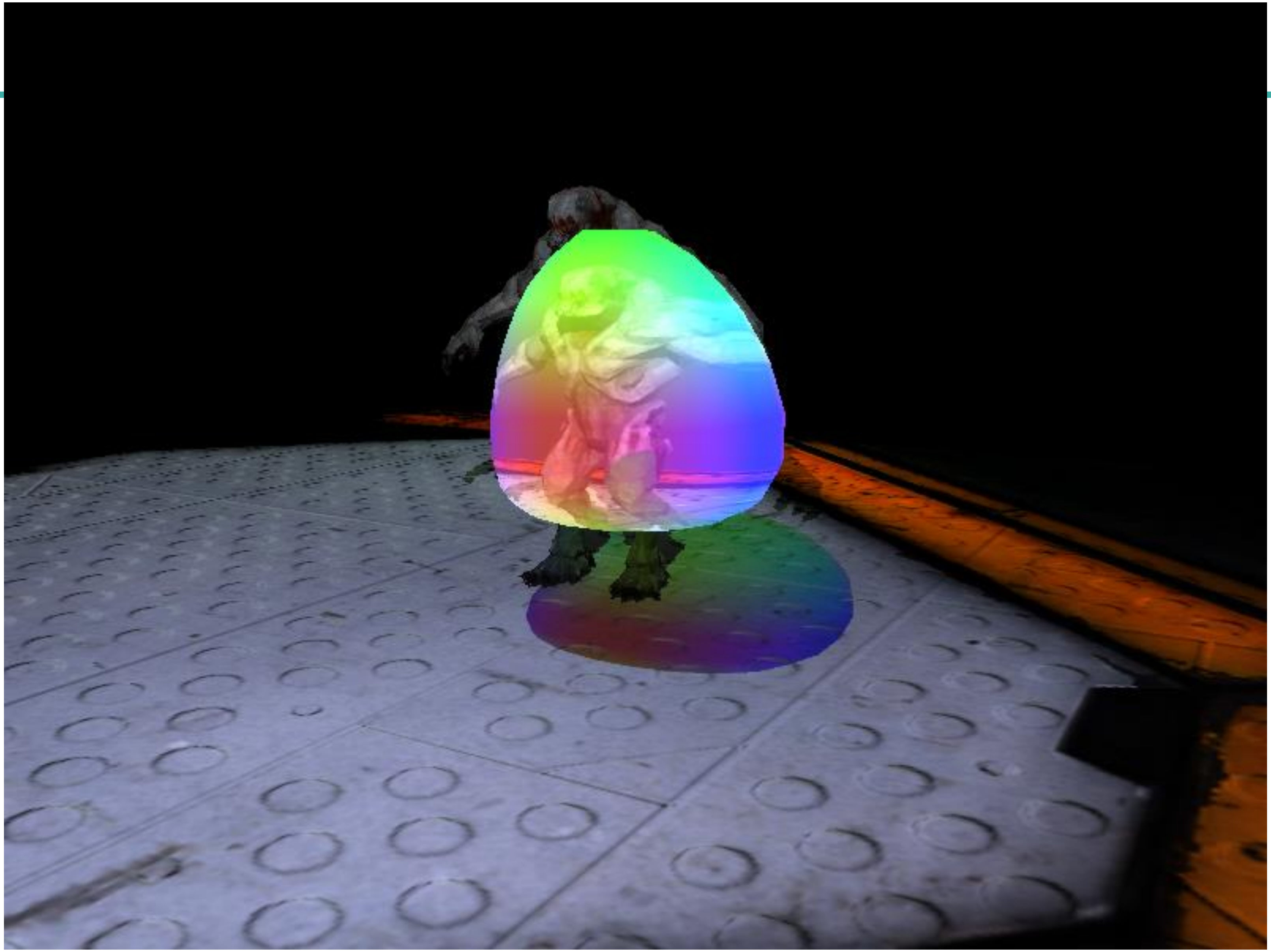
Module Overview

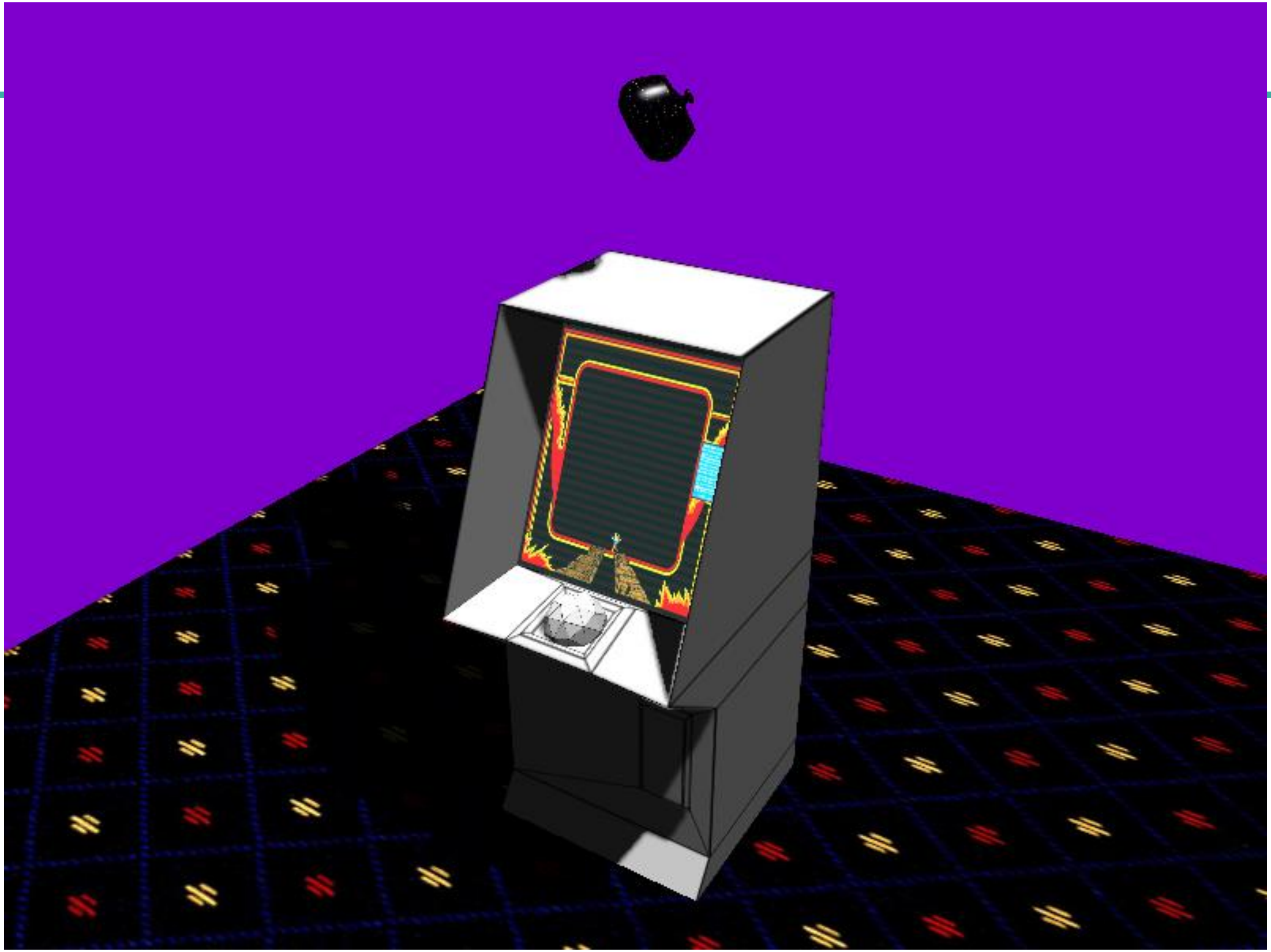
- Module breakdown
  - Each week
    - A Lecture
    - A two hour lab
    - I expect 6 hours of independent work a week
- The development software required is installed on many machines throughout the university (not just the lab machine) also accessible for home

