How to build a Graphics Engine in Testable Steps

1) Get the “BootStrap” Project from the portal. This contains a core library and two example projects that can be run. Set Example3D as the main project, and run it. You should see some 3D shapes rendered by the AIE Gizmo library.
2) Add a moving camera to the Example3D. You should now be able to move the camera around the scene easily and naturally
3) Add a procedural mesh that doesn’t use the Gizmo library to the Example3D code. This is a big step that requires a lot of fiddly set up. When you’re finished, you should be able to see your own mesh in the scene.
4) Load a model, and feed its data to the OpenGL system so we can see the model outline
5) Feed the model normals to OpenGL and apply basic diffuse lighting.
6) Add specular lighting.
7) Apply some basic texturing using a planar mapping.
8) Feed the model UVs to OpenGL and apply diffuse texturing.
9) Create a Model class that encapsulates the loading and rendering of a model. Render two copies of the same model and a third copy of a different one.
10) Add FBX Loading to the Model class.
11) Add Animated Models
12) Create a Scene and Instance system. Render a large array of a single model.
13) Set up a Framebuffer class and use it to make a virtual TV screen
14) Use the Framebuffer for a post-processing effect
15) Use the Framebuffer for deferred rendering

1. Bootstrap

Bootstrap can be found on the portal page for **Subject Resources**. Download it (and the fix if you like) and get it compiling.

2. Camera

Example3D starts with a fixed camera. We're going to add a camera that does the following.

W,S – Move Forward and Back

A,D – pan left and right

Z,X – pan up and down

Right mouse hold and mouse move – rotate the camera angle around, with no dutch tilt.

We can express our camera direction as a quaternion, but it's much easier to think in polar coordinates. Consider two angles, theta and phi. Theta is the longitude angle – if you sit in a swivel chair and spin, while keeping your gaze horizontal, you're changing theta. Theta is zero when you're looking along the x-axis.

Phi is the angle of elevation. If you're looking horizontally, phi is zero. Looking down, phi becomes negative. Looking up, phi becomes positive.

In this model there is no yaw. You can't tilt your head from side to side.

The forwards vector of this camera is:

glm::vec3 forward = (cos(phi)*cos(theta), sin(phi), cos(phi)*sin(theta));

Let's unpack this.

When phi = 0 the y term becomes zero, and the cos(phi) becomes 1, so the forward vector is

forward = (cos(theta), 0, sin(theta)

This is a circle in the x,z plane. As we increase phi, the y component grows and the horizontal circle shrinks as it travels up the unit sphere.

For any value of theta and phi, the forward vector has length of 1 exactly.

Have a look at the Application3D code.

This happens in the startup:

```
// create simple camera transforms
m_viewMatrix = glm::lookAt(vec3(10), vec3(0), vec3(0, 1, 0));
m_projectionMatrix = glm::perspective(glm::pi<float>() * 0.25f,
                getWindowWidth() / (float)getWindowHeight(),
                0.1f, 1000.f);
```

And in the draw function we use these values (and recalculate projection in case the window resizes):

```
m_projectionMatrix = glm::perspective(glm::pi<float>() * 0.25f,
                getWindowWidth() / (float)getWindowHeight(),
```

```
                        0.1f, 1000.f);

        Gizmos::draw(m_projectionMatrix * m_viewMatrix);
```

We want to replace these matrices with ones derived from our camera class.

So the basic camera class should look like this:

```
class Camera
{
public:
        Camera() : theta(0), phi(-20), position(-10,4,0) {}

        glm::mat4 GetProjectionMatrix(float w, float h);
        glm::mat4 GetViewMatrix();

private:
        float theta;
        float phi;
        glm::vec3 position;
};
```

We can calculate the view and projection matrices using modified versions of the hardcoded equations in the Application3D

Theta and Phi are in degrees for ease of human interpretation, so we have to convert them to radians.

The glm::lookAt function is a very handy utility. We pass where the camera is, and a point its looking at (and an up vector) to get a view matrix. Here, we've calculated the point we're looking at as the camera position plus the forwards vector as described above.

The view matrix provided by glm::lookAt is the inverse of a transform made up from these vectors. We calculate a forward by subtracting the look-at point from the position. We then get a right vector as the unit cross product of the forward with the up vector we provide. Finally we get an up vector as the cross product between the right and forwards vectors.

Finally, Inverting this gives us the world-to-view matrix we require, and which glm::lookAt returns.

glm::perspective does all the arcane maths described in the Camera lecture on portal.

```
const float deg2Rad = 3.14159f/180.0f;

glm::mat4 Camera::GetProjectionMatrix(float w, float h)
{
        return glm::perspective(glm::pi<float>() * 0.25f,
                                                w/h,
                                                0.1f, 1000.f);
}
```

```
glm::mat4 Camera::GetViewMatrix()
{
	float thetaR = theta * deg2Rad;
	float phiR = phi* deg2Rad;
	glm::vec3 forward(cos(phiR)*cos(thetaR), sin(phiR), cos(phiR)*sin(thetaR));
	return glm::lookAt(position, position + forward, glm::vec3(0, 1, 0));
}
```

We can use these in Application3D draw and test them now before we go further. Add a member camera

```
	Camera camera;
```

And change the code in draw to:

```
void Application3D::draw() {

	// wipe the screen to the background colour
	clearScreen();

	// update perspective in case window resized
	m_projectionMatrix = camera.GetProjectionMatrix(getWindowWidth(),
getWindowHeight());
	m_viewMatrix = camera.GetViewMatrix();
```

and run. With the values for theta, phi and position we put in the constructor for camera we should be above the plane, on the negative x-axis, looking dwon the x-axis (theta = 0) and slightly down (phi = -20).

We're now ready to start moving the camera.

Add an Update function to camera, and call it in Application3D::update. Here's the code for panning the camera.

```
void Camera::Update()
{
	aie::Input* input = aie::Input::getInstance();
	float thetaR = theta * deg2Rad;
	float phiR = phi* deg2Rad;

	//calculate the forwards and right axes and up axis for the camera
	glm::vec3 forward(cos(phiR)*cos(thetaR), sin(phiR), cos(phiR)*sin(thetaR));
	glm::vec3 right(-sin(thetaR), 0, cos(thetaR));
	glm::vec3 up(0, 1, 0);

	float deltaTime = 0.1f;

	// use WASD, ZX keys to move camera around
	if (input->isKeyDown(aie::INPUT_KEY_X))
		position += up * deltaTime;

	if (input->isKeyDown(aie::INPUT_KEY_Z))
		position += -up * deltaTime;

	if (input->isKeyDown(aie::INPUT_KEY_A))
		position += -right* deltaTime;
```

```
if (input->isKeyDown(aie::INPUT_KEY_D))
	position += right* deltaTime;

if (input->isKeyDown(aie::INPUT_KEY_W))
	position += forward* deltaTime;

if (input->isKeyDown(aie::INPUT_KEY_S))
	position += -forward* deltaTime;
```

Run this and test it. You should now be able to pan the camera around using the keyboard. If nothing happens, check that you're calling camera.Update from Application3D::Update.

Finally, we'll add rotation using the mouse. When the right mouse is held down, we want to increment theta and phi by the x and y changes in the mouse position each frame. To do this, we need to store if the last frame's mouse coordinates. So add these members to the Camera:

```
int lastMouseX;
int lastMouseY;
```

And add this code to the update

```
// get the current mouse coordinates
float mx = input->getMouseX();
float my = input->getMouseY();
// if the right button is down, increment theta and phi
if (input->wasMouseButtonPressed(aie::INPUT_MOUSE_BUTTON_RIGHT))
{
		theta += 0.05f * (mx - lastMouseX);
		phi -= 0.05f * (my - lastMouseY);

}
// store this frames values for next frame
lastMouseX = mx;
lastMouseY = my;
```

And that's a basic fly camera! You can neaten up the code to store forwards if you like, or add tilt, or clamp the values for phi at +_/-70 to avoid Gimbal lock at the poles, or do whatever you like to it.

3. Rendering a Procedural Mesh

3.1 Setting up Geometry

The **Rendering Geometry** tutorial on portal covers the basics of how to render a procedural mesh in OpenGL. Much of what we set up here is useable for rendering models later.

This is a large and fiddly step, requiring everything to be right order to work, and introduces us to the way that OpenGL works.

OpenGL works with GLuint's, which are just unsigned ints. Typically, you pass in a block of data, and are returned a integer to store and identify that block of data with in the future. Think of it as a cloakroom ticket system. Once you pass data into the graphics card you're given a ticket (the integer) but you can't see or interact with that data again.

OpenGL can feel clunky, but remember its written to be useable from many languages, including non-OO ones. Its our job as engine programmers to wrap it up into something nicer.

Work your way through that tutorial first.

