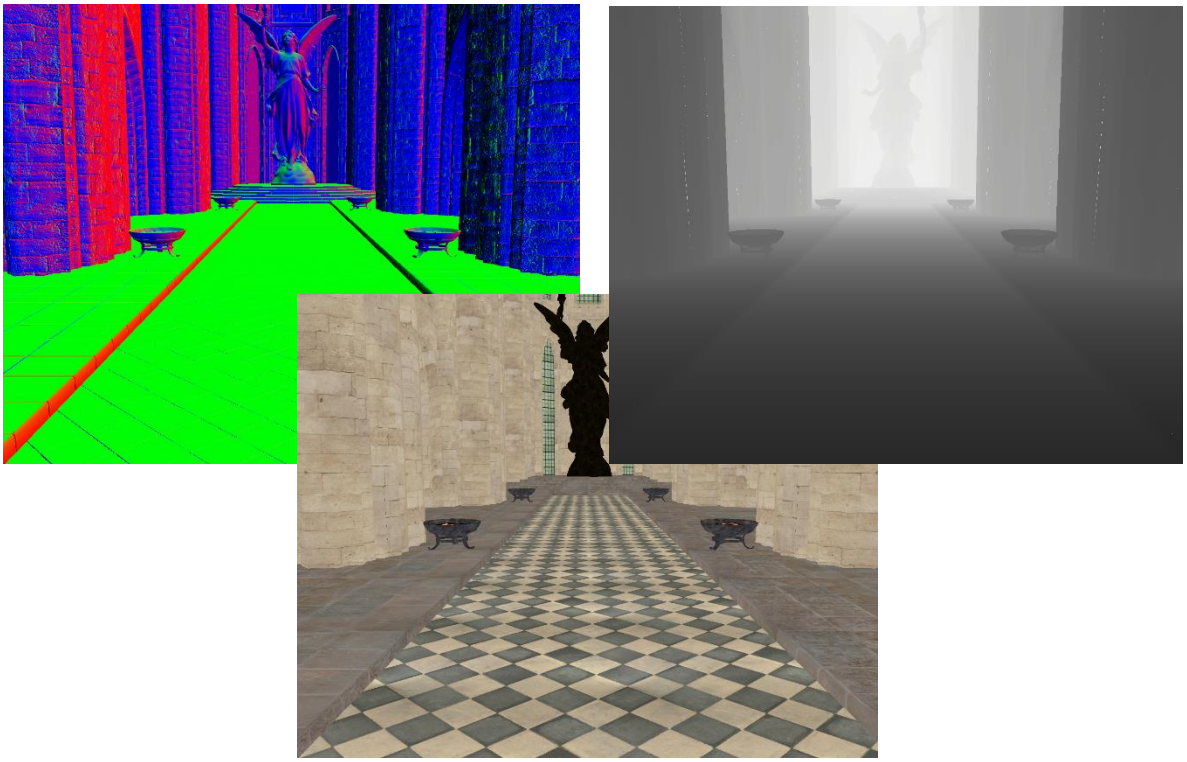
Tutorial 9: Deferred Rendering

This tutorial covers how to modify an existing forward renderer to use deferred rendering.



This tutorial will continue on from the last by modifying the previously created code. It is assumed that you have created a copy of last tutorial's code that can be used for the remainder of this one.

Part 1: Deferred Rendering

Overdraw is an issue where complex calculations are performed to shade a Fragment and then write it to the output buffers. Only a later Fragment turns out to have a closer depth and overwrites the original data. This causes waisted computation calculating Fragment information (often the most complex part of the pipe in modern renderers) that is not needed.

Standard rendering uses a simple "Forward" algorithm that renders all objects and performs fragment shading, depth/stencil testing and output in a single pass. Further render optimisations can be applied to the traditional rendering approach to improve performance by breaking rendering into separate passes. Deferred Rendering uses Multiple Render Targets (MRT) to store data for each object in an initial first pass. The data is stored in what are called "G-Buffers" where each buffer is a texture containing specific data for each pixel on the screen. Each G-Buffer stores data required for rendering. For example one buffer can hold the normals for the rendered object in each pixel. Another buffer can hold the diffuse colour while other buffers hold any other material properties needed.

The second pass then does not need to render geometry as the data is already stored in the G-Buffers. This pass just reads the stored G-Buffers and performs shading calculations based on the stored data. The position doesn't need to be stored as it can be calculated from the stored depth

value by using the cameras projection values to reconstruct it. All other required data to perform lighting equations is read from one or more of the G-Buffers.

A full screen quad can be rendered in the second pass to cover the whole screen causing the Fragment shader to run on all pixels. During this operation the shader just reads in the data from the buffers then performs shading equations as usual. By first storing all the data in the buffers it ensures that the lighting equations are only performed once per pixel.