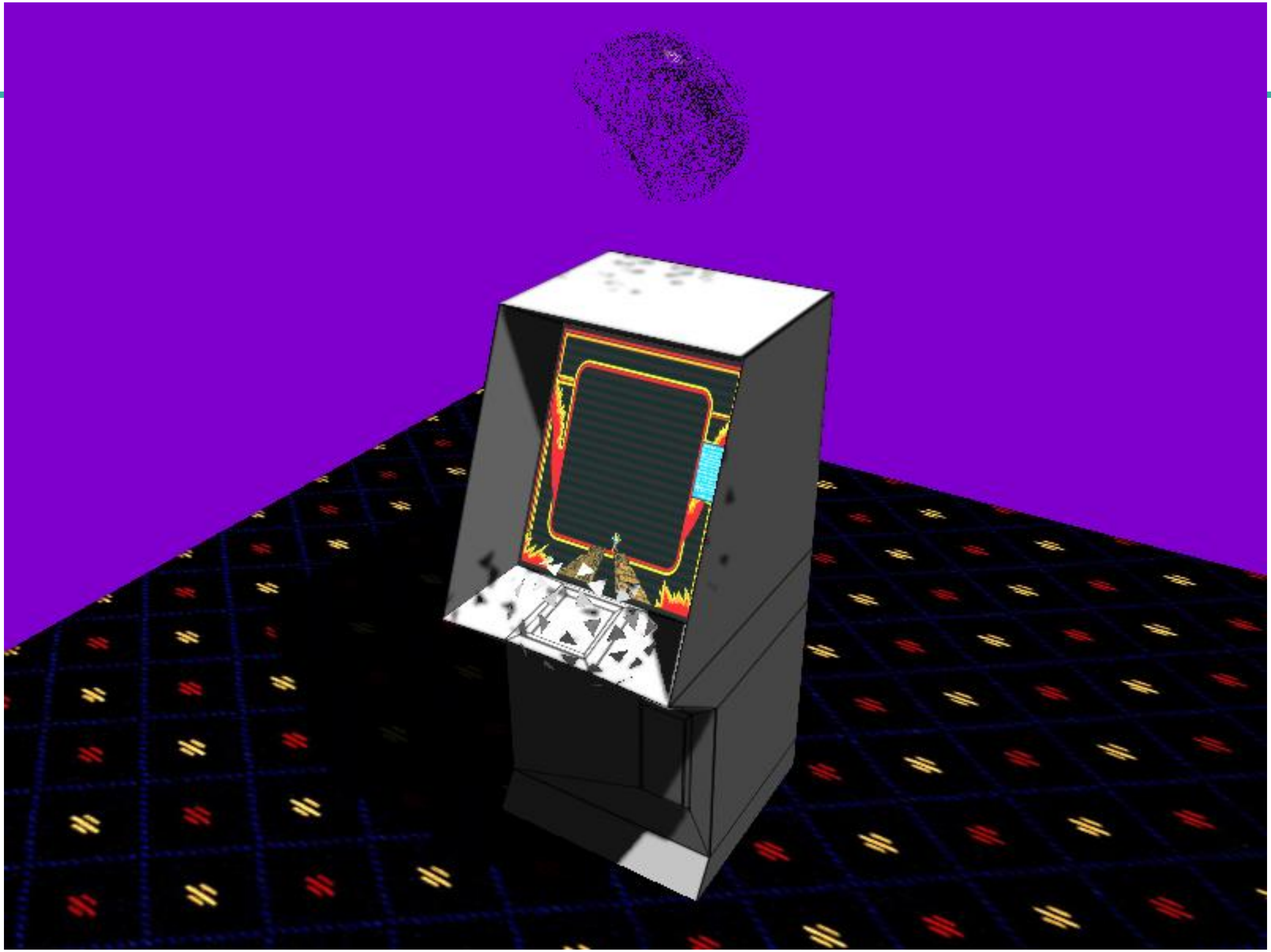
Module Overview

- Assessment
  - 100% coursework
    - Create your own shaders and display them with a simple application
    - Documentation explaining the shaders created in detail
- Some examples from last year