Here's the lines that actually draw the mesh.

```
glBindVertexArray(m_VAO);
unsigned int indexCount = (rows - 1) * (columns - 1) * 6;
glDrawElements(GL_TRIANGLES, indexCount, GL_UNSIGNED_INT, 0);
```

Although we set up three integers to hold the data, we only use the m_VAO, and not the m_VBO and m_IBO integers. This is because the VAO is a "master ticket" that internally refers to the the other two.

We also have to pass in how many indices we're using – this could be stored as a member.

I'll now step through the OpenGL in my working GenerateGrid function, which creates these tickets, and describe what it does line by line in comments.

```
// function to create a grid
void Application3D::GenerateGrid(unsigned int rows, unsigned int cols)
{
	//generate the data as covered in tutorial on Portal.
…
	//OpenGL code
	//create a Vertex Array Object and store a ticket for it.
	glGenVertexArrays(1, &m_VAO);
	// create vertex and index buffers, and store.
	glGenBuffers(1, &m_VBO);
	glGenBuffers(1, &m_IBO);

	// set the current Vertex Array Object to the one we created a few lines ago.
	glBindVertexArray(m_VAO);

	// set the current Vertex Buffer Object to the one we created earlier. This also
sets this VBO as the one associated with the current VAO.
	glBindBuffer(GL_ARRAY_BUFFER, m_VBO);
	// feed the columns array to it
	glBufferData(GL_ARRAY_BUFFER, (rows * cols) * sizeof(Vertex), aoVertices,
GL_STATIC_DRAW);
	// turn on slots 0 and 1 in this Vertex Buffer Object
	glEnableVertexAttribArray(0);
	glEnableVertexAttribArray(1);
	// make slot 0 for the current VBO start at the beginning of the data, with each
vertex separated in memory by the size of a Vertex structure.
	// this separataion between vertices, or stride, is the fifth argument passed in.
	glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, sizeof(Vertex), 0);
	// make slot 1 for the current VBO start one vec4 into the data, with each vertex
separated in memory by the size of a Vertex structure.
	// This gives us a "striped" data block which goes vertex, colour, vertex, colour,
since that's the way our data is laid out.
	glVertexAttribPointer(1, 4, GL_FLOAT, GL_FALSE, sizeof(Vertex),
(void*)(sizeof(vec4)));

	// set the current index buffer to the one we created earlier, and also make this
the index buffer for the current VAO
	glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, m_IBO);
	// feed in our array of indices
	glBufferData(GL_ELEMENT_ARRAY_BUFFER, (rows - 1) * (cols - 1) * 6 *
	sizeof(unsigned int), auiIndices, GL_STATIC_DRAW);

	// set the current VAO to zero now, we're done setting up our data
	glBindVertexArray(0);
	// set the current VBO and IBO to zero. CAREFUL! If you do this before setting the
current VAO to zero, it
	// messes with the VAO and sets its VBO and IBO to zero again.
	glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
	glBindBuffer(GL_ARRAY_BUFFER, 0);

	// we've now got our vertex data copied into graphics card memory and have a
ticket for it.
	// we can delete the CPU side memory if we no longer need it.
	delete[] aoVertices;
}
```

This function describes how we set up a block of ready-to-eat data for the graphics card, pass it in, and store a ticket to use when we want to render it every frame.

3.2 Shaders

For the rendering to work, we also need shaders. The shaders in the tutorial are created from string variables set up in code and compiled in the startup function. Let's take a quick look at that code.

```
	// shaders as hard-coded strings, not shown here.
	const char* vsSource = …
	const char* fsSource = …

	int success = GL_FALSE;
	// create a new vertex and fragement shader, and get tickets for them
	unsigned int vertexShader = glCreateShader(GL_VERTEX_SHADER);
	unsigned int fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
	// set the source for the vertex shader to our code string above, and compile
	glShaderSource(vertexShader, 1, (const char**)&vsSource, 0);
	glCompileShader(vertexShader);
	// and the same for the fragement shader
	glShaderSource(fragmentShader, 1, (const char**)&fsSource, 0);
	glCompileShader(fragmentShader);
	// create a new shader pipeline and store the ticket for it
	m_programID = glCreateProgram();
	// attache our vertex and fragmenrt shaders to it - they both know where thewy fit
in the pipeline due to the flags passed in on their creation
	glAttachShader(m_programID, vertexShader);
	glAttachShader(m_programID, fragmentShader);
	// link the whole pipeline
	glLinkProgram(m_programID);
	// get the error code and report it if there was a problem
	glGetProgramiv(m_programID, GL_LINK_STATUS, &success);
	if (success == GL_FALSE)
	{
		…
	}
	// free memory for the compiled components, we only need the whole linked pipeline
	glDeleteShader(fragmentShader);
	glDeleteShader(vertexShader);
```

This m_programID , which we store for the whole pipeline, is used ion drawing here.

```
	glUseProgram(m_programID);
```

which just sets the current rendering pipeline for all subsequent draw calls.

So we write shader code as strings, compile it at startup, and get a ticket for our compiled and linked shader code. We then use this ticket before rendering.

This is a good start. I'll provide some utility code for loading shader code from text files later, which will make the whole process easier to use.

3.3 Shader code

Let's have a look at the shader code. This is the vertex shader.

```
version 410
layout(location=0) in vec4 position;
layout(location=1) in vec4 colour;
out vec4 vColour;
uniform mat4 MVP;

void main()
{
    vColour = colour;
    gl_Position = MVP * position;
}
```

It starts with a version number, this is OpenGL 4.10

The layout location lines correspond to the slots we referred to in setting up the geometry. Slot zero is the per-vertex position in model space. Slot 1 is the vertex colour. Compare with these lines in GenerateGrid.

```
      glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, sizeof(Vertex), 0);
      glVertexAttribPointer(1, 4, GL_FLOAT, GL_FALSE, sizeof(Vertex),
(void*)(sizeof(vec4)));
```

The 0 and 1 as the first arguments to glVertxAttribPointer correspond to the location=0 and location=1 in the shader code.

Next in the shader code we have an out variable vColour, which will be assigned to in here and passed to the fragment shader.

We then have a uniform, which is a variable that gets passed in from the CPU code before we call draw, as explained later. This one is a matrix, called MVP, which stands for the Model-View-Perspective matrix, ie the composite transformation that takes us from model space to clip space, where OpenGL does its drawing.

Finally there's a main function (like in C), which gets called on every vertex.

vColour (the out variable) gets assigned to colour (the in variable). So we're taking the per-vertex colour and passing it through to the fragment shader.

gl_Position is a special variable, whioch must be set to give the clip space position of the vertex for drawing. Here we take the in variable of the model space vertex, and transform it with the MVP.

Now lets have a look at the fragment shader.

```
version 410
in vec4 vColour;
out vec4 fragColor;

void main()
{
    fragColour = vColour;
}
```

Our in variable, vColour must match the name of the out variable from our vertex shader.

We pass out a single vec4 which will be the final pixel colour. In our main function we just assign this value.

Together, this is about as simple as a shader pipeline can be. Transform the incoming vertices into clip space and render them with a flat colour.

3.4 Uniforms

Uniforms are variables that can be passed from CPU code to the shader code on the graphics card. The one uniform we immediately need is the MVP matrix. Our processed geometry (referenced by m_VAO) is in world space.

We pass the uniform value in like so in our drawing code:

```
        glm::mat4 mvp = m_projectionMatrix * m_viewMatrix;

…
        // get a ticket for the address of a named uniform in our shader code
        unsigned int projectionViewUniform = glGetUniformLocation(m_programID, "MVP");
        // pass a matrix through to that address
        glUniformMatrix4fv(projectionViewUniform, 1, GL_FALSE, (float*)&mvp);
```

Why is this called a “uniform”? The answer is that this value is the same for every vertex. The big block of vertex data referenced by m_VAO is different for each vertex, but uniforms are the same for every vertex in a drawing call.

OpenGL has a number of different functions for passing in floats, matrices, vectors, integers and so on as uniforms. We’ll really push this when we get to animated models.

And that’s how you render geometry! Once all this works, you’ll see a flat plane with some colour gradients on it for all your effort. But rest assured, we’ve laid some very solid foundations here.

4. Rendering a model from a file

Start with the **Tutorial – Using the TinyObjLoader** tutorial from the **Rendering Geometry** page on Portal.

Grab the **Stanford OBJ** file from **Subject Resources** on portal and save the contents into a folder called Data inside the bin directory.

4.1 Loading the model Data

Add these variables to the Application3D class.

```
std::vector<tinyobj::shape_t> shapes;
std::vector<tinyobj::material_t> materials;
```

Make a function in Application3D called LoadOBJMode like so.

```
void Application3D::LoadObjModel(const char* name)
{
	std::string err;
	tinyobj::LoadObj(shapes, materials, err, name);
}
```

And call it from Application3D::start like this

```
LoadObjModel("data\\Bunny.obj");
```

Check that this works by stepping through the code and examining shapes after the load model call. In a debug build, that load line should take a couple of seconds to run. Examining shapes in the debugger will show, why – the model has over 100,000 vertices!