Since deferred rendering uses fixed buffers to hold material data it does not easily support different material algorithms per object. As such reflection and transparency have to be treated specially. To handle these an accumulation buffer is required. This buffer is used to store the ambient as well as any non-standard lighting effects during the first pass. During the second lighting pass the contribution of each light is then added to the existing accumulation buffer to give the final result.

1. To implement deferred rendering we need to add 3 new programs. The first will render the geometry into the G-Buffers. The second will then read the buffers and then blend together lighting information into a final buffer. The third and last pass will then output the blended buffer to the screen.

```
GLuint g_uiMainProgram;
GLuint g_uiDeferredProgram2;
GLuint g_uiPostProcProgram;
```

2. To implement deferred shading we need to split our existing code across 2 different shader programs. The first program does all the same vertex and tessellation operations. The only thing that needs to change is the Fragment shader as this needs to be updated so that it outputs data without performing any lighting operations. The Fragment shader still needs to perform geometry specific operations like normal and parallax mapping however. Since we are using an accumulation output buffer we also need to perform ambient lighting and the reflection and refraction code as well. By removing the lighting inputs from the main Fragment shader only a portion of the previous inputs are still required.

```
layout(binding = 1) uniform CameraData {
    mat4 m4ViewProjection;
    vec3 v3CameraPosition;
};
layout(binding = 3) uniform ReflectPlaneData {
    mat4 m4ReflectVP;
};
layout(location = 1) uniform float fEmissivePower;
layout(location = 3) uniform float fBumpScale;

layout(binding = 0) uniform sampler2D s2DiffuseTexture;
layout(binding = 1) uniform sampler2D s2SpecularTexture;
layout(binding = 2) uniform sampler2D s2RoughnessTexture;
layout(binding = 3) uniform samplerCube scRefractMapTexture;
layout(binding = 4) uniform sampler2D s2ReflectTexture;
layout(binding = 5) uniform samplerCube scReflectMapTexture;
layout(binding = 9) uniform sampler2D s2NormalTexture;
layout(binding = 10) uniform sampler2D s2BumpTexture;

layout(location = 0) in vec3 v3PositionIn;
layout(location = 1) in vec3 v3NormalIn;
layout(location = 2) in vec2 v2UVIn;
layout(location = 3) in vec3 v3TangentIn;
```

3. Unlike previously we now have multiple outputs from the Fragment shader. Each of these outputs corresponds to an attached output buffer. For this tutorial we have a colour output value for the accumulation buffer. We then have another output for the normal buffer which we will only store the first 2 values of. We then have another buffer to store the diffuse colour and then one final output where we will store the specular colour as well as the roughness. We use "`layout(location)`" to manually specify which attached output buffer each value will be written to.

```
layout(location = 0) out vec3 v3AccumulationOut;
layout(location = 1) out vec2 v2NormalOut;
layout(location = 2) out vec3 v3DiffuseOut;
layout(location = 3) out vec4 v4SpecularRoughOut;
```

Accumulation Colour	
Normal X	Normal Y
Diffuse Colour	
Specular Colour	Roughness

Since we are only storing 2 components of the normal we need a way to reconstruct the 3rd value. Previously we have calculated this value using tangent space normal which will always have a positive z component and thus can be calculated by assuming the input normal has unit length. However, as we are now dealing with world space values then this no longer holds as the z component may point in either the negative or positive direction. To avoid having to store all 3 normal components we will instead use stereoscopic mapping to map the 3 dimensional vector to 2 components which can be stored and then later used to reconstruct the correct 3 component vector.

A vector $\hat{\vec{\boldsymbol{v}}}$ can be converted to a 2-dimensional stereoscopic mapped vector $\hat{\vec{\boldsymbol{v}}}'$ by using:

$$\hat{\vec{\boldsymbol{v}}}' = \frac{\hat{\vec{\boldsymbol{v}}}_{\{x,y\}}}{\hat{\vec{\boldsymbol{v}}}_{\{z\}} + 1}$$

where $\hat{\vec{\boldsymbol{v}}}_{\{x,y\}}$ are the combined 'x', 'y' components and $\hat{\vec{\boldsymbol{v}}}_{\{z\}}$ is the 'z' component of the vector r4espectively.

This can then be used to reconstruct the original vector such that:

$$\hat{\vec{\boldsymbol{v}}}'' = \begin{pmatrix} \hat{\vec{\boldsymbol{v}}}' \\ 1 \end{pmatrix}$$

$$\hat{\vec{\boldsymbol{v}}} = \frac{2\hat{\vec{\boldsymbol{v}}}''}{\hat{\vec{\boldsymbol{v}}}'' \cdot \hat{\vec{\boldsymbol{v}}}''} - \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

We will use these equations in later parts to the tutorial to store the normal in just 2 components and then later when reading back the normal buffer reconstruct the original 3 component vector.