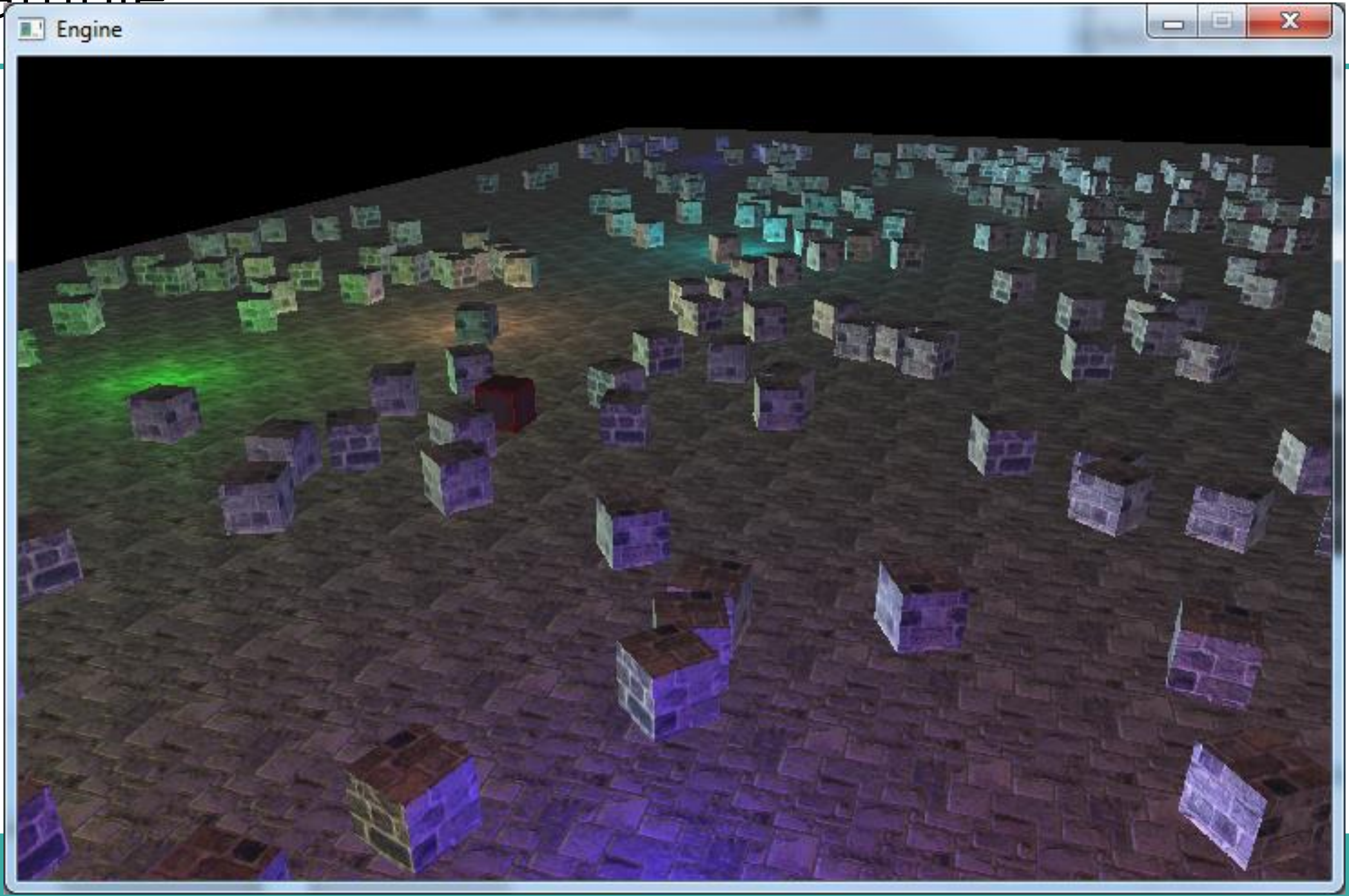


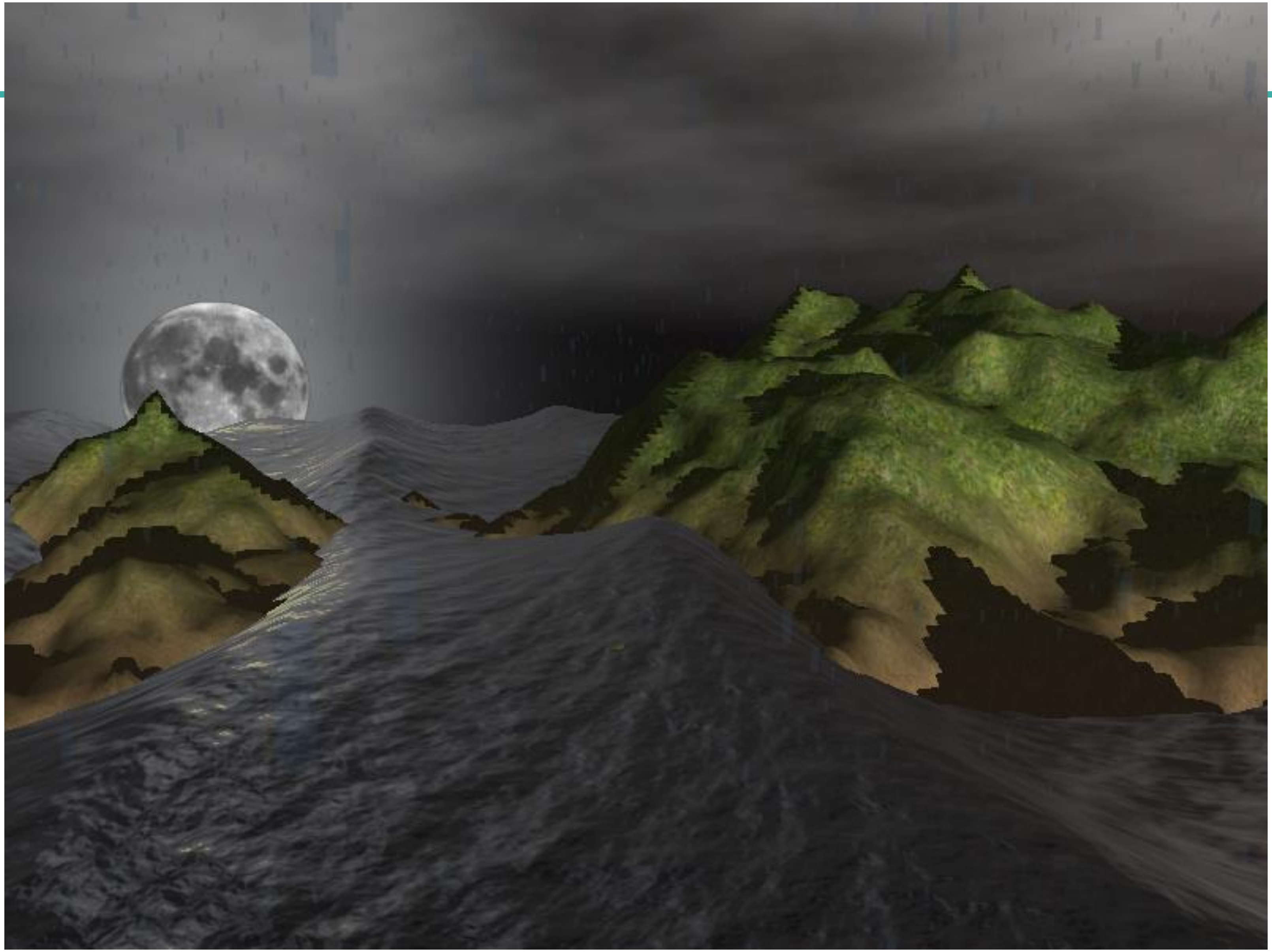






Example





What is expected of you

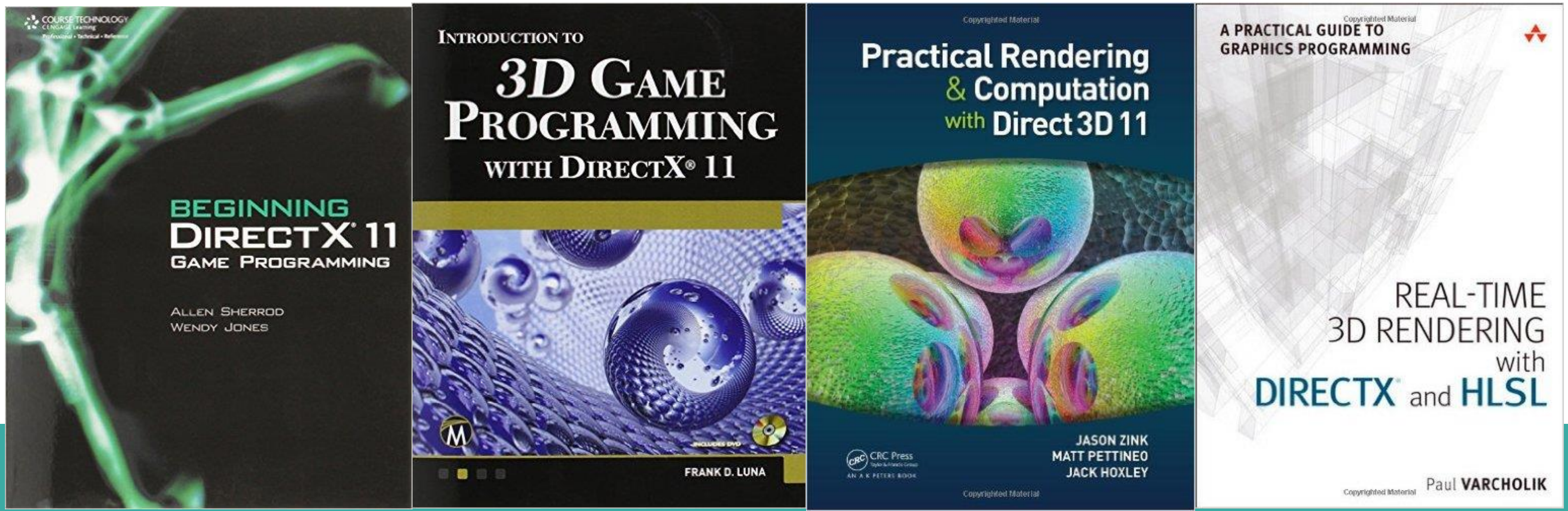
- 6 hours personal study time
- Lab tasks will require some research
- No food or drink in the lab

What you can expect

- Lecture + lab content will be available (at least) 24 hours in advance of class
  - Difficult when class is 9am Monday
- Recorded lectures
  - Published 1 week after
  - Using new software
- I’ve setup a slack channel for the module
  - For asking questions and helping class mates both in and out of class time

Recommended reading

- Beginning DirectX11 Game Programming by Allen Sherrod
- 3D Game Programming with DirectX11 by Frank Luna
- Practical Rendering & Computation with Direct3D11 by Jason Zink
- Real-Time 3D Rendering with DirectX and HLSL: A Practical Guide to Graphics Programming by Paul Varcholik



Reminder

- Your 3rd year grades WILL contribute to honours classification!!!!!!

Overview

- Overview of Direct3D
- What are shaders
- The programmable pipeline
- Framework
- My first shader(s)

This is Direct3D

- We will be using Direct3D/X 11
  - Compared to last year the pipeline has changed
    - Last year we used an Emulated Fixed Function Pipeline
    - Now a programmable pipeline
- Last year we used an old version of OpenGL
  - Newer versions are also Programmable Pipeline
  - Techniques are transferrable

Direct3D

- Quite big differences between versions of Direct3D
  - Direct3D 11 added
    - General-purpose computing on the GPU (GPGPU) and an API DirectCompute to handle it
    - True multi threaded rendering support
    - Shader Model 5.0
    - Allows increased texture resolution
    - The list goes on