These Standford models are very high res 3D scans of museum pieces.

Grab the CreateBuffers function from the portal tutorial – it's very similar to the code we embedded in GenerateGrid earlier.

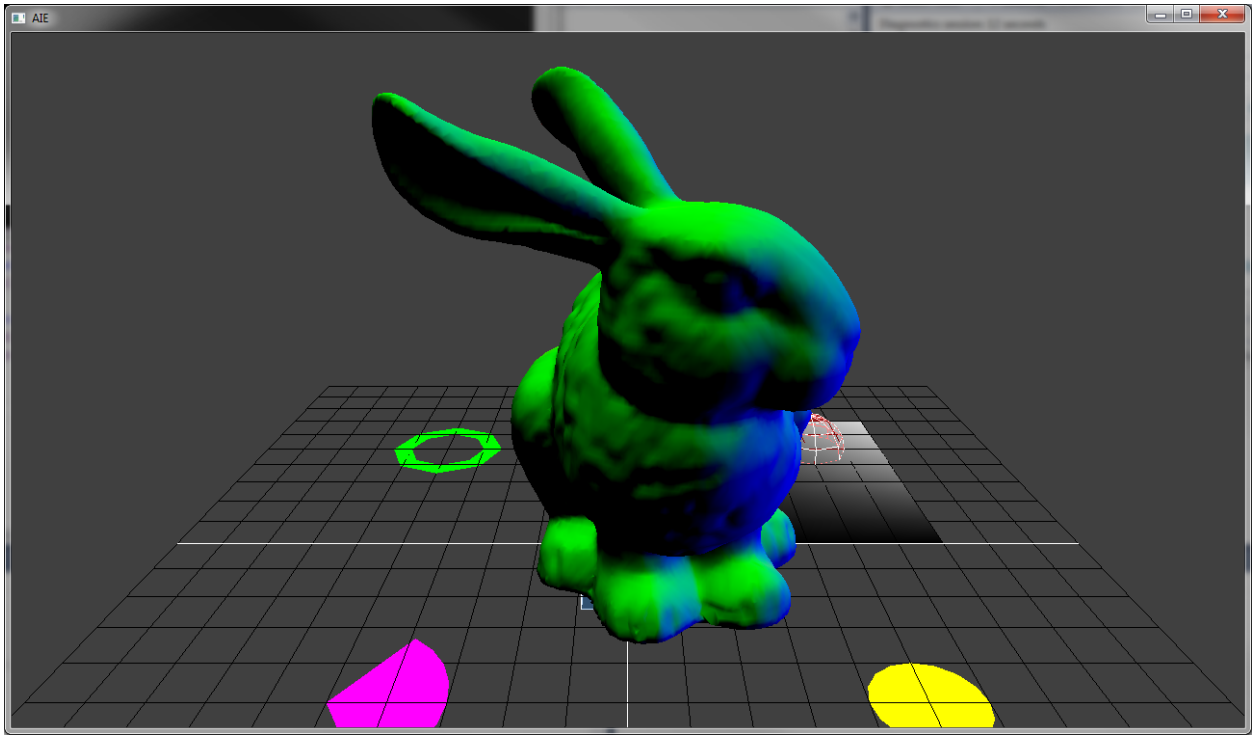
Make this a member function and add the following member to Application3D:

```
struct GLInfo
{
	unsigned int m_VAO;
	unsigned int m_VBO;
	unsigned int m_IBO;
	unsigned int m_index_count;
};

std::vector<GLInfo> m_gl_info;
```

And call CreateBuffers from LoadObjModel.

Add the drawing code from the Rendering Geometry tutorial to the end of Application::Draw, and you should be able to run and see a very brightly coloured rabbit, after moving the camera around!



What's going on here?

Let's have a look at CreateBuffers.

Our previous code fed in two slots worth of per-vertex data. 0 was the positions, and 1 was the vertex colours. In CreateBuffers we have the following:

```
glEnableVertexAttribArray(0); //position
glEnableVertexAttribArray(1); //normal data
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_TRUE, 0,
            (void*)(sizeof(float)*shapes[mesh_index].mesh.positions.size()));
```

The first slot is being filled with positions as before, the second is being filled with normals instead of vertex colours. The x,y and z values are being interpreted as RGB values. If y is greater than zero, for example, so the face is facing upwards, the green value gets increased. Face pointing straight up are very green. Faces fgacing along the z axis are very blue (on the right).
Can you tell me without looking what colour the rabbit's bottom is?

This happy little accident reveals a very powerful trick in graphics programming. If you want to see a particular value inside shader code (where you can't debug it) – try feeding that variable to the RGB channels!

5. Adding simple lighting

We'll need to modify the shaders to turn the normals as vertex colours that we see into diffuse lighting. At this point, it's a good idea to refactor our shader system to use text files rather than hgardcoded strings. I've done this for you, since there are far more exciting ways to spend your time than writing a text file loader.

Go to \\adlfs1\shares\Shared\Students\Games Programming Year 2\OpenGL\Code Snippets\ and copy Shader.cpp and Shader.h to the project and add them in Visual Studio.

Grab the shaders folder and stick it in your bin folder to get the shaders we've used so far.

We can now modify our Application code to read the shaders from file. Add an include to Shaders.h to Application.cpp and replace the shader loading code in star with this one line:

```
m_programID = Shader::CompileShaders("shaders\\BasicVertexShader.txt",
"shaders\\BasicFragmentShader.txt");
```

Check that this still works – you should still see the technicolor rabbit.

We can now edit our shader code for the fragment shader. Here's a really simple diffuse shader that takes the dot product of the normal (passed in as vColour here) with a fixed light direction and uses the dot product to create a greyscale colour.

```
#version 410
in vec4 vColour;
out vec4 fragColor;
uniform float slider2;

void main()
{
	// hardcode a light direction
	vec4 light = vec4(0.5,0.7,0.5,0);

	// get the intensity
	float intensity = clamp(dot(light, vColour), 0, 1);

	fragColor = vec4(intensity, intensity, intensity, 1);
}
```

Add a small amount, say 0.2 to intensity to get ambient light.

```
float intensity = clamp(dot(light, vColour), 0, 1) + 0.1;
```

We're getting a decent visual effect here, but what are we doing wrong? The normal we pass in is in model space, and the light vector that we're dotting it with is in world space. If we were to rotate the rabbit, we'd see the light move with the rabbit. Let's set that up to illustrate this problem, and then fix it.

Change the line where we calculate mvp in Application3D::draw to look like this:

```
modelMatrix = glm::rotate(0.7f, vec3(0, 1, 0));
glm::mat4 mvp = m_projectionMatrix * m_viewMatrix * modelMatrix;
Gizmos::draw(m_projectionMatrix * m_viewMatrix);
```

We're now adding a model matrix, which we define as a twist around the y axis of 0.7 radians (around 45 degrees). Let's animate this quickly.

```
static float angle = 0;
angle += 0.01f;
modelMatrix = glm::rotate(angle, vec3(0, 1, 0));
glm::mat4 mvp = m_projectionMatrix * m_viewMatrix * modelMatrix;
Gizmos::draw(m_projectionMatrix * m_viewMatrix);
```

The rabbit rotates but appears to be taking the light with him.

We need to multiply our normal by this model matrix inside the shaders, to turn them into normals in world space.

We'll pass this in as another uniform, the matrix M (and rename the variables to better represent what we're doing here). We ensure that the w component of the normal is zero by using the expression vec4(normal.xyz, 0) because the normal is a direction rather than a position, and thus we don't want it to be affected by the translation component in the M matrix.

```
#version 410
layout(location=0) in vec4 position;
layout(location=1) in vec4 normal;
out vec4 worldNormal;

uniform mat4 MVP;
uniform mat4 M;

void main()
{
    worldNormal = M * vec4(normal.xyz, 0);
    gl_Position = MVP * position;
}
```

```
#version 410
in vec4 worldNormal;
out vec4 fragColor;
uniform float slider2;

void main()
{
	// hardcode a light direction
	vec4 light = vec4(0.5,0.7,0.5,0);

	// get the intensity
	float intensity = clamp(dot(light, worldNormal), 0, 1) + 0.1;

	fragColor = vec4(intensity, intensity, intensity, 1);
}
```

And we'll need to pass the model matrix in as a uniform from Application3D::draw()

```
unsigned int modelUniform = glGetUniformLocation(m_programID, "M");
glUniformMatrix4fv(modelUniform, 1, GL_FALSE, (float*)&modelMatrix);
```

If all that worked correctly, the rabbit should now rotate under a fixed light.