4. The main function now just needs to perform normal and parallax mapping, calculate any emissive, reflective or refractive light and then combine it with the ambient term and write it to the accumulation buffer. The normal, diffuse, specular and roughness values are then written to their corresponding outputs as well.

```
void main() {
    // *** Normalize the inputs here ***

    // *** Generate bitangent here ***

    // *** Perform Parallax Occlusion and then Bump Mapping here ***

    // *** Get texture data here ***

    // Add in ambient contribution
    vec3 v3RetColour = v3DiffuseColour * vec3(0.3f);

    // Add in any reflection contribution
    v3RetColour = ReflectMapUniform(v3RetColour, v3Normal, v3ViewDirection,
v3SpecularColour, fRoughness);

    // Add in any refraction contribution
    v3RetColour = RefractMapUniform(v3RetColour, v3Normal, v3ViewDirection,
v4DiffuseColour, v3SpecularColour);

    // Add in any emissive contribution
    v3RetColour = EmissiveUniform(v3RetColour, v3DiffuseColour);

    // Output to deferred G-Buffers
    v3AccumulationOut = v3RetColour;
    v2NormalOut = v3Normal.xy / (1.0f + v3Normal.z);
    v3DiffuseOut = v3DiffuseColour;
    v4SpecularRoughOut = vec4(v3SpecularColour, fRoughness);
 }
```

5. Based on the Fragment shaders new main function you can remove any other functions not used by the shader (these include the lighting falloff, GGX and shadowing functions etc.).

6. Next we need to make the 2nd shader program that will be run as part of the deferred rendering second pass. This program will be run by rendering a single full-screen quad that will cover the entire screen and cause the Fragment shader to be run on each pixel. For efficiency we will create the full screen quad in screen space so all we have to do is create a new Vertex shader that will pass through the screen space vertices.

```
#version 430 core

layout(location = 0) in vec2 v2VertexPos;

void main() {
    gl_Position = vec4(v2VertexPos, 0.0f, 1.0f);
}
```

7. Now we need to make the second Fragment shader. This shader will have all the parts that were removed from the previous Fragment shader in the previous steps. The inputs for this shader will then be all those required for lighting calculations.

```
struct PointLight {...};
struct SpotLight {...};
#define MAX_LIGHTS 16
layout(binding = 1) uniform CameraData {...};
layout(binding = 2) uniform PointLightData {...};
layout(binding = 5) uniform SpotLightData {...};
layout(binding = 6) uniform CameraShadowData {...};

layout(location = 0) uniform int iNumPointLights;
layout(location = 2) uniform int iNumSpotLights;

layout(binding = 6) uniform sampler2DArrayShadow s2aShadowTexture;
layout(binding = 7) uniform samplerCubeArrayShadow scaPointShadowTexture;
```

8. However, now we have additional inputs for each of the G-Buffers. These are bound to the shader as textures and are accessed as such. So we have 4 new texture inputs corresponding to each G-Buffer in order. Here we use the next available texture binding locations.

```
layout(binding = 11) uniform sampler2D s2DepthTexture;
layout(binding = 12) uniform sampler2D s2NormalTexture;
layout(binding = 13) uniform sampler2D s2DiffuseTexture;
layout(binding = 14) uniform sampler2D s2SpecularRoughTexture;
```

9. The Fragment shader gets all its data from the G-Buffers now so there are no longer any inputs from the Vertex shader. The only output is a single colour value that will get stored in the accumulation buffer.

```
out vec3 v3AccumulationOut;
```

10. Within the shaders main function, the first thing we must now do is to determine the UV coordinate to read in data from the G-Buffers. This can be done by getting the coordinates of the current pixel within the screen using the inbuilt OpenGL variable "gl_FragCoord". We can then divide by the number of pixels in the G-Buffers to get the normalized UV coordinates.

```
// Get UV coordinates
vec2 v2UV = gl_FragCoord.xy * v2InvResolution;
```

11. To get the UV coordinates we need the inverse resolution to be added as an input. We will do this by adding another UBO at the next available binding location.

```
layout(binding = 7) uniform InvResolution {
    vec2 v2InvResolution;
};
```

12. Now we have the UV value, the next step is to read in the data from the G-Buffers. This is done as a simple texture read to get the corresponding data for the current pixel. In the case of the specular colour and the roughness values we just need to unpack them. The normal is the most complex as we need to convert it from stereoscopic mapping back into world space.

```
// Get deferred data
float fDepth = texture(s2DepthTexture, v2UV).r;
vec3 v3Normal = vec3(texture(s2NormalTexture, v2UV).rg, 1.0f);
v3Normal *= 2.0f / dot(v3Normal, v3Normal);
v3Normal -= vec3(0.0f, 0.0f, 1.0f);
vec3 v3DiffuseColour = texture(s2DiffuseTexture, v2UV).rgb;
vec4 v4SpecularRough = texture(s2SpecularRoughTexture, v2UV);
vec3 v3SpecularColour = v4SpecularRough.rgb;
float fRoughness = v4SpecularRough.a;
```