What are shaders

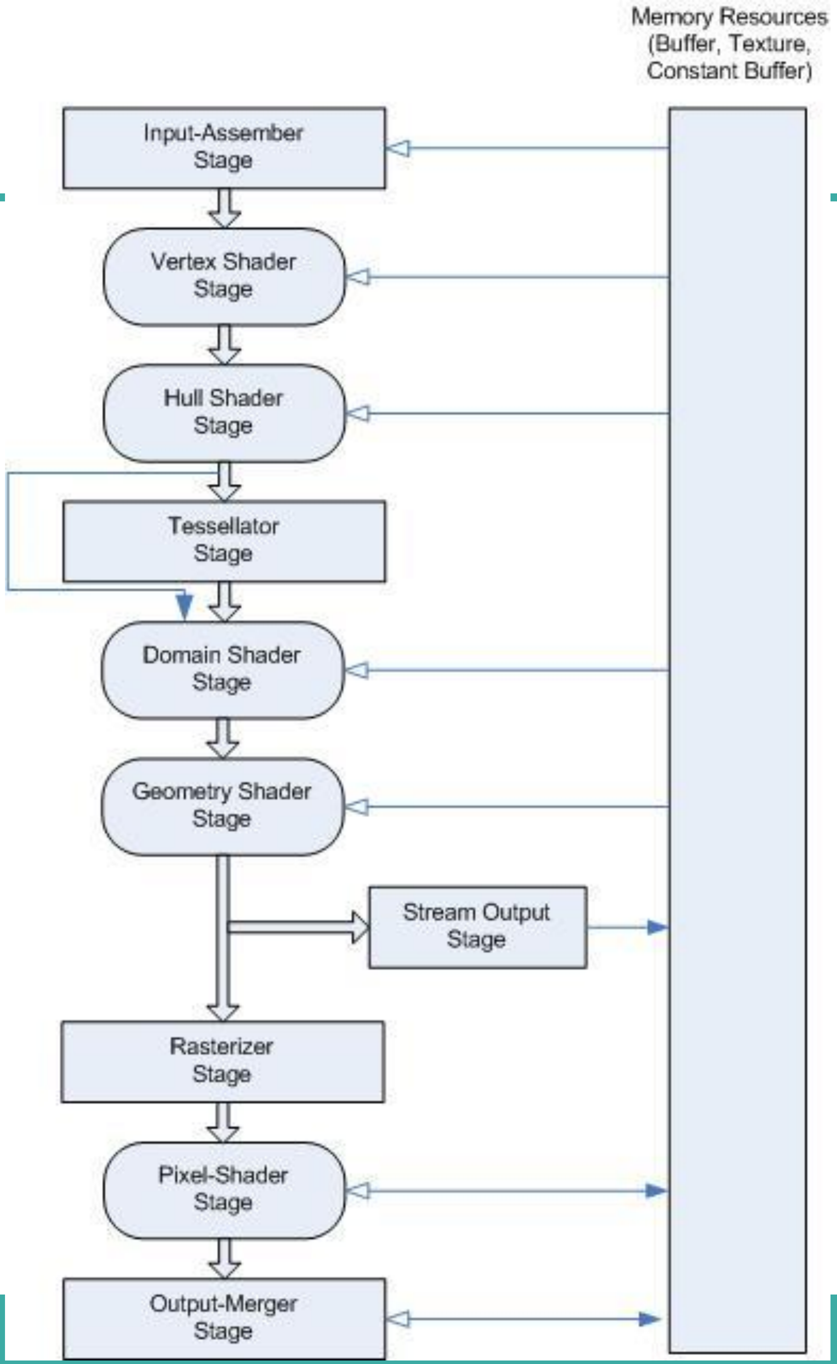
- Separate programs that run on the GPU
- Expose the stages of the rendering pipeline
- Allowing us to reprogram the rendering pipeline
  - Moving us away from the fixed function pipeline
  - To the programmable pipeline
- Written in the High Level Shading Language (HLSL)
  - A C like programming language
  - Also GLSL OpenGL Shading Language

Types of shaders

- Vertex shader
  - Vertex operations
- Hull shader
- Domain shader
  - Both Hull and Domain combine with the tessellator to control mesh subdivision
- Geometry shader
  - Manipulates groups of vertices controlling primitive geometry. Can generate geometry
- Pixel / fragement shader
  - Decides the final colour output value for pixels
- Compute shader
  - Performs general purpose (parallel) computing on the GPU

The Direct3D 11 Pipeline

- Input-Assembler Stage
  - Supplies data to the pipeline (triangles, lines, points)
- Rasteriser stage
  - Generation of fragment data from geometric data
  - Additionally the rasteriser produces a depth value for each fragment
- Output Merger Stage
  - Combines various types of output data: pixel shader values, depth and stencil buffer data to generate the final result



Vertex Shader

- A vertex shader is the point in the pipeline where **you** are given control over every vertex (position, normals, etc)
  - Last year with the fixed-function pipeline it had a built-in set of functionality to process them
- A vertex shader always takes a single input
  - a vertex
  - and produces a single output vertex
- Typically, transform the vertex into screen space
- Additional processing is possible
  - Manipulate vertex values
  - Generate additional data

Fragment/pixel shader

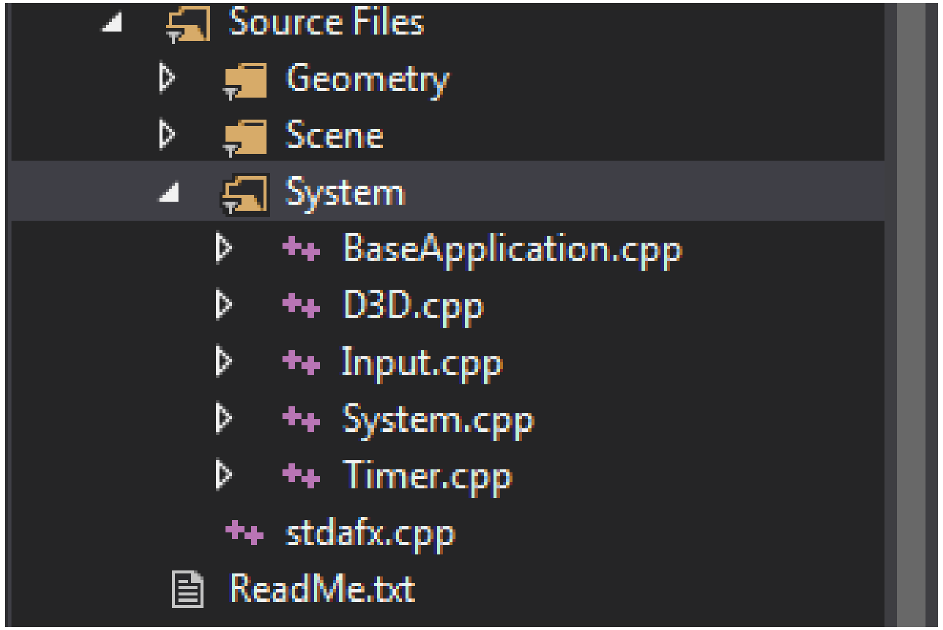
- Pixel shaders give **you** access to every pixel of the final render
- Before anything is drawn you are given the chance to make changes to the colour of each pixel
  - Including:
    - Changing the colour
    - Selecting colour based on a texture
    - Making it transparent
    - Combining colours and manipulating them for special effects
- Most commonly you will apply lighting or texture to the pixel colour
- Like the vertex shader you will deal with one pixel at a time
- We will discuss the other shaders in more detail as we use them

The framework

- I will provide a framework we will extend
- Why?
  - Don’t have time to build our own
  - Already does camera, input, setup window etc
- Gives you the opportunity to practice working with an existing project/code base
- Consists of
  - A static library containing most of the base code you will need
  - The DirectXTK (tool kit) for shader and image loading
  - The imGUI toolkit for GUI rendering and processing
  - An example project using the libraries to render a triangle
- Note: the framework underwent re-factoring over the summer
  - There may be some typos/wrong variable names. I think I got them all.