If it didn't, check the Shader Debugging Checklist in \\adlfs1\shares\Shared\Students\Games Programming Year 2\OpenGL and try to figure out where things have gone wrong.

(Even if your rabbit did work, take ten minutes to read this document).

6. Adding specular lighting

For specular lighting we need the camera position too.

We take a ray from the camera to the world position, reflect it in the normal, and then dot it with the world light direction.

This requires us to pass the camera position in as a uniform. We can specify the uniform in the fragment shader, where we need it, instead of in the vertex shader. Note – don't put uniforms with the same name in both shaders as this becomes confusing very quickly.

```
#version 410
in vec4 worldNormal;
in vec4 worldPosition;

out vec4 fragColor;
uniform float slider2;
uniform vec4 cameraPosition;

void main()
{
	// hardcode a light direction
	vec4 light = vec4(0.5,0.7,0.5,0);

	// get the diffuse and ambient intensity
	float intensity = clamp(dot(light, worldNormal), 0, 1) + 0.1;

	// get the unit vector from position to camera
	vec4 toCamera = normalize(worldPosition - cameraPosition);
	vec4 refl = reflect(toCamera, worldNormal);
	float specular = clamp(dot(refl, light), 0, 1);
	// raise to the power of 4 for tighter specualr highlights
	specular = specular *specular;
	specular = specular *specular;

	// final colour is magenta diffuse/ambient with white specular for a plastic look
    fragColor = vec4(intensity+specular, specular, intensity + specular, 1);
}
```

We pass the uniform in via Application3d::Draw as before, anywhere between glUseProgam and glDrawElements

```
	unsigned int camUniform = glGetUniformLocation(m_programID, "cameraPosition");
	glUniform4f(camUniform, camera.GetPos().x, camera.GetPos().y, camera.GetPos().z,
1);
```

7. Adding textures

The Standford models such as the bunny don't have any UV coordinates, since they were laser-scanned from museum pieces.

We can work around this for now and demonstrate texture loading by setting up UV coordinates based on world coordinates within the shader, called planar mapping.

Go to the **Textures** tutorial on portal, and follow the steps there to load a texture into memory, send it to the graphics card and receive a ticket, and then pass that ticket to your fragment shader via a uniform.

You'll need to:

1) Add an m_texture variable to the Application
2) Write a local function LoadTexture that returns a uint, given a filename
3) Call this and assign to m_texture in startup
4) Set the texture in your draw code using glActiveTexture and glBindTexture, and then pass slot 0 to a new uniform
5) Modify your shader to use the texture function to modulate the diffuse/ambient light

Here's my fragment shader with diffuse texture capabilities added.

```
#version 410
in vec4 worldNormal;
in vec4 worldPosition;

out vec4 fragColor;
uniform float slider2;
uniform vec4 cameraPosition;
uniform sampler2D diffuse;

void main()
{
	// hardcode a light direction
	vec4 light = vec4(0.5,0.7,0.5,0);

	// get the diffuse and ambient intensity
	float intensity = clamp(dot(light, worldNormal), 0, 1) + 0.1;

	// get the unit vector from position to camera
	vec4 toCamera = normalize(worldPosition - cameraPosition);
	vec4 refl = reflect(toCamera, worldNormal);
	float specular = clamp(dot(refl, light), 0, 1);
	// raise to the power of 4 for tighter specular highlights
	specular = specular *specular;
	specular = specular *specular;

	vec2 uv = worldPosition.xy;
	vec4 texColor = texture(diffuse, uv);

	// final colour is magenta diffuse/ambient with white specular for a plastic look
	fragColor = texColor*intensity + vec4(specular, specular, specular, 1);
```

}

8. Applying model UVs

We now need to load a model that has UV data, and pass that data to the fragment shader to replace the planar mapping that we're using.

We need to pass the uv data from CreateBuffers to our vertex shader first.

We need to attach the uv's on to the large data array we're passing to the graphics card, and set up a third channel (slot 2) to identify where they are.

We modify CreateBuffers like so:

```
		glGenVertexArrays(1, &m_gl_info[mesh_index].m_VAO);
		glGenBuffers(1, &m_gl_info[mesh_index].m_VBO);
		glGenBuffers(1, &m_gl_info[mesh_index].m_IBO);
		glBindVertexArray(m_gl_info[mesh_index].m_VAO);
		unsigned int float_count = shapes[mesh_index].mesh.positions.size();
		float_count += shapes[mesh_index].mesh.normals.size();
		float_count += shapes[mesh_index].mesh.texcoords.size();
		std::vector<float> vertex_data;
		vertex_data.reserve(float_count);
		vertex_data.insert(vertex_data.end(),
			shapes[mesh_index].mesh.positions.begin(),
			shapes[mesh_index].mesh.positions.end());
		vertex_data.insert(vertex_data.end(),
			shapes[mesh_index].mesh.normals.begin(),
			shapes[mesh_index].mesh.normals.end());
		vertex_data.insert(vertex_data.end(),
			shapes[mesh_index].mesh.texcoords.begin(),
			shapes[mesh_index].mesh.texcoords.end());
		m_gl_info[mesh_index].m_index_count =
			shapes[mesh_index].mesh.indices.size();
		glBindBuffer(GL_ARRAY_BUFFER, m_gl_info[mesh_index].m_VBO);
		glBufferData(GL_ARRAY_BUFFER,
			vertex_data.size() * sizeof(float),
			vertex_data.data(), GL_STATIC_DRAW);
		glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, m_gl_info[mesh_index].m_IBO);
		glBufferData(GL_ELEMENT_ARRAY_BUFFER,
			shapes[mesh_index].mesh.indices.size() * sizeof(unsigned int),
			shapes[mesh_index].mesh.indices.data(), GL_STATIC_DRAW);
		glEnableVertexAttribArray(0); //position
		glEnableVertexAttribArray(1); //normal data
		glEnableVertexAttribArray(2); //uvs
		glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
		glVertexAttribPointer(1, 3, GL_FLOAT, GL_TRUE, 0,
			(void*)(sizeof(float)*shapes[mesh_index].mesh.positions.size()));
		glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 0,
			(void*)(sizeof(float)*(shapes[mesh_index].mesh.positions.size()+
shapes[mesh_index].mesh.normals.size())));
		glBindVertexArray(0);
		glBindBuffer(GL_ARRAY_BUFFER, 0);
		glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
```

Note that the call to VertexAttribPointer has a 2 as the second argument, denoting the amount of data per element here is 2 floats, not 3 like like the position and normal, since we only pass xy for the texture coordinates, not xyz.

The vertex shader needs to receive the uvs into channel 2 now, and pass them through to the fragment shader as an out variable.

```
#version 410
layout(location=0) in vec4 position;
layout(location=1) in vec4 normal;
layout(location=2) in vec2 uvs;
out vec4 worldNormal;
out vec4 worldPosition;
out vec2 uv;

uniform mat4 MVP;
uniform mat4 M;

void main()
{
    worldNormal = M * normal;
       worldPosition = M * position;
    gl_Position = MVP * position;
       uv = uvs;
}
```

And the fragment shader picks them up as an in variable with the same name, and uses them.

Comment out the planar mapping line, or delete it.

```
#version 410
in vec4 worldNormal;
in vec4 worldPosition;
in vec2 uv;

…

       //vec2 uv = worldPosition.xy;
       vec4 texColor = texture(diffuse, uv);

…
}
```

Do all this and you should get a grey bunny with white highlights, because the bunny has no UV data. (Check using the debugger after loading if you like)

Grab the **Simple Shapes FBX & OBJ** file from the **Subject Resources** on portal and copy these contents into your bin folder, and load cube.obj up instead. You should see the texture applied to the cube properly now. Check with the sphere too.

9. Creating a Model Class

It's time to encapsulate this code and create a model class so we can load different obj files and render them simultaneously. Our aim here is to be able to load both the rabbit and the cube, and draw multiple copies of each.

Here's a start for how we want our model class to look.

We could create a Model member for each mesh we want to draw in our Application, call Load in the startup function, and then draw each of them in the draw function.

```
class Model
{
public:
	bool Load(const char* filename);
	void Draw(glm::mat4 transform);

}
```

We'll need to move some members from the Application to the Model class, so that we can have multiple models loaded, and move some functionality to Model. Specifically, these functions and variables can go into Model.

```
void CreateBuffers();

std::vector<tinyobj::shape_t> shapes;
std::vector<tinyobj::material_t> materials;

struct GLInfo
{
	unsigned int m_VAO;
	unsigned int m_VBO;
	unsigned int m_IBO;
	unsigned int m_index_count;
};

std::vector<GLInfo> m_gl_info;
```

The code from Our global function LoadObjModel can be moved to Model::Load. Code from Application::Draw can be moved to Model::Draw. We can restructure our Application to have a member Model

```
Model bunny;
```

In startup we replace LoadModelObj with

```
bunny.Load("data\\bunny.obj");
```

And move the following drawing code to Model::Draw and call that from Application3D:Draw. After all this, the application should still draw the bunny as before.

```
void Model::Draw(glm::mat4 transform)
{
	for (unsigned int i = 0; i < m_gl_info.size(); ++i) {
		glBindVertexArray(m_gl_info[i].m_VAO);
		glDrawElements(GL_TRIANGLES, m_gl_info[i].m_index_count,
			GL_UNSIGNED_INT, 0);
	}
}
```

We now need to consider how to pass the individual transform into each Model::Draw call and make use of it.