13. We are using the existing depth buffer to retrieve the current world space position. This requires using the current screen coordinates which can be calculated from the UV coordinates to determine the view-projection space position. We then need to convert this to world space. This requires the inverse view-projection matrix. To do this we need to extend the existing "`CameraData`" input to also include the inverse view-projection transform. This must be done in all shaders that use the camera data.

```
layout(binding = 1) uniform CameraData {
    mat4 m4ViewProjection;
    vec3 v3CameraPosition;
    mat4 m4InvViewProjection;
};
```

14. Now we can use the new transform to calculate the world space position. To do this we must convert the depth value and the UV coordinates to clip space [-1,1]. We can then use the inverse view projection matrix to transform the new values into world space.

```
// Calculate position from depth
fDepth = (fDepth * 2.0f) - 1.0f;
vec2 v2NDCUV = (v2UV * 2.0f) - 1.0f;
vec4 v4Position = m4InvViewProjection * vec4(v2NDCUV, fDepth, 1.0f);
vec3 v3PositionIn = v4Position.xyz / v4Position.w;
```

15. Finally, the main function just needs to loop over each of the point and spot lights and add the lighting contribution as done previously in other tutorials. This code can just be copied directly from these previous tutorials.

```
// Normalise the inputs
vec3 v3ViewDirection = normalize(v3CameraPosition - v3PositionIn);

// Loop over each point light
vec3 v3RetColour = vec3(0.0f);
for (int i = 0; i < iNumPointLights; i++) {
    ...
}

// Loop over each spot light
for (int i = 0; i < iNumSpotLights; i++) {
    ...
}

v3AccumulationOut = v3RetColour;
```

16. Finally copy across any additional functions from the old Fragment shader that are needed by the main function. These include the lighting falloff, shadowing and GGX functions etc.

17. The final shader we need to write is the last shader which will render the accumulation buffer to the screen. This will be used by just rendering a full screen quad again so we can reuse the previous Vertex shader. A new Fragment shader is needed that will just read in the value from the accumulation buffer and pass it out.

```
#version 430 core

layout(binding = 7) uniform InvResolution {
    vec2 v2InvResolution;
};
layout(binding = 15) uniform sampler2D s2AccumulationTexture;

out vec3 v3ColourOut;

void main() {
    // Get UV coordinates
    vec2 v2UV = gl_FragCoord.xy * v2InvResolution;

    // Pass through colour data
    v3ColourOut = texture(s2AccumulationTexture, v2UV).rgb;
}
```

18. Now we need to modify our host code so that it loads the new shaders. The first program to load is the deferred rendering 1st pass. This will use the same vertex and tessellation shaders as the main program from the last tutorial so all that needs to be done is to instead load the 1st of the new Fragment shaders in place of the old one (here it is loaded in `g_uiMainProgram`). Next we need to add code to add in the second deferred rendering program. This will use the new Vertex shader and the second of the new Fragment shaders (here it is loaded in `g_uiDeferredProgram2`). Finally, the 3rd program is made from the new Vertex shader again and the last of the new Fragment shaders (here it is loaded in `g_uiPostProcProgram`). Also remember to update the existing "`glProgramUniform1i`" calls to set the number of lights as this is now needed by the second deferred program. Also ensure that the tessellation uniform is also being set this time for the first deferred program.

19. Once the shader program has been set up we now need to add initialise code to create 2 new frame buffers for deferred rendering. The first frame buffer will be set up so that it uses MRT. This is done by binding multiple outputs to the frame buffer at different locations. For this tutorial we need 5 new textures to store the G-Buffer data in. We need a second framebuffer for the second deferred pass were we will perform operations on the contents of the G-Buffers. We also need to create the full screen quad that will be used in the second and third render passes. We also need a UBO to hold the inverse view-projection matrix that we used to calculate the UV coordinates within the Fragment shaders.

```
// Deferred rendering data
GLuint g_uiFBODeferred;
GLuint g_uiFBODeferred2;
GLuint g_uiDepth;
GLuint g_uiAccumulation;
GLuint g_uiNormal;
GLuint g_uiDiffuse;
GLuint g_uiSpecularRough;

// Screen quad
GLuint g_uiQuadVAO;
GLuint g_uiQuadVBO;
GLuint g_uiQuadIBO;

// Inverse resolution UBO
GLuint g_uiInverseResUBO;
```

