Rendering in Multiple Passes

Notes for a Msc Course in Computer Graphics

University of Minho
António Ramires

Introduction

Up until now we have been rendering everything with a single set of shaders. The geometry was fed to the graphics pipeline and, on the other end, an image was produced as output. However, not all situations can be solved with a single pass, shadows being the most common example. On the other hand, there may be some scenarios where rendering is more efficient when using multiple passes.

Lets consider a simple example: rendering a lit object. The shaders in the lighting section are able to simulate lighting on an object using a single pass. The fragment shader will be executed each time a pixel passes the depth test, potentially causing it to be executed many more times than the number of pixels available. How many more pixels is a quantity that is scene dependent as well as camera position dependent. A sorting mechanism could be used to reduce overdrawing, but it will always be present.

If the lighting fragment shader is a heavy shader from a performance point of view, then overdrawing could cause the application to slow down. In these situations an approach called *deferred rendering* can help us to get performance back on track. As mentioned, this is not a solution for all problems, and it can cause a performance decrease when there is little to no overdraw, or when the pixel operations are very simple, so use this with care.

The main goal of this approach is to divide the rendering process in two (or more) steps. In the first step, where all the geometry is present, hence overdraw is likely to occur, we only store the required information to be able to compute the illumination per pixel. In this first step we don't perform any complex computation, we just store values. For instance to compute Phong's lighting model we need to know several pieces of information, namely the light's direction, the eye vector (a vector pointing from the camera to the point, in camera space), and the normal. All these vectors must be in camera space.

So the first step will just save these three vertices in textures. Then the second step will receive as input those three textures and create the final rendered object.

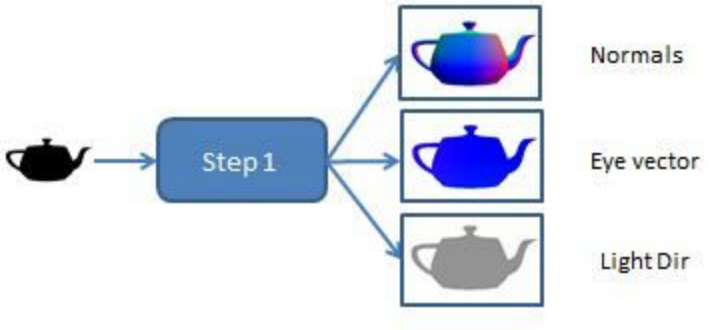

Figure. Inputs and outputs of step 1

The next step will draw a single plane. It will receive the three previously created textures as inputs and it will output the final image on screen.

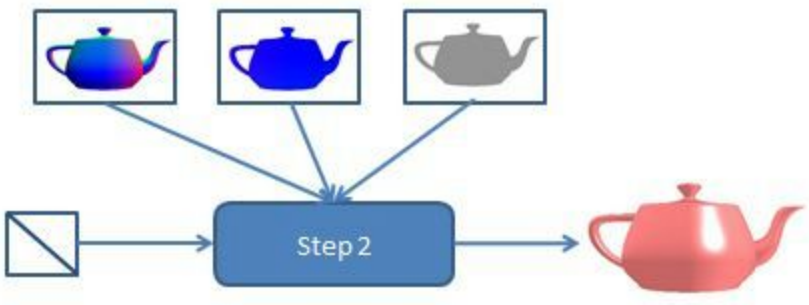

Figure. Inputs and outputs of step 2

OpenGL Setup

Regarding the OpenGL side of the application, there is some work that needs to be done as far as setup goes.

OpenGL renders always to a frame buffer, which by default is the same as rendering to screen. A frame buffer is a set of buffers we can write to. For instance in the default frame buffer we commonly write to a color buffer, and implicitly to a depth buffer.

The API lets us define our own frame buffer, where we can include the set of buffers we require for our application. In the example above we would need a depth buffer and three colour buffers. So lets see how to define a frame buffer that suits our needs.

Frame buffers behave very much as any other object in OpenGL. To set them up, we have to generate them, bind them, add information to them, and unbind them at the end. This last step

is optional, but it is a good practice.