The transform is passed in via the “M” and “MVP” uniforms. We need to the m_programID to access the uniforms. We can do this for now by passing in the cameraMatrix (for constructing MVP) and the programID as follows:

```
void Model::Draw(glm::mat4 transform, glm::mat4 cameraMatrix, unsigned int programID)
{
	glm::mat4 mvp = cameraMatrix * transform;

	unsigned int projectionViewUniform = glGetUniformLocation(programID, "MVP");
	glUniformMatrix4fv(projectionViewUniform, 1, GL_FALSE, (float*)&mvp);

	unsigned int modelUniform = glGetUniformLocation(programID, "M");
	glUniformMatrix4fv(modelUniform, 1, GL_FALSE, (float*)&transform);


	for (unsigned int i = 0; i < m_gl_info.size(); ++i) {
		glBindVertexArray(m_gl_info[i].m_VAO);
		glDrawElements(GL_TRIANGLES, m_gl_info[i].m_index_count,
			GL_UNSIGNED_INT, 0);
	}
}
```

We’ll neaten this up later as we architect the Graphics engine.

Here’s some test code of mine from Application::Draw that uses this Model::Draw to draw a bunny at the origin and a sphere moving up and down the x axis (Sphere is another Model member of the Application, loaded with the sphere mesh in Startup()

```
	glm::mat4 cameraMatrix = m_projectionMatrix * m_viewMatrix;

	…

	bunny.Draw(modelMatrix, cameraMatrix, m_programID);

	static float x = -5;
	x += 0.1f;
	if (x > 5)
		x -= 10;
	modelMatrix = glm::translate(vec3(x, 0, 0)) * modelMatrix;
	sphere.Draw(modelMatrix, cameraMatrix, m_programID);
```

10. Loading FBX Files

OBJ files are a relatively simple format that allows us to load a simple polygonal mesh. The FBX file format is more versatile, and most importantly allows us to load animated models.

We'll need to set up a bit of groundwork here to allow us to load FBX files.

The tutorial **Using the FBX Loader** in **Rendering Geometry** is great reference for this task – read through it now.

Copy the FBXLoader folder from \\adlfs1\shares\Shared\Students\Games Programming Year 2\OpenGL\dep to the dependencies folder in your project.

You'll need to link the FBXLoader to your project

1. In the Linker->Input Tab, add FBXLoader_d.lib for debug builds, or FBXLoader.lib for release builds, to the Additional Dependencies list.
2. In Linker->General add the path for the FBXLoader folder, e.g. $(ProjectDir)..\dependencies\FBXLoader to Additional Library Directories
3. You'll need to comment out #define STB_IMAGE_IMPLEMENTATION from your code. The FBXLoader is already built on the STB library functions and includes them.

Add a FBXFile pointer member to the Model class

```
FBXFile* fbxFile;
```

Whether we load an OBJ file or FBX file, the end result should be the same – that we have a model with a set of valid GLInfo structures that can be used to draw it. The intermediate steps will be different. For an OBj file we load the data into the shapes and materials arrays and then turn them into OpenGL data in CreateBuffers. We'll need a different version of CreateBuffers for FBX files, one that converts data from the FBXFile structure.

Rename CreateBuffers to CreateBuffersOBJ, and add a CreateBuffersFBX, and modify the code in Model::Load to allow for loading of different file types.

```
bool Model::Load(const char* filename)
{
	if (strstr(filename, ".obj") != NULL)
	{
		std::string err;
		tinyobj::LoadObj(shapes, materials, err, filename);
		CreateBuffersOBJ();
		return true;
	}

	if (strstr(filename, ".fbx") != NULL)
	{
		// load the FBX file as per tutorial
		CreateBuffersFBX();
		return true;
	}

	return false;
```

```
}
```

The code for loading the FBX file, and for creating OpenGLBuffers are both in the tutorial. We need to add a few lines to CreateBuffersFBX to work with our Model class. Before the loop, resize the glInfo array:

```
m_gl_info.resize(fbxFile->getMeshCount());
```

and at the bottom of the loop, write into each element of this array

```
m_gl_info[i].m_VAO = glData[0];
m_gl_info[i].m_VBO = glData[1];
m_gl_info[i].m_IBO = glData[2];
m_gl_info[i].m_index_count = mesh->m_indices.size();
```

Note that the CreateBuffers code from the tutorial writes out positions to slot 0 and normal to slot 1, which should work with our existing shaders.

If you replace the load of sphere.obj with sphere.fbx it should now work, but you'll lose the texture on your sphere. This is because we're no longer feeding UV coordinates to our shader.

We can fix this by adding the necessary calls to CreateBuffersFBX.

```
glEnableVertexAttribArray(0); // position
        glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE,
                sizeof(FBXVertex), 0);
        glEnableVertexAttribArray(1); // normal
        glVertexAttribPointer(1, 4, GL_FLOAT, GL_TRUE,
                sizeof(FBXVertex),
                ((char*)0) + FBXVertex::NormalOffset);
        glEnableVertexAttribArray(2); // UVs
        glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE,
                sizeof(FBXVertex),
                ((char*)0) + FBXVertex::TexCoord1Offset);
        glBindVertexArray(0);
```

Lets just compare the two CreateBuffers functions.

In CreateBuffersOBJ we create a large temporary array containing the positions, then the normal, then the UVs. Think of this as Neopolitan Icecream. Each call to glVertexAttribArry has 0 for the stride (the fifth argument), saying that the offset between each vertex's data is packed as tightly as possible. Each call then has an offset (the sixth and final argument) indicating where in the data buffer it starts. The normal, for example, start after the positions, the UVs after the positions and normals.

In CreateBuffersFBX, each call has a non-zero stride, specifically the sizeof a FBXVertex structure.

The data here is stored as a single array of FBXVertex structures. We use a set of defines such as FBXVertex::NormalOffset to indicate where each one starts.

The following two diagrams show this different layouts for Positions (Chocolate), Normals (Vanilla) and UVs (Strawberry) for the two file formats.

FBX File Data

Position 1
Colour 1
Normal 1
Tangent 1
Binormal 1
Indices 1
Weights 1
UV1 1
UV2 1
Position 2
Colour 2
Normal 2
Tangent 2
Binormal 2
Indices 2
Weights 2
UV1 2
UV2 2
…
Position N
Colour N
Normal N
Tangent N
Binormal N
Indices N
Weights N
UV1 N
UV2 N

OBJ File data

Position 1
Position 2
…
Position N
Normal 1
Normal 2
…
Normal N
UV 1
UV 2
…
UV N

11. Animation

We're now ready to set up animated models.

The tutorial in the **Animation** page of portal covers the basics here, and you can find some animated character models there.

Unzip the character models and replace the bunny with the fbx model. It should load the character in their rest pose. Note that these characters are massive, and you may want to scale them down a bit like so.

```
	// draw a scaled down model
	buddha.Draw(glm::scale(vec3(0.001f,0.001f,0.001f))*modelMatrix, cameraMatrix,
m_programID);
```

Having scaled them down, they may look unlit. This is because the worldNormals we're calculating in the vertex shader are being scaled down too. We can fix this by normalizing them in the vertex shader:

```
void main()
{
	worldNormal = normalize(M * vec4(normal.xyz, 0));
	worldPosition = M * position;
	gl_Position = MVP * position;
	uv = uvs;
}
```

Read the tutorial. We can summarise our task list to get the character animated like so:

1. Pass through extra per-vertex data from the model, such as bone weights and indices. (Ignore tangents for now) We'll only want to do this for animated models, so we need a way to flag this in our Model class.
2. Create a new Vertex Shader that takes these values and applies them, along with transformation matrices for the current bones. This vertex shader should output the same information to the fragment shader as our existing vertex shader.
3. Compile this as a separate shader pipeline, and use appropriately.
4. Use the code from the tutorial to send bone transforms to the shader every frame for animated models

First off, we should determine if a model is animated when we load it. This function will do the trick:

```
	bool isAnimated() { return fbxFile && fbxFile->getSkeletonCount() > 0; }
```

Only FBX files can be animated, and they have a function for checking how many active skeletons there are in the model.