20. In the initialise section we now need to setup the first new framebuffer and its output textures. To do this we will first setup the new framebuffers and then each of the new textures. For best performance and minimal storage requirements we will try and create each output buffer using the most optimum format. For the depth buffer we will use full 32bit floating point for best precision. For the accumulation buffer we need to store RGB values but we want to support High Dynamic Range output (as this will be used in later tutorials) so we use a floating point format but we use one that still only occupies a total of 32bit for all 3 channels. We do this by using an 11bit floating point red and green channel and a 10bit floating point blue. Since we are trying to reduce memory space we will store the normal by using 16bit floating point values for each channel. Finally as the diffuse and specular textures are just 8bit integer inputs we will only store them as such. For the diffuse

```
// Create first deferred rendering frame buffer
glGenFramebuffers(1, &g_uiFBODeferred);
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, g_uiFBODeferred);
// Setup depth attachment
glGenTextures(1, &g_uiDepth);
glBindTexture(GL_TEXTURE_2D, g_uiDepth);
glTexStorage2D(GL_TEXTURE_2D, 1, GL_DEPTH_COMPONENT32F, g_iWindowWidth,
g_iWindowHeight);
// Setup accumulation attachment
glGenTextures(1, &g_uiAccumulation);
glBindTexture(GL_TEXTURE_2D, g_uiAccumulation);
glTexStorage2D(GL_TEXTURE_2D, 1, GL_R11F_G11F_B10F, g_iWindowWidth,
g_iWindowHeight);
// Setup normal attachment
glGenTextures(1, &g_uiNormal);
glBindTexture(GL_TEXTURE_2D, g_uiNormal);
glTexStorage2D(GL_TEXTURE_2D, 1, GL_RG16F, g_iWindowWidth, g_iWindowHeight);
// Setup diffuse attachment
glGenTextures(1, &g_uiDiffuse);
glBindTexture(GL_TEXTURE_2D, g_uiDiffuse);
glTexStorage2D(GL_TEXTURE_2D, 1, GL_RGB8, g_iWindowWidth, g_iWindowHeight);
// Setup specular and rough attachment
glGenTextures(1, &g_uiSpecularRough);
glBindTexture(GL_TEXTURE_2D, g_uiSpecularRough);
glTexStorage2D(GL_TEXTURE_2D, 1, GL_RGBA8, g_iWindowWidth, g_iWindowHeight);
```

we will use a 3 component format and since the specular has roughness packed into its fourth element we need to use a four component format.

21. Since we are going to use each of the G-Buffers as an input texture in the second deferred render pass we also need to setup texture filtering settings. Since the buffers contain data we don't want to perform any filtering on it when reading as each value already corresponds to a single pixels data. So for each of the new textures we need to add the relevant code to disable their texture filtering options after each one is bound.

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
```

22. Next we need to attach each of the outputs to the frame buffer. The depth buffer will be attached to the depth attachment while the remainder of the buffers will be attached to each of the MRT outputs. Attaching buffers is the same as we have done previously except now we are attaching more than one. The additional buffers are attached to the next output binding point (GL_COLOR_ATTACHMENT1 and so forth).

```
// Attach frame buffer attachments
glFramebufferTexture2D(GL_DRAW_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
GL_TEXTURE_2D, g_uiDepth, 0);
glFramebufferTexture2D(GL_DRAW_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
 GL_TEXTURE_2D, g_uiAccumulation, 0);
glFramebufferTexture2D(GL_DRAW_FRAMEBUFFER, GL_COLOR_ATTACHMENT1,
 GL_TEXTURE_2D, g_uiNormal, 0);
glFramebufferTexture2D(GL_DRAW_FRAMEBUFFER, GL_COLOR_ATTACHMENT2,
 GL_TEXTURE_2D, g_uiDiffuse, 0);
glFramebufferTexture2D(GL_DRAW_FRAMEBUFFER, GL_COLOR_ATTACHMENT3,
 GL_TEXTURE_2D, g_uiSpecularRough, 0);
```

23. Once each of the attachments are bound we need to enable them by using "glDrawBuffers" and specify how many attachments should be enabled and which output attachments should be enabled for the currently bound framebuffer.

```
// Enable frame buffer attachments
GLenum uiDrawBuffers[] = {GL_COLOR_ATTACHMENT0, GL_COLOR_ATTACHMENT1,
 GL_COLOR_ATTACHMENT2, GL_COLOR_ATTACHMENT3};
glDrawBuffers(4, uiDrawBuffers);
```

24. Next we need to set up the second deferred frame buffer. This buffer only has a single output texture as it is used to add lighting to the accumulation texture in the second deferred pass.

```
// Create and attach second frame buffer attachments
glGenFramebuffers(1, &g_uiFBODeferred2);
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, g_uiFBODeferred2);
glFramebufferTexture2D(GL_DRAW_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
 GL_TEXTURE_2D, g_uiAccumulation, 0);
```

25. Since there is only one set of G-Buffer textures we can then bind them just the once during initialisation in order to simplify things. To ensure future texture operations don't affect the current texture units we reset the active texture to unit '0'.

```
// Bind deferred textures
glActiveTexture(GL_TEXTURE11);
glBindTexture(GL_TEXTURE_2D, g_uiDepth);
glActiveTexture(GL_TEXTURE12);
glBindTexture(GL_TEXTURE_2D, g_uiNormal);
glActiveTexture(GL_TEXTURE13);
glBindTexture(GL_TEXTURE_2D, g_uiDiffuse);
glActiveTexture(GL_TEXTURE14);
glBindTexture(GL_TEXTURE_2D, g_uiSpecularRough);
glActiveTexture(GL_TEXTURE15);
glBindTexture(GL_TEXTURE_2D, g_uiAccumulation);
glActiveTexture(GL_TEXTURE0);
```

26. The next initialisation step is to setup the full screen quad that we need to perform the last few deferred render operations. Just like with previous objects this one will need a VAO, VBO and an IBO.