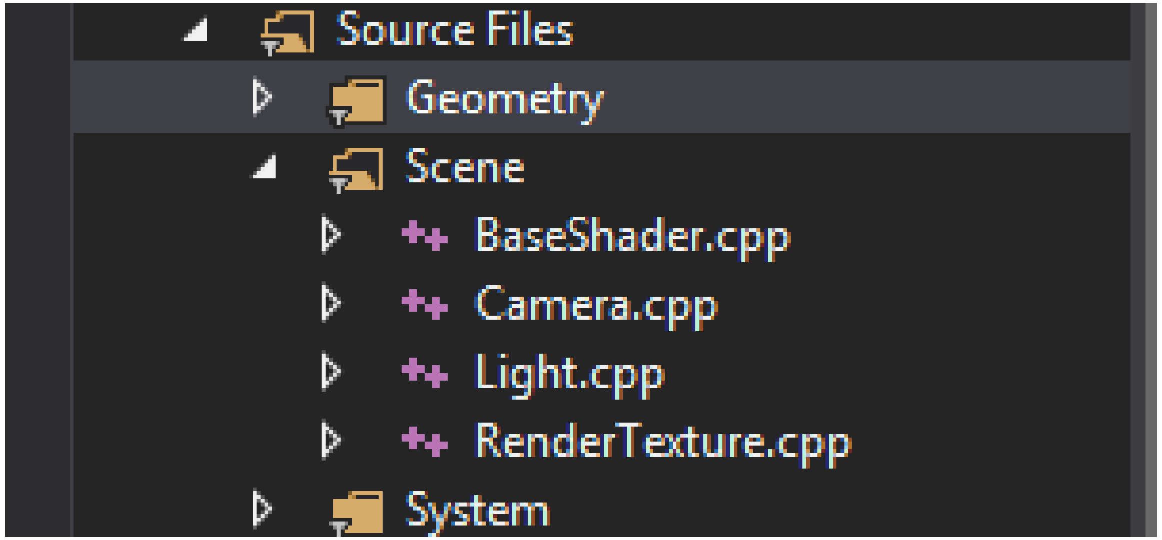
The framework - System

- What classes are already provided for you
- System
  - System
    - Creates window
    - Handle gaming loop
  - Base Application
    - Setup of our scene
    - Some stuff already setup
    - For each lab we will inherit from this class
  - D3D class
    - Configure and initialises D3D
  - Input class
    - Handles keyboard and mouse input
  - Timer class
    - Calculates frame time



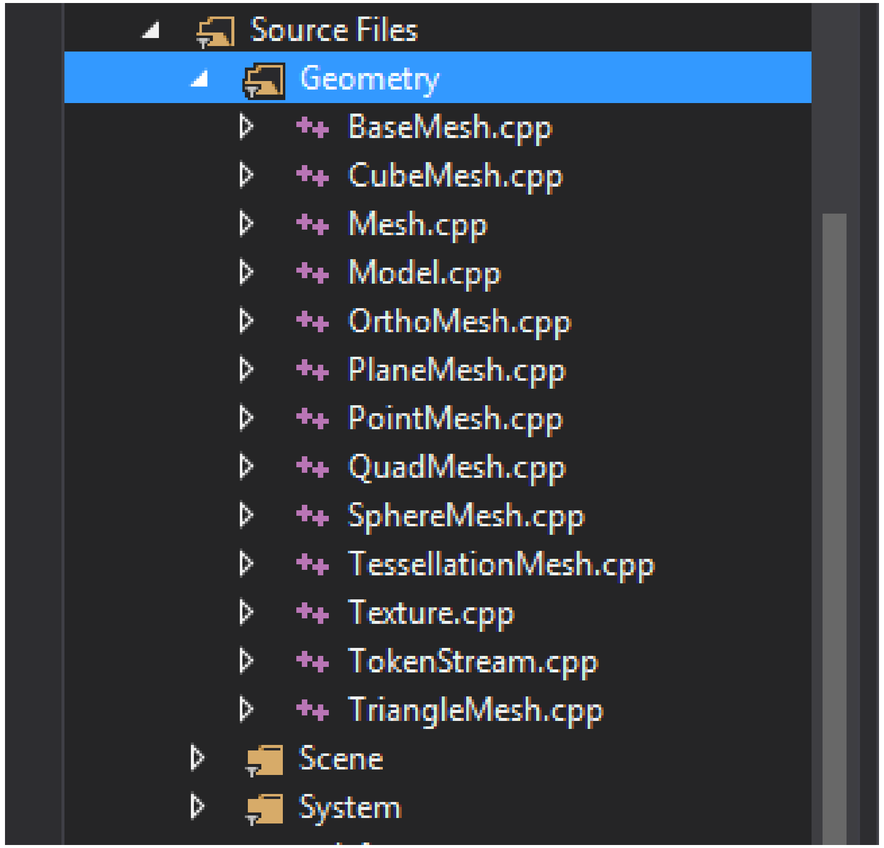
The framework - Scene

- BaseShader
  - Functionality for loading, compiling and rendering with shaders
- Camera class
  - Stores position and rotation of camera
  - Generates the view matrix
- Light
  - Store data for light source
- RenderTexture
  - A texture resource we can render to



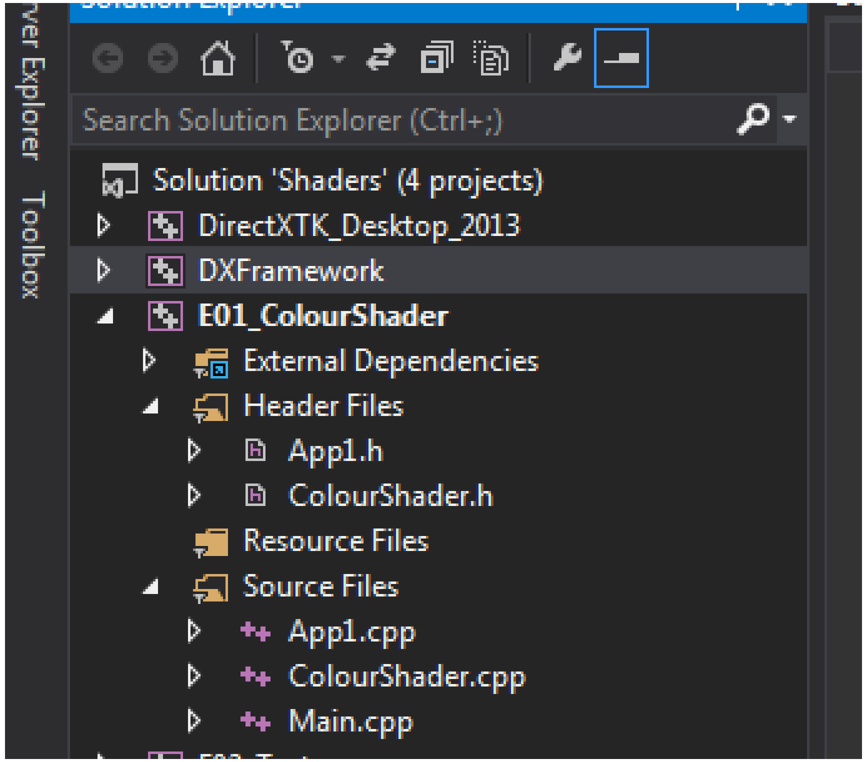
The framework - Geometry

- Base Mesh
    - Base functions for meshes
    - Other mesh classes inherit this parent class
- Texture class
    - Loads and stores a texture
- A collection of meshes, including
    - Spheres, planes and model loading