If a model is animated, we need to pass the bone weights and indices through when we create the OpenGL buffers.

```
	glEnableVertexAttribArray(2); // UVs
		glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, sizeof(FBXVertex),
((char*)0) + FBXVertex::TexCoord1Offset);
		if (isAnimated())
		{
			glEnableVertexAttribArray(3); // weights
			glVertexAttribPointer(3, 4, GL_FLOAT, GL_FALSE, sizeof(FBXVertex),
((char*)0) + FBXVertex::WeightsOffset);
			glEnableVertexAttribArray(4); // indices
			glVertexAttribPointer(4, 4, GL_FLOAT, GL_FALSE, sizeof(FBXVertex),
((char*)0) + FBXVertex::IndicesOffset);
		}
		glBindVertexArray(0);
```

We then create a copy of our vertex shader and pick up these new buffers and apply them to the position we calculate, similar to the tutorial. Instead of using the position as it is to get worldPosition and gl_Position, we're calculated a model space position P based on the local bone transforms and the weights for the current vertex.

```
#version 410
layout(location=0) in vec4 position;
layout(location=1) in vec4 normal;
layout(location=2) in vec2 uvs;
layout(location=3) in vec4 weights;
layout(location=4) in vec4 indices;

out vec4 worldNormal;
out vec4 worldPosition;
out vec2 uv;

uniform mat4 MVP;
uniform mat4 M;

// we need to give our bone array a limit
const int MAX_BONES = 128;
uniform mat4 bones[MAX_BONES];

void main()
{
	// cast the indices to integer's so they can index an array
	ivec4 index = ivec4(indices);

	// sample bones and blend up to 4
	vec4 P = bones[ index.x ] * position * weights.x;
	P += bones[ index.y ] * position * weights.y;
	P += bones[ index.z ] * position * weights.z;
	P += bones[ index.w ] * position * weights.w;

	worldPosition = M * P;
	gl_Position = MVP * P;

	worldNormal = normalize(M * vec4(normal.xyz, 0));
```

```
	uv = uvs;
}
```

To use this new shader, we need to add a second programID to our application and set it up in startup to use this new vertex shader

```
	m_programID = Shader::CompileShaders("shaders\\BasicVertexShader.txt",
"shaders\\BasicFragmentShader.txt");
	m_animProgramID = Shader::CompileShaders("shaders\\AnimVertexShader.txt",
"shaders\\BasicFragmentShader.txt");
```

And pass this through in the draw calls for any animated models we draw.

In Model::Draw we update uniforms for the matrices. We can update the bone transform uniforms here too, if the model is animated, using the code from the tutorial.

```
void Model::Draw(glm::mat4 transform, glm::mat4 cameraMatrix, unsigned int programID)
{
	glm::mat4 mvp = cameraMatrix * transform;

	// get a ticket for the address of a named uniform in our shader code
	unsigned int projectionViewUniform = glGetUniformLocation(programID, "MVP");
	// pass a matrix through to that address
	glUniformMatrix4fv(projectionViewUniform, 1, GL_FALSE, (float*)&mvp);

	unsigned int modelUniform = glGetUniformLocation(programID, "M");
	glUniformMatrix4fv(modelUniform, 1, GL_FALSE, (float*)&transform);

	if (isAnimated())
	{
		// grab the skeleton and animation we want to use
		FBXSkeleton* skeleton = fbxFile->getSkeletonByIndex(0);
		skeleton->updateBones();
		int bones_location = glGetUniformLocation(programID, "bones");
		glUniformMatrix4fv(bones_location, skeleton->m_boneCount, GL_FALSE,
			(float*)skeleton->m_bones);
	}

	for (unsigned int i = 0; i < m_gl_info.size(); ++i) {
		glBindVertexArray(m_gl_info[i].m_VAO);
		glDrawElements(GL_TRIANGLES, m_gl_info[i].m_index_count,
			GL_UNSIGNED_INT, 0);
	}
}
```

We can add the code from the tutorial for updating the skeleton into a new update function:

```
void Model::Update(float timer)
{
	if (isAnimated())
	{
		// grab the skeleton and animation we want to use
		FBXSkeleton* skeleton = fbxFile->getSkeletonByIndex(0);
		FBXAnimation* animation = fbxFile->getAnimationByIndex(0);

		// evaluate the animation to update bones
```

```
			skeleton->evaluate(animation, timer);
			for (unsigned int bone_index = 0; bone_index < skeleton->m_boneCount;
++bone_index)
			{
				skeleton->m_nodes[bone_index]->updateGlobalTransform();
			}
		}
}
```

Finally, we need to call these from Application::Draw and make sure that the correct shader pipleine is being used.

```
		glUseProgram(m_animProgramID);
		camUniform = glGetUniformLocation(m_animProgramID, "cameraPosition");
		glUniform4f(camUniform, camera.GetPos().x, camera.GetPos().y, camera.GetPos().z,
1);

		// set texture slot 0 to use the texture we created earlier
		glActiveTexture(GL_TEXTURE0);
		glBindTexture(GL_TEXTURE_2D, m_texture);
		// tell the shader where it is - use slot 0
		loc = glGetUniformLocation(m_animProgramID, "diffuse");
		glUniform1i(loc, 0);

		// temporary code to provide an animation timer
		static float animTimer = 0;
		animTimer += 0.1f;
		buddha.Update(animTimer);

		// draw a scaled down model
		buddha.Draw(glm::scale(vec3(0.001f, 0.001f, 0.001f))*modelMatrix, cameraMatrix,
m_animProgramID);
```

The important thing is that all uniforms get set (including the Model::Update call) after the current pipeline has been set using glUseProgram, and before the call to Model::Draw.

The lighting on animated characters is not quite there yet. We need to transform the normal from the base pose into the correct bone space the same way we did for the positions.

```
		vec4 norm = vec4(normal.xyz, 0);
		vec4 N = bones[ index.x ] * norm * weights.x;
		N += bones[ index.y ] * norm * weights.y;
		N += bones[ index.z ] * norm * weights.z;
		N += bones[ index.w ] * norm * weights.w;

		worldNormal = normalize(M * N);
```

12. Architecting the Graphics Engine

We've now got code for loading animated and static models, texturing them, and drawing them with diffuse and specular lighting. We have a Model, Camera and Shader classes so far, but too much code is in the Application still.

Its time to start moving this code into a better OO framework.

We'll need the following classes.

Instance

An Instance is like a Unity GameObject. It contains a transform matrix, the model used, the texture for the model, the rendering pipeline to use, and so on. Many Instances can point to the same model and draw themselves in different positions.

If the model is animated, the Instance would hold the current animation and timer info too.

Instance::Draw would call Model::Draw with the correct info from the Scene.

Here's a very useful snippet from my Instance class. I'm not great at working out transformation matrices from raw numbers, so I thought I'd specify them as position, Euler angles and scale, just like Unity does. These get set in the constructor (or from other programme code), and the transform is calculated from them every frame.

```
	glm::vec3 m_pos;
	glm::vec3 m_euler;
	glm::vec3 m_scale;

	glm::mat4 m_transform;
```

These get turned into a transform matrix every frame with this function

```
void Instance::UpdateTransform()
{
	float rad = 6.28f / 360.0f;
	m_transform = glm::translate(m_pos)
		* glm::rotate(m_euler.z * rad, vec3(0, 0, 1))
		* glm::rotate(m_euler.y * rad, vec3(0, 1, 0))
		* glm::rotate(m_euler.x * rad, vec3(1, 0, 0))
		* glm::scale(m_scale);
}
```

Scene

Contains a collection of Instances, Cameras and Lights.

A Scene can draw itself by calling Draw on all Instances. Example code from the .h file:

```
void Draw();
std::vector<Instance*> m_instances;
Camera camera;
std::vec3 m_lightDir;
glm::vec3 m_pointLights[4];
glm::vec3 m_pointLightColours[4];
float m_pointLightPowers[4];
```

Texture

A utility class for loading textures and referencing them.

Model's can have default textures that get selected if nothing else is specified, but Instances should be able to override these with custom textures if desired.

All classes should have good constructors so we can set up scenes in code something like this, where we can specify each instance's position and orientation.

We've already got a Texture class, either the Bootstrap one or the simple one I gave out.

Shader

The Shader class with static functions for loading and compiling shaders from file can be extended to have a member id to store the compiled program handle.

It can help to have utility functions like SetVector(const char*name, vec3 value) and these can contain error handling.

Shader->Use() can call glUseProgram with the shader's id. This has to be done before any setting of uniforms, so probably call this first in Instance::Draw

(For a more optimal engine, sort Instances per shader and only call Shader::Use and set the uniforms for lights and cameras at the start of this list.)

How to Draw a Scene: shaders and uniforms

For everything to render corretcly, we need to call the following in the right order:

1. glUseProgram - set the current shader
2. Set all uniforms. This includes M and MVP matrixes, camera positions, lighting information and textures
3. The call to DrawElements or DrawArray

Uniforms set before we call glUseProgram get wiped when we reload the shader program. Uniforms set after the draw call are obviously too late and have no effect on the draw call.