The following snippet of code takes care of the initial stages:

```
GLuint fbo;
...
// Generate one frame buffer
glGenFramebuffers(1, &fbo);
// bind it
glBindFramebuffer(GL_FRAMEBUFFER, fbo);
```

After the frame buffer is created, and bound, we need to add render targets. A frame buffer can have a depth buffer, a stencil buffer, and a number of colour buffers. These targets come in two shapes: textures or render buffers. While some claim that render buffers may be further optimized by the driver, textures provide more flexibility, as a texture can work both as output, from a frame buffer, and input, as a regular texture.

In here we are going to use textures for the colour buffers, since we want to feed them to the next step, and a render buffer will be used to store the depths.

The following function creates empty textures. i.e. textures that have no initial data stored in them.

```
GLuint createRGBATexture(int w, int h) {

      GLuint tex;

      glGenTextures(1, &tex);
      glBindTexture(GL_TEXTURE_2D, tex);

      glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,
GL_LINEAR);
      glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,
GL_LINEAR);
      glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
      glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);

      glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, w,h,
                                 0, GL_RGBA, GL_FLOAT,
                                 NULL);

      glBindTexture(GL_TEXTURE_2D, 0);
      return(tex);
```

```
}
```

To add a texture, using its texture ID, to a frame buffer we can use the `glFramebufferTexture` function. For instance to add a texture for the first color target we could write:

```
tex = createRGBATexture(w,h);
glFramebufferTexture(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, tex, 0);
```

The second parameter specifies the attachment we're making. Allowed values are `GL_COLOR_ATTACHMENT`*i*, `GL_DEPTH_ATTACHMENT`, `GL_STENCIL_ATTACHMENT` or `GL_DEPTH_STENCIL_ATTACHMMENT`. The third parameter is the texture ID, and the last the mipmap level.

Render buffers need to be created, bound and initialized, as do all types of buffers in OpenGL. Then we specify its storage type and dimensions with `glRenderbufferStorage`. For instance, the following code creates a render buffer that can be used to store 24 bit depth components.

```
GLuint fb;
glGenRenderbuffers(1, &fb);
glBindRenderbuffer(GL_RENDERBUFFER, fb);
glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT24, w, h);
```

Finally we attach the render buffer to the bound frame buffer, for instance,

```
glFramebufferRenderbuffer(GL_FRAMEBUFFER,
                      GL_DEPTH_ATTACHMENT,
                      GL_RENDERBUFFER, fb);
```

To check if everything is OK we can call

```
GLenum e = glCheckFramebufferStatus(GL_DRAW_FRAMEBUFFER);
```

and if `e == GL_FRAMEBUFFER_COMPLETE` we can move on to the next stage.

The following function creates a frame buffer with *n* textures as color attachments and a render buffer to store the depths. It assumes that `fbo`, and `texFBO` are constants defined as follows:

```
GLuint fbo, texFBO[n];
```

This function should be called in the setup phase of the application. It takes as parameters the size of the frame buffer (it is not required that this size is the same as the windows size), and the

number of colour attachments.

```
void prepareFBO(int w, int h, int colorCount) {

      // Generate one frame buffer
      glGenFramebuffers(1, &fbo);
      // bind it
      glBindFramebuffer(GL_DRAW_FRAMEBUFFER, fbo);
      // attach textures for colors
      for (int i = 0; i < colorCount; ++i) {

            texFBO[i] = createRGBATexture(w,h);
            glFramebufferTexture(GL_FRAMEBUFFER,
                        GL_COLOR_ATTACHMENT0+i, texFBO[i], 0);
      }
      // attach renderbuffer for depth
      GLuint fb;
      glGenRenderbuffers(1, &fb);
      glBindRenderbuffer(GL_RENDERBUFFER, fb);
      glRenderbufferStorage(GL_RENDERBUFFER,
                              GL_DEPTH_COMPONENT24, w,h);
      glFramebufferRenderbuffer(GL_FRAMEBUFFER,
                              GL_DEPTH_ATTACHMENT,
                              GL_RENDERBUFFER, fb);

      // check if everything is OK
      GLenum e = glCheckFramebufferStatus(GL_DRAW_FRAMEBUFFER);
      if (e != GL_FRAMEBUFFER_COMPLETE) {
            printf("There is a problem with the FBO\n");
      }
      // unbind fbo
      glBindFramebuffer(GL_DRAW_FRAMEBUFFER, 0);
}
```

Now, in the shader setup, for step 1, we have to bind the outputs in the fragment shader.

```
glBindFragDataLocation(pProgram1, 0,"cout0");
glBindFragDataLocation(pProgram1, 1,"cout1");
glBindFragDataLocation(pProgram1, 2,"cout2");
```

where `pProgram1` is the GLSL program for step 1, or, using VSL,

```
step1Shader.setProgramOutput(0,"cout0");
```

```
step1Shader.setProgramOutput(1,"cout1");
step1Shader.setProgramOutput(2,"cout2");
```

where `step1Shader` is the shader object in VSL.