Example project

- Additional project, Colour shader
- Will use the framework to render a simple coloured triangle with a simple set of vertex and pixel/fragment shaders
- Main
  - Entry point
- App1
  - Our application
  - Initialises shaders, geometry
  - Contains the render function
- ColourShader (handler class)
  - Loads shaders
  - Pass data to the shader(s) / GPU



Example project

- Colour shader class
  - Loads our first shader
- Accompanying shader files
  - Colour_vs.hlsl
  - Colour_ps.hlsl
- Our first set of shaders will render a coloured triangle
- Minimum shaders we need
  - A vertex shader
  - A pixel/fragment shader
  - Other shaders are only required as needed

Main

- Won’t change a whole lot week to week

```cpp
// Main.cpp
#include "../DXFramework/System.h"
#include "App1.h"

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, PSTR pScmdline, int iCmdshow)
{
    App1* app = new App1();
    System* system;

    // Create the system object.
    system = new System(app);

    // Initialize and run the system object.
    system->run();

    // Shutdown and release the system object.
    delete system;
    system = 0;

    return 0;
}
```

My first shaders

- Application needs to
  - Initialise triangle mesh
  - Initialise colour shader handler
- By inheriting from BaseApplication we can call the parents init()
  - Required as it initialises (default) objects including input, camera, timer etc

```
void App1::init(HINSTANCE hinstance, HWND hwnd, int screenWidth, int screenHeight, Input *in)
{
    // Call super init function (required!)
    BaseApplication::init(hinstance, hwnd, screenWidth, screenHeight, in);

    // Create Mesh object
    mesh = new TriangleMesh(renderer->getDevice(), renderer->getDeviceContext(), L"../res/DefaultDiffuse.png");

    colourShader = new ColourShader(renderer->getDevice(), hwnd);

}
```

Deconstructor

```
App1::~App1()
{
    // Run base application deconstructor
    BaseApplication::~BaseApplication();

    // Release the Direct3D object.
    if (mesh)
    {
        delete mesh;
        mesh = 0;
    }

    if (colourShader)
    {
        delete colourShader;
        colourShader = 0;
    }
}
```

Frame / update()

- Frame() is called by the System object during the game loop
- Is responsible for per frame updates
  - Increment variables
    - If you are rotating something etc
  - User input
- Basic user input and camera is handled by the BaseApplication
- Ends by calling the render function

Frame

```cpp
bool App1::frame()
{
	bool result;

	result = BaseApplication::frame();
	if (!result)
	{
		return false;
	}

	// Render the graphics.
	result = render();
	if (!result)
	{
		return false;
	}

	return true;
}
```

My first shaders

- Application render ()
  - Begin scene, setting background colour
  - Calculate view matrix, get world and projection matrices
  - Push Mesh data onto graphics card
  - Set shader parameters
    - Pass required data to GPU
    - In this case matrix data
  - Call render for the colour shader
  - End scene

Render

```
bool App1::render()
{
    XMMATRIX worldMatrix, viewMatrix, projectionMatrix;

    //// Clear the scene. (default blue colour)
    renderer->beginScene(0.39f, 0.58f, 0.92f, 1.0f);

    //// Generate the view matrix based on the camera's position.
    camera->update();

    //// Get the world, view, projection, and ortho matrices from the camera and Direct3D objects.
    worldMatrix = renderer->getWorldMatrix();
    viewMatrix = camera->getViewMatrix();
    projectionMatrix = renderer->getProjectionMatrix();

    //// Send geometry data (from mesh)
    mesh->sendData(renderer->getDeviceContext());
    //// Set shader parameters (matrices and texture)
    colourShader->setShaderParameters(renderer->getDeviceContext(), worldMatrix, viewMatrix, projectionMatrix);
    //// Render object (combination of mesh geometry and shader process
    colourShader->render(renderer->getDeviceContext(), mesh->getIndexCount());

    // Render GUI
    gui();

    //// Present the rendered scene to the screen.
    renderer->endScene();

    return true;
}
```

Mesh

- All meshes will use vertex and index buffers
  - These function similar to vertex arrays we covered last year
- Vertex buffer have a user-defined type
  - This type struct will describe what data is stored in the buffer
    - Vertices, normals, texture coordinates, colour etc
- Index buffer is an order list of indices
  - Order to render the vertex buffer
- I have provided a collection of pre-made meshes
  - Have a look at them

Mesh struct

```
struct VertexType
{
	XMFLOAT3 position;
	XMFLOAT2 texture;
	XMFLOAT3 normal;
};
```

Mesh construction

```
vertices = new VertexType[vertexCount];
indices = new unsigned long[indexCount];

// Load the vertex array with data.
vertices[0].position = XMFLOAT3(0.0f, 1.0f, 0.0f);  // Top.
vertices[0].texture = XMFLOAT2(0.0f, 1.0f);
vertices[0].normal = XMFLOAT3(0.0f, 0.0f, -1.0f);

vertices[1].position = XMFLOAT3(-1.0f, 0.0f, 0.0f);  // Bottom left.
vertices[1].texture = XMFLOAT2(0.0f, 0.0f);
vertices[1].normal = XMFLOAT3(0.0f, 0.0f, -1.0f);

vertices[2].position = XMFLOAT3(1.0f, 0.0f, 0.0f);  // Bottom right.
vertices[2].texture = XMFLOAT2(1.0f, 0.0f);
vertices[2].normal = XMFLOAT3(0.0f, 0.0f, -1.0f);

// Load the index array with data.
indices[0] = 0;  // Top/
indices[1] = 1;  // Bottom left.
indices[2] = 2;  // Bottom right.
```