Consider where the information for setting all uniforms is stored:

M matrix – this is the Instance transform.

MVP – the instance transform and the Scene's camera

Camera Position and lighting information – stored in the Scene.

Since different Instances may use different shaders, we'll call glUseProgam at the start of Instance::Draw as a first pass.

This is sub-optimal (the shader pipeline has to be reloaded per obejct), but it's robust and functional, which are more important considerations for now. I'll leave it to each of you to work out how to optimise this if you like.

After this we'll want to set all the non-instance specific uniforms associated with the scene, ie the camera position and light data. We'll need a pointer to the Scene.

In Model::Draw we pass through a transform matrix, so it makes sense to pass the Scene through here too, so we can calculate MVP from the camera and transform data.

Here's some skeleton code to help you lay all this out.

The Scene iterates over all Instances, and passes itself into the Draw calls so the Instances can access its camera and light data.

```
void Scene::Draw()
{
	for (int i = 0; i < m_instances.size(); i++)
	{
		m_instances[i].Draw(this);
	}
}
```

The Instances call a Scene function to set all the camera and light scenes for the shader. They then set their textures up and call Model::Draw, passing in their own transform and shader, and the camera matrix.

```
void Instance::Draw(Scene* scene)
{
	scene->UseShader(m_shader);
	shader->SetTexture("diffuse", 0, m_texture);
	m_model->Draw(m_transform, scene->GetCameraMatrix(), shader);
}
```

Scene::UseShader looks like this, and activates the given shader and sets some uniforms on it:

```
void Scene::UseShader(Shader* shader)
{
	unsigned int id = shader->GetID();
	int loc;
	glUseProgram(id);
	loc = glGetUniformLocation(id, "cameraPosition");
	glUniform3f(loc, m_camera->GetPos().x, m_camera->GetPos().y, m_camera-
>GetPos().z);
	loc = glGetUniformLocation(id, "lightDir");
	glUniform3f(loc, m_lightDir.x, m_lightDir.y, m_lightDir.z);
}
```

Model::Draw does this – sets the model transform dependent uniforms, processes some extra data for animated models and then does the draw call.

```
void Model::Draw(glm::mat4 transform, glm::mat4 cameraMatrix, Shader* shader)
{
	glm::mat4 mvp = cameraMatrix * transform;

	shader->SetMatrix("MVP", mvp);
	shader->SetMatrix("M", transform);

	if (isAnimated())
	{

		// grab the skeleton and animation we want to use
```

```
            FBXSkeleton* skeleton = fbxFile->getSkeletonByIndex(0);
            skeleton->updateBones();
            int bones_location = glGetUniformLocation(shader->GetID(), "bones");
            glUniformMatrix4fv(bones_location, skeleton->m_boneCount, GL_FALSE,
                  (float*)skeleton->m_bones);
      }

      for (unsigned int i = 0; i < m_gl_info.size(); ++i) {
            glBindVertexArray(m_gl_info[i].m_VAO);
            glDrawElements(GL_TRIANGLES, m_gl_info[i].m_index_count,
                  GL_UNSIGNED_INT, 0);
      }
}
```

So we can see the order of operations:

Scene::UseShader – glUseProgram and some uniforms

Instance::Draw – some more uniforms for the textures

Model::Draw – some more uniforms (MVP and M) and then the Draw call

This all satisfies the order we specified earlier.

This should provide a good framework for setting up a Scene class with Instances.

Client Code that uses the Scene

The Application3D class maybe contains a few models, a camera, and so on at the moment. We want to replace the Camera with a Scene that holds the camera, and move the light info in there too.

Models and Textures can remain members of the Application, and be loaded in startup().

If we make a good Instance constructor we can add Instances to the Scene in one line

e.g

Instance(Model* model, Texture* texture, vec3 pos, vec3 rot, vec3 scale)

{

// copy the arguments to our member variables
}

In Application3D::startup we can have these members:

Scene m_scene;

Model buddha;

Model bunny;

Model soulspear;

Texture checkers;

And in startup() we'd call:

```
// load assets
buddha.Load(“models/Buddha.fbx”);
bunny.Load(“models/bunny.obj”);
checkers.Load(“textures/checks.bmp”);

// set up a scene
m_scene.m_instances.push_back(new Instance(&buddha, &checkers, vec3(0,0,0), vec3(0,0,0), vec3(1)));
m_scene.m_instances.push_back(new Instance(&soulspear, &checkers, vec3(5,0,0), vec3(0,0,0),
vec3(1)));
```

and so on...

Adding a UI

The IMGUI library is built into bootstrap, and makes it very easy to add interactive elements to your graphics engine demo.

To do this, add lines like this to your Application3D::draw function

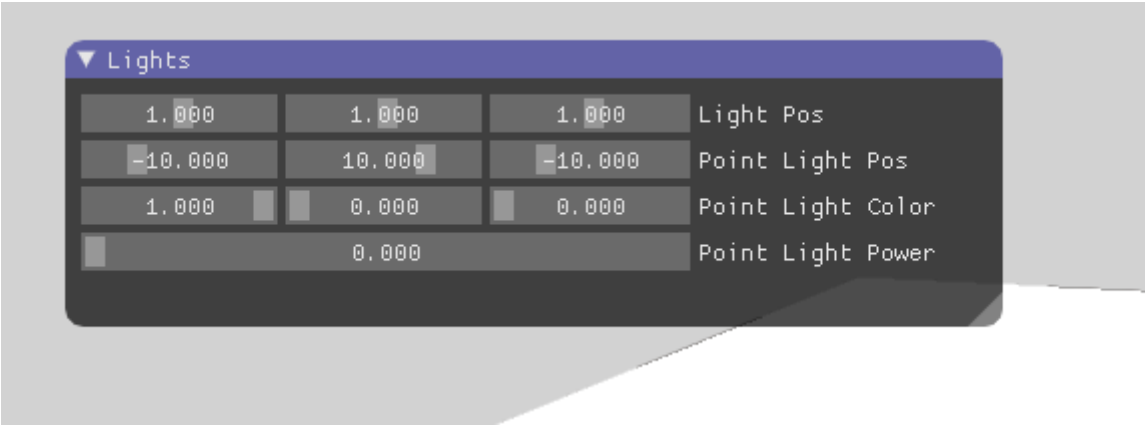
```
ImGui::Begin("Lights");
ImGui::SliderFloat3("Light Pos", m_scene.m_lightDir, -20, 20);

ImGui::SliderFloat3("Point Light Pos", &m_scene.m_pointLightPos[0].x, -20, 20);
ImGui::SliderFloat3("Point Light Color",&m_scene.m_pointLightColor[0].x, 0, 1);
ImGui::SliderFloat("Point Light Power”, &m_scene.m_PointLightPower[0], 0, 100);
ImGui::End();
```

This creates a floating UI box called “Lights” (the argument to ImGui::Begin()) with four rows in it, three vectors and one single slider.

Each entry takes a label to display, the memory address of the float to modify or the first float for multiple values like vectors, and the min and max range for the value.

It displays a set of sliders like so:



13. FrameBuffers

The OpenGL code for Framebuffers is covered in the Canvas tutorial under Post Processing: Frame Buffers. This tutorial is adapted from that.

Its best to create a Framebuffer class from the get-go here, as it makes post-processing much easier to implement when we get to that.

Think of the FrameBuffer as a special texture. We can use it like a texture and display it over a model, but we can also render the scene to it first.

We'll use it in the following ways:

1) Create a FrameBuffer and draw it on a screen in our scene, like a security camera
2) Create a FrameBuffer and then draw it over a quad that fills the whole screen using an unlit shader. This is all a post processing effect!
3) Do deferred rendering. Here we make 3 shader pipelines for rendering the scene and putting the position, normals and albedo (ie texture data) on to three different textures. We then make a post processing effect that reads these three textures and recreates a fully lit scene form them!

Lets create a Frame Buffer class.