```
// Generate the full screen quad
glGenVertexArrays(1, &g_uiQuadVAO);
glGenBuffers(1, &g_uiQuadVBO);
glGenBuffers(1, &g_uiQuadIBO);
glBindVertexArray(g_uiQuadVAO);
```

27. Next we create the VBO. As we are rendering in screen coordinates we only need the x and y value so only two elements are needed for the position. We only need a point for each corner of the quad so we will create a list of 4 points. The new VBO will hold the points for the bottom-left, bottom-right, top-right and top-left respectively.

```
// Create VBO data
glBindBuffer(GL_ARRAY_BUFFER, g_uiQuadVBO);
GLfloat fVertexData[] = {
    -1.0f, -1.0f,
     1.0f, -1.0f,
     1.0f,  1.0f,
    -1.0f,  1.0f
};
glBufferData(GL_ARRAY_BUFFER, sizeof(fVertexData), fVertexData,
GL_STATIC_DRAW);
glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 2 * sizeof(GLfloat),
 (const GLvoid *)0);
glEnableVertexAttribArray(0);
```

28. Next we setup the IBO data by specifying 2 counter-clockwise triangles making up the quad. Since there are only 4 vertices we can store the indices using just a single byte each.

```
// Create IBO data
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, g_uiQuadIBO);
GLubyte ubIndexData[] = {
    0, 1, 3,
    1, 2, 3
};
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, g_uiQuadIBO);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(ubIndexData), ubIndexData,
 GL_STATIC_DRAW);
```

29. As the accumulation buffer needs to be added to by the second pass we need to setup the ability to blend the new values with the existing contents of the accumulation buffer so that new values don't erase the data already stored in there. To do this we need to use OpenGL's blend ability. To set this up we to need to specify how the new data will be blended in with the existing contents of the buffer. The first part of this is specifying the weighting to be applied to the new and existing data when combining them. Since we want equal amounts of the new and existing data we set the 2 weighting factors to be '1' using "GL_ONE" and passing it into "glBlendFunc". We then specify the function that will be used to combine the 2 values. We just want to add the new value to the existing so we specify the "GL_FUNC_ADD" function by using "glBlendEquation". With these two combined the accumulation will take '1' times the new value and add it to '1' times the existing value.

```
// Setup blending parameters
glBlendFunc(GL_ONE, GL_ONE);
glBlendEquation(GL_FUNC_ADD);
```

30. Next we need to initialise the UBO used to store the inverse resolution. This is as simple as inverting the existing width/height values and then binding the UBO to the corresponding binding location.

```
// Setup inverse resolution
glGenBuffers(1, &g_uiInverseResUBO);
vec2 v2InverseRes = 1.0f / vec2((float)g_iWindowWidth,
(float)g_iWindowHeight);
glBindBuffer(GL_UNIFORM_BUFFER, g_uiInverseResUBO);
glBufferData(GL_UNIFORM_BUFFER, sizeof(vec2), &v2InverseRes,
GL_STATIC_DRAW);
glBindBufferBase(GL_UNIFORM_BUFFER, 7, g_uiInverseResUBO);
```

31. Now the initialisation is complete you should add in the necessary code to clean-up the newly created items during program exit.

32. As the second deferred pass needs the inverse view-projection transform to calculate the world-space position we need to update the existing "CameraData" object to hold the additional data.

```
struct CameraData
{
    aligned_mat4 m_m4ViewProjection;
    aligned_vec3 m_v3Position;
    aligned_mat4 m_m4InvViewProjection;
};
```

33. We can now modify the update function so that the inverse view-projection is also calculated when setting the camera data.

```
// Calculate inverse view projection
mat4 m4InvViewProjection = inverse(m4ViewProjection);

// Create updated camera data
CameraData Camera = {
    m4ViewProjection,
    g_SceneData.m_LocalCamera.m_v3Position,
    m4InvViewProjection};
```

34. Next we need to update the render function to use deferred rendering. To simplify this we will create 2 new functions that handle the first two passes and then the final pass respectively. The first function will be responsible for the first 2 deferred render passes. It will first bind the deferred render buffer and then set the first render program. It will then call "GL_RenderObjects" to render all objects into the G-Buffers. It will then bind the second deferred program and framebuffer and render the full screen quad. Since this full

```
void GL_RenderDeferred(ObjectData * p_Object = NULL, GLuint uiAccumBuffer = 0,
GLenum uiTextureTarget = GL_TEXTURE_2D)
{
    // Bind deferred frame buffer
    glBindFramebuffer(GL_DRAW_FRAMEBUFFER, g_uiFBODeferred);
    // Attach optional accumulation buffer
    if (uiAccumBuffer > 0)
        glFramebufferTexture2D(GL_DRAW_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
uiTextureTarget, uiAccumBuffer, 0);