We also have to bind the three texture samplers to step 2 shader, which, using VSL, we could do as follows:

```
step2Shader.setUniform("normalTex", 0);
step2Shader.setUniform("lightDirTex", 1);
step2Shader.setUniform("eyeVectorTex", 2);
```

The names that appear in the function calls above must be the names of the samplers in the shaders.

OpenGL Rendering Cycle

The rendering cycle requires that we define the two steps. For each step we must bind the appropriate frame buffer, set the viewport (if using different viewports on different steps), define the drawing buffers, and then draw as usual.

In the above example we have a first step which is a normal rendering, with a camera setup, and matrices involved, and a second step which is only drawing a plane.

Note that setting the matrices in each step is only strictly required if the shaders use those matrices, otherwise this step can be saved in the release version. By setting the matrices in every step we are basically staying on the safe side against future shader changes.

The rendering cycle could be written as follows:

```
// PASS 1
// Bind FPO
glBindFramebuffer(GL_FRAMEBUFFER, fbo);

// Set Drawing buffers
GLuint attachments[3] = {  GL_COLOR_ATTACHMENT0,
                           GL_COLOR_ATTACHMENT1,
                           GL_COLOR_ATTACHMENT2};
glDrawBuffers(3,  attachments);

// clear color and depth
```

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

// use step 1 shader
glUseProgram(step1Shader.getProgramIndex());

// Load identity matrices
vsml->loadIdentity(VSMathLib::VIEW);
vsml->loadIdentity(VSMathLib::MODEL);

// this is the resolution of the render target
glViewport(0, 0, FBO_WIDTH,FBO_HEIGHT);
// set camera
vsml->lookAt(camX, camY, camZ, 0,0,0, 0,1,0);

// pass light position in camera space to shader
float l[4] = {1.0, 1.0, 1.0, 0.0};
float lTrans[4];
vsml->multMatrixPoint(VSMathLib::VIEW, l, lTrans);
step1Shader.setUniform("lightDir", lTrans);

// render model
myModel.render();

// PASS 2
// get back to the default frame buffer, i.e. output to the screen
glBindFramebuffer(GL_FRAMEBUFFER, 0);
// use shader for step 2
glUseProgram(step2Shader.getProgramIndex());
// clear default frame buffer
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

// load identity matrices
vsml->loadIdentity(VSMathLib::MODEL);
vsml->loadIdentity(VSMathLib::VIEW);
// save projection matrix
vsml->pushMatrix(VSMathLib::PROJECTION);
vsml->loadIdentity(VSMathLib::PROJECTION);

glViewport(0, 0, WINDOW_WIDTH, WINDOW_HEIGHT);

// render plane
vsml->rotate(90, 1.0, 0.0, 0.0);
plane.render();
```

```
vsml->popMatrix(VSMathLib::PROJECTION);
```

Shaders

All that is left are the shaders themselves. For step 1 we only need to store the info, hence the shaders are very simple:

Vertex Shader

```
#version 420

uniform mat4 pvm;
uniform vec3 lightDir;

in vec4 position;
in vec3 normal;

out vec3 normalV, eyeV, lightDirV;

void main () {

	normalV = normalize( normalMatrix * normal);
	lightDirV = normalize(lightDir);
	eyeV = -normalize(vec3(modelViewMatrix * position));

	gl_Position = pvm * position;
}
```

Fragment Shader

```
#version 420

in vec3 normalV, eyeV, lightDirV;

out vec4 cout0, cout1, cout2;

void main() {
	cout0 = vec4(normalize(normalV) * 0.5 + 0.5, 0.0);
	cout1 = vec4(lightDirV * 0.5 + 0.5, 0.0);
	cout2 = vec4(normalize(eyeV) * 0.5 + 0.5, 0.0);
```

```
}
```

In step 2, the vertex shader passes the texture coordinates to the fragment shader, which uses them to retrieve the saved data. After this step, the fragment shader