Width and height are the width and height of the texture.

m_fbo is the OpenGL handle for using the frame buffer as a render target (as opposed to the actual screen)

m_fboTexture is the Open GL handle to the frame buffer when we want to use it as a texture.

m_fboDepth is a handle to the depth buffer we need for rendering for this rendering target. We can ignore it as its only used internally, not by our client code.

```
class FrameBuffer
{
public:
	FrameBuffer(int w, int h) : width(w), height(h) { SetUp(); }
	void SetUp();
	void RenderScene(Scene& scene);

	unsigned int m_fbo; // frame buffer object
	unsigned int m_fboTexture;
	unsigned int m_fboDepth;
	int width, height;
};
```

The Setup code comes from the Canvas Tutorial. Here it is in full, with comments as to what each line is doing. Its basically allocating a frame buffer on the GPU of the given size, and storing pointers for the framebuffer itself and the texture data it contains.

```
// sets up our render target
void FrameBuffer::SetUp()
{
	// generate a new framebuffer, and set it as the current frame buffer for subsequent operations
	glGenFramebuffers(1, &m_fbo);
	glBindFramebuffer(GL_FRAMEBUFFER, m_fbo);
	// generate a new texture, and set it as the current texture
	glGenTextures(1, &m_fboTexture);
	glBindTexture(GL_TEXTURE_2D, m_fboTexture);
	// allocate for width x height, with RGB 8 bytes each on the GPU
	glTexStorage2D(GL_TEXTURE_2D, 1, GL_RGB8, width, height);
	// set some filtering parameters on the texture.
	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
	// attach this texture and its data to the current framebuffer
	glFramebufferTexture(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, m_fboTexture, 0);
	// create a new depth buffer and set it as the current one
	glGenRenderbuffers(1, &m_fboDepth);
	glBindRenderbuffer(GL_RENDERBUFFER, m_fboDepth);
	// allocate memory on the GPU for the depth buffer
	glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT24, width, height);
	// attach this depth buffer to the current frame buffer
	glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_RENDERBUFFER, m_fboDepth);
	// test everything's set up correctly
	GLenum drawBuffers[] = { GL_COLOR_ATTACHMENT0 };
	glDrawBuffers(1, drawBuffers);
	GLenum status = glCheckFramebufferStatus(GL_FRAMEBUFFER);
```

```
	if (status != GL_FRAMEBUFFER_COMPLETE)
		printf("Framebuffer Error!\n");
	// we're done! detach the frame buffer, so the current one is once again the default screen
	glBindFramebuffer(GL_FRAMEBUFFER, 0);

}
```

Here's a simple function for rendering a scene to this frame buffer. This won't appear on the screen, but will render to a texture that we can then display as a texture on the object.

This is how we use the frame buffer, by setting it to the current frame buffer in the first line. Think of m_fbo as like m_VAO, it contains internal references to the depth buffer and texture handles the same way that m_VAO is enough to point the graphics card at m_VBO and m_IBO for a model render.

```
// draws the specified scene into our render target
void FrameBuffer::RenderScene(Scene& scene)
{
	glBindFramebuffer(GL_FRAMEBUFFER, m_fbo);
	glViewport(0, 0, width, height);
	glClearColor(0.75f, 0.75f, 0.75f, 1);  // could pass this in as argument, or ignore altogether
	glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

	// draw the actual scene here
	scene.Draw((float)width, (float)height, false);

	// restore normal frame buffer after
	glBindFramebuffer(GL_FRAMEBUFFER, 0);
	glViewport(0, 0, 1280, 720);
	glClearColor(0.0f, 0.0f, 0.25f, 1);
	glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
}
```

This is enough for us to make a video camera screen. We can put a FrameBuffer member in our Application3D, and initialise it with SetUp() in startup().

We can then grab its m_fboTexture member (via a public accessor) and assign that to an object. Try assigning it to a cube maybe?

In our Application3D::draw call we then need to render the scene to the frame buffer before we call the normal draw function.

```
frameBuffer->RenderScene(m_scene);
```

Do all this right and the scene should appear wrapped around the cube!

14. Making a Post Processing Effect

To make a postprocessing effect, we basically just make a quad, and position it correctly in front of the camera with the FrameBuffer's texture on it. Rather than actually paste a quad into our scene, we use a simplified shader to position it in clip space to get 1 to 1 pixel accuracy.

On that note, we'll want a FrameBuffer the same dimensions as the screen for a good postprocessing effect.

In Application3D add a member:

```
FrameBuffer* ppFrameBuffer;
```

And initialise it in the startup() function:

```
ppFrameBuffer = new FrameBuffer(getWindowWidth(), getWindowHeight());
```

And draw to it in our draw() function:

```
ppFrameBuffer->RenderScene(m_scene);
```

That's how handy our FrameBuffer class is! We now need to be able to render the framebuffer contents to the screen. We have to set up a couple of very simple shaders and a really simple model to do this.

Here's a utility function that sets up a Model to be the quad we need to render a post processing effect with. You may want to replace my lazy 1280, 720 values with a width and height you pass in to this function, but this will work with a default window. The halfTexel termsa are there to position each screen pixel in the direct centre of a texel, so no unintended interpolation happens.

Think of this function as an alternative to Load – once this is called the model can be drawn because all its gl_info structures have been filled in.

```
void Model::MakePostProcessQuad()
{
```

```
	m_gl_info.resize(1);


	// fullscreen quad
	glm::vec2 halfTexel = 1.0f / glm::vec2(1280, 720) * 0.5f;
	float vertexData[] = {
		-1, -1, 0, 1, halfTexel.x, halfTexel.y,
		1, 1, 0, 1, 1 - halfTexel.x, 1 - halfTexel.y,
		-1, 1, 0, 1, halfTexel.x, 1 - halfTexel.y,
		-1, -1, 0, 1, halfTexel.x, halfTexel.y,
		1, -1, 0, 1, 1 - halfTexel.x, halfTexel.y,
		1, 1, 0, 1, 1 - halfTexel.x, 1 - halfTexel.y,
	};
	glGenVertexArrays(1, &m_gl_info[0].m_VAO);
	glBindVertexArray(m_gl_info[0].m_VAO);
	glGenBuffers(1, &m_gl_info[0].m_VBO);
	glBindBuffer(GL_ARRAY_BUFFER, m_gl_info[0].m_VBO);
	glBufferData(GL_ARRAY_BUFFER, sizeof(float) * 6 * 6,
		vertexData, GL_STATIC_DRAW);
	glEnableVertexAttribArray(0);  // position
	glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE,
		sizeof(float) * 6, 0);
	glEnableVertexAttribArray(1); // UVs
	glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE,
		sizeof(float) * 6, ((char*)0) + 16);
	glBindVertexArray(0);
	glBindBuffer(GL_ARRAY_BUFFER, 0);
}
```

The coordinates for this are in clip space already, covering the screen from -1 to 1 in X and Y, with Z = 0.

So we need a vertex shader that just assigns these positions to gl_Position, and passes the uv coordinates through to the fragment shader.

```
#version 410

layout (location = 0) in vec4 position;
```

```
layout (location = 1) in vec2 texCoord;

out vec2 fTexCoord;

void main() {

	gl_Position = position;

	fTexCoord = texCoord;

}
```

Our fragment shader will just sample a texture, which we'll pass in from our Framebuffer when we draw.

```
#version 410

in vec2 fTexCoord;

out vec4 FragColour;

uniform sampler2D target;

void main()

{

	FragColour = texture(target, fTexCoord);

}
```

We'll need to compile these shaders into a pipeline, and draw the single quad model using them. Note what we're doing here with the FrameBuffer – we're just grabbing its texture handle to pass through to a texture uniform.

```
void FrameBuffer::Draw(unsigned int shader)

{

	// draw out full-screen quad

	glUseProgram(shader);


	glActiveTexture(GL_TEXTURE0);

	glBindTexture(GL_TEXTURE_2D, m_fboTexture);

	int loc = glGetUniformLocation(shader, "target");

	glUniform1i(loc, 0);


	glBindVertexArray(m_model->GetVAO());

	glDrawArrays(GL_TRIANGLES, 0, 6);

}
```

If this all works, you should see the scene rendering as normal. However, you're now set up to use the post-processsing path to do interesting things by modifying your fragment shader.

Have a look at the shader code in the Canvas Post Processing tutorial for some examples to get you started – Box Blur and Distort. (These also illustrate nicely how to use functions in your shader code)

15. Deferred Rendering

Deferred rendering is an optional exercise using FrameBuffers in a more advanced manner.

Here's your step by step rough guide to setting up deferred rendering.

1) Add three frame buffers to your application code to store positions, normals and albedo.
2) Create a simple fragment shader for each one of these frame buffers, that draw the scene and write out the position xyz, normal xyz and the texture colour to the rgb values of the output. Note that the RGB values in the framebuffers are one byte each, so you'll need to normalize your output into the range [0,1] and then un-normlaize it in the fragment shader again.
3) Compile each of these three fragment shaders with your animated and non-animated vertex shaders, and store these six handles in your application.
4) Every frame, before drawing, render to each of these three frame buffers, setting up the default shader pipelines each time so that animated or static models use the right path.
5) Create a post processing pipeline for using these three buffers to render the scene. The vertex shader will be the simple post processing one we sued earlier. The fragment shader should be a copy of your standard fragment shader, but instead of receiving worldNormal, worldPosition and the texture value as in variables, you'll read them from these three other framebuffers.
6) Modify FrameBuffer::Draw so you can pass the three handles for the textures of your position, normal and albedo frame buffers, and set them to the right texture uniforms in your fragment shader.