    // Bind first deferred program
    glUseProgram(g_uiMainProgram);
    // Render all objects
    GL_RenderObjects(p_Object);

    // Disable depth checks
    glDisable(GL_DEPTH_TEST);
    glDepthMask(GL_FALSE);
    // Bind full screen quad
    glBindVertexArray(g_uiQuadVAO);

    // Bind second deferred frame buffer
    glBindFramebuffer(GL_DRAW_FRAMEBUFFER, g_uiFBODeferred2);
    // Attach optional accumulation buffer
    if (uiAccumBuffer > 0)
        glFramebufferTexture2D(GL_DRAW_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
uiTextureTarget, uiAccumBuffer, 0);

    //Enable blending
    glEnablei(GL_BLEND, 0);

    // Bind second deferred program
    glUseProgram(g_uiDeferredProgram2);
    // Draw full screen quad
    glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_BYTE, 0);

    // Disable blending
    glDisablei(GL_BLEND, 0);
    // Reset accumulation buffer
    if (uiAccumBuffer > 0) {
        glFramebufferTexture2D(GL_DRAW_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
GL_TEXTURE_2D, g_uiAccumulation, 0);
        glBindFramebuffer(GL_DRAW_FRAMEBUFFER, g_uiFBODeferred);
        glFramebufferTexture2D(GL_DRAW_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
GL_TEXTURE_2D, g_uiAccumulation, 0);
        // Enable depth tests again
        glEnable(GL_DEPTH_TEST);
        glDepthMask(GL_TRUE);
    }
}
```

screen quad doesn't need to be depth tested we disable the depth test. We also enable the blend mode by using "glEnablei" with the blend mode "GL_BLEND" and then specifying '0' so that blend is only enabled on the framebuffers first output (i.e. at index '0'). Since we now only have a single output for this pass we use the second deferred framebuffer. The new function is similar to "GL_RenderObjects" in that it accepts an optional pointer to an existing object that should be skipped during rendering. However it also includes 2 additional optional variables. These can be used to specify a different output texture that should be used for the accumulation buffer. This is useful for rendering directly to an existing texture without needing a final pass (we will use this for reflection textures). If an accumulation texture is specified then it is attached to the deferred rendering framebuffers. Deferred rendering will then render to the texture as normal. After the deferred passes the deferred framebuffers are reset to normal and the depth mode is also re-enabled. If no accumulation buffer was specified then the depth mode will be reset in the second rendering function.

35. The second function is responsible for performing the final pass that outputs the accumulation buffer. This is in a separate function so that the final render output can be skipped when not required (like when rendering reflections). This way the same functions can be used to output to screen or to texture. The second function must now use the final program and then it will draw the full-screen quad. Once it is completed it then re-enables the depth test and disables the blend function.

```
void GL_RenderPostProcess()
{
    // Bind final program
    glUseProgram(g_uiPostProcProgram);

    // Bind default frame buffer
    glBindFramebuffer(GL_DRAW_FRAMEBUFFER, 0);
    glClear(GL_COLOR_BUFFER_BIT);

    // Draw full screen quad
    glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_BYTE, 0);

    // Enable depth tests again
    glEnable(GL_DEPTH_TEST);
    glDepthMask(GL_TRUE);
}
```

36. Now we can use the new deferred functions to update the render function. Instead of the existing code to bind the framebuffer and camera and then call "GL_RenderObjects" we instead call the first deferred function.

```
// Bind default camera
glBindBufferBase(GL_UNIFORM_BUFFER, 1, g_SceneData.m_uiCameraUBO);

// Perform deferred render pass
GL_RenderDeferred();

// Perform final pass
GL_RenderPostProcess();
```

37. This completes the regular render code however we now need to update all the code that creates the reflection maps. Previously we have had 2 framebuffers that we have used for reflections (one for planar and one for cube map). Because deferred rendering requires a lot of memory to store all the G-Buffers a deferred framebuffer that holds a cube map would be notably large. So to avoid issues we will instead just use the existing deferred rendering code to render each face of the environment map one at a time. This means we no longer need the cube map framebuffer. As we now have an FBO for deferred rendering we don't even need a reflection framebuffer. As a result we can simplify the existing 2 framebuffers and associated render buffers to just a single UBO.

```
GLuint g_uiReflectCameraUBO;
```

38. This simplifies the initialisation of the reflection framebuffers and associated data as now all we need to do is generate the UBO.

```
// Generate FBO camera data
glGenBuffers(1, &g_uiReflectCameraUBO);
```

39. You should also update the code in the quit function that deletes the framebuffers and UBOs to match.

40. The first thing we will update is the render code for planar reflections. Since we added the inverse view-projection to the camera data then we need to also update the creation of the planar camera UBO to add the inverse transform.