Colour Shader object

- Handles are shader files
- Responsible for rendering the model and invoking our shader programs
  - Will load and configure our shaders
  - Will pass data to the shaders/GPU
- Two methods for processing shaders
  - Compile at build-time or run-time
  - The framework compiles shaders at build time. Visual studio will return any compilation errors
  - The compiled shaders need loaded at run-time

Colour_vs.hlsl

```
// colour vertex shader
// Simple geometry pass
// texture coordinates and normals will be ignored.

cbuffer MatrixBuffer : register(cb0)
{
    matrix worldMatrix;
    matrix viewMatrix;
    matrix projectionMatrix;
};

struct InputType
{
    float4 position : POSITION;
    float2 tex : TEXCOORD0;
    float3 normal : NORMAL;
};

struct OutputType
{
    float4 position : SV_POSITION;
    float2 tex : TEXCOORD0;
    float3 normal : NORMAL;
};
```

Colour_vs.hlsl

```
OutputType main(InputType input)
{
    OutputType output;


    // Change the position vector to be 4 units for proper matrix calculations.
    input.position.w = 1.0f;

    // Calculate the position of the vertex against the world, view, and projection matrices.
    output.position = mul(input.position, worldMatrix);
    output.position = mul(output.position, viewMatrix);
    output.position = mul(output.position, projectionMatrix);

    // Store the texture coordinates for the pixel shader.
    output.tex = input.tex;

    // Calculate the normal vector against the world matrix only.
    output.normal = mul(input.normal, (float3x3)worldMatrix);

    // Normalize the normal vector.
    output.normal = normalize(output.normal);

    return output;
}
```

Colour_ps.hlsl

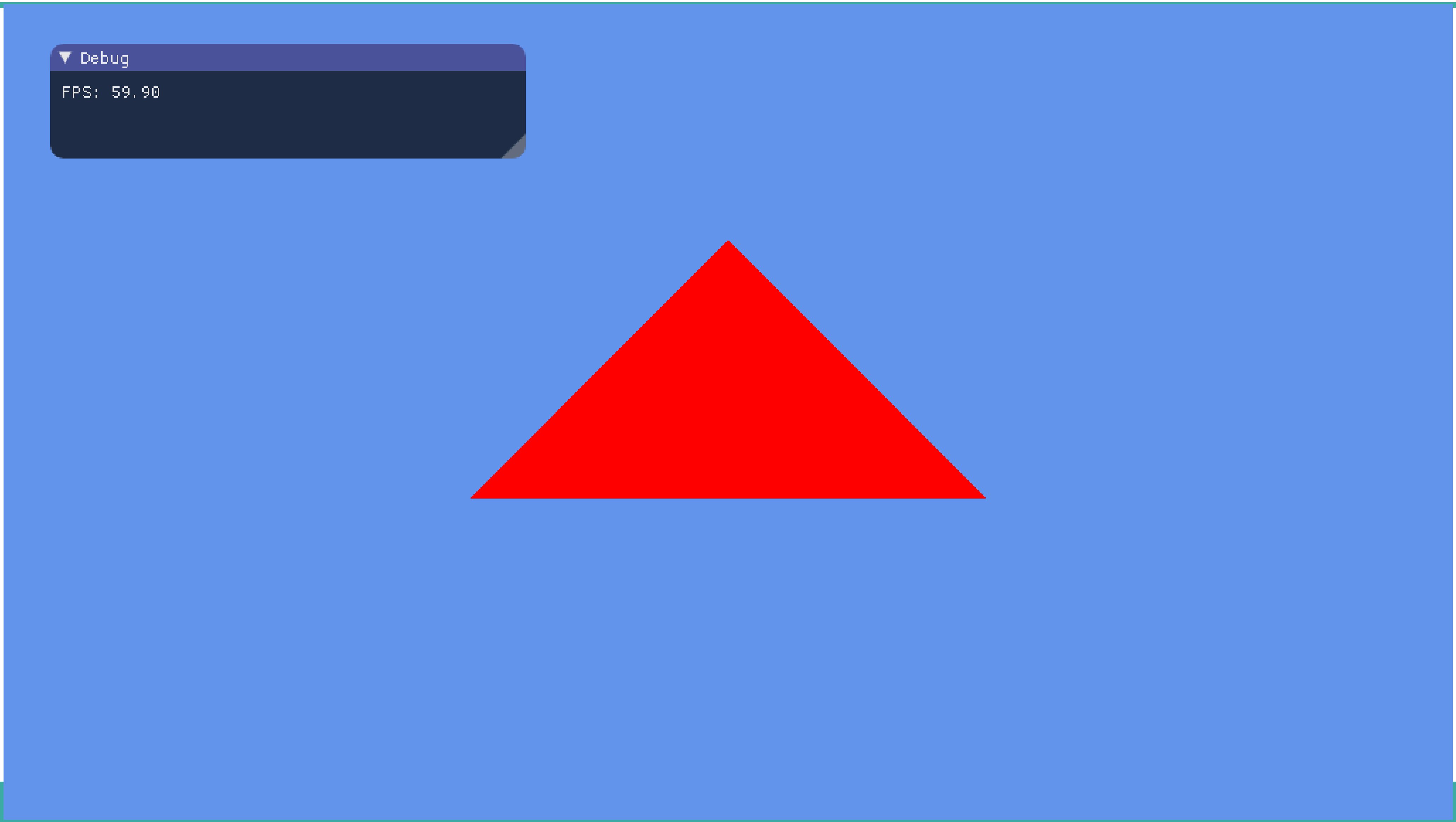
```
// Colour pixel/fragment shader
// Basic fragment shader outputting a colour

struct InputType
{
    float4 position : SV_POSITION;
    float2 tex : TEXCOORD0;
    float3 normal : NORMAL;
};


float4 main(InputType input) : SV_TARGET
{
    float4 colour = float4(1.0, 0.0, 0.0, 1.0);

    return colour;
}
```

Result



The labs

- We will be setting up our framework and working with simple shaders
- The framework will provide a good base for other labs
- Instead of copying the whole framework for each lab
  - You can create a new Lab Application project within the solution
- Lab sheets will consist of two sections
  - Lab tasks
  - Research tasks/questions
    - Requiring you to do some research your own
- Recommended reading
  - https://www.3dgep.com/introduction-to-directx-11/#DirectX_11_Pipeline

Resources

- HLSL Reference site
- http://msdn.microsoft.com/en-us/library/windows/desktop/ff471376(v=vs.85).aspx

- New compiler functions differ previous versions
- Look up table
- http://blogs.msdn.com/b/chuckw/archive/2013/08/21/living-without-d3dx.aspx