```
// Create updated camera data
mat4 m4ViewProjection = m4ReflectProj * m4ReflectView;
mat4 m4InvViewProjection = inverse(m4ViewProjection);
CameraData Camera = {
    m4ViewProjection,
    v3ReflectPosition,
    m4InvViewProjection};

// Update the camera buffer
glBindBuffer(GL_UNIFORM_BUFFER, g_uiReflectCameraUBO);
glBufferData(GL_UNIFORM_BUFFER, sizeof(CameraData), &Camera,
GL_STATIC_DRAW);
glBindBufferBase(GL_UNIFORM_BUFFER, 1, g_uiReflectCameraUBO);
```

41. Now we need to update the drawing code for planar reflections. We now need to perform the full deferred rendering pass by calling "GL_RenderDeferred". However we don't need the final post-process pass as we can simply render it directly to our existing texture. So we pass in the texture as an additional input to the deferred render function removing the need to perform any additional passes.

```
// Perform deferred render pass
GL_RenderDeferred(p_Object, p_Object->m_uiReflect);
```

42. Now we need to update the environment map rendering code. This code has previously used a geometry shader to render all 6 cube map faces at once. Doing this with deferred rendering would require a significant amount of G-Buffer storage. So we will use our existing deferred shaders to render each cube map face one at a time. As a result we do not need the old cube map reflection shader program which can now be removed. In its place we will use the deferred render program. To do this we need to replace all the existing render code from after the cube map view-projections have been calculated all the way through till we bind the reflection texture and generate its mipmaps. In its place we will first set the viewport to be square to match the cube map faces. We can use the existing deferred render buffers during the deferred pass but only render to a square subsection of the G-Buffers. This will work assuming that window width is always greater than the height and the cube map faces are all set to be square with their width and height both set to be exactly the size of the window height. Check to ensure that the cube map face sizes are set to both use the screen height in the scene loading code.

```
// Update the viewport (must also update inverse resolution UBO)
glViewport(0, 0, g_iWindowHeight, g_iWindowHeight);
glBindBufferBase(GL_UNIFORM_BUFFER, 7, g_uiReflectInverseResUBO);

// Bind the UBO buffer as camera
glBindBufferBase(GL_UNIFORM_BUFFER, 1, g_uiReflectCameraUBO);

// Loop over each face in cube map
for (int i = 0; i < 6; i++) {
    // Create updated camera data
    CameraData Camera = {
        m4CubeViewProjections[i],
        v3Position,
        inverse(m4CubeViewProjections[i])};

    // Update the objects projection UBO
    glBindBuffer(GL_UNIFORM_BUFFER, g_uiReflectCameraUBO);
    glBufferData(GL_UNIFORM_BUFFER, sizeof(CameraData), &Camera,
GL_STATIC_DRAW);

    // Perform deferred render pass
    GL_RenderDeferred(p_Object, p_Object->m_uiReflect,
GL_TEXTURE_CUBE_MAP_POSITIVE_X + i);
}

// Reset to default viewport
glViewport(0, 0, g_iWindowWidth, g_iWindowHeight);
glBindBufferBase(GL_UNIFORM_BUFFER, 7, g_uiInverseResUBO);
```

After we have set the viewport we just bind the reflection camera UBO and then loop over each of the 6 cube map faces. For each face we need to get the corresponding camera data and load it into the UBO. We then call the deferred rendering function and set the colour output attachment to the current cube map face. Here we again take advantage of being able to render directly to the reflection texture by using the deferred render functions optional inputs. Here we actually use the 3rd parameter so we can specify the cube map face that should be used.

43. To use the previous code you must add additional code that creates and initialises "g_uiReflectInverseResUBO". This code is identical to that used for "g_uiInverseResUBO" except that the dimensions correspond to those of each cube map face.

44. You should now be able to compile and run your program using deferred rendering.

Extra:

45. Modify the post-process Fragment shader so that instead of outputting the calculated lighting colour it instead outputs the values retrieved from one of the G-Buffers. This will allow you to see the contents of each of the G-Buffers. Modify the program so that it supports changing between normal rendering and outputting of each of the G-buffers based on user input key presses.
    Note: Be aware the depth will look strange unless you convert it to linear space. This requires performing an inverse projection operation to convert the stored depth and then scaling it based on the cameras near and far view distances.

```
// Get depth from input texture
float fDepthVal = texture(s2DepthTexture, v2UV).r;

// Convert depth to -1->1 range
fDepthVal = (2.0f * fDepthVal) - 1.0f;

// Calculate world space depth by doing inverse projection operation (using -z)
float fDepth = (2.0f * v2NearFar.x * v2NearFar.y) /
((v2NearFar.y + v2NearFar.x) - (fDepthVal * (v2NearFar.y - v2NearFar.x)));

// Scale the new depth value between near and far (0->1 range)
fDepth = (fDepth - v2NearFar.x) / (v2NearFar.y - v2NearFar.x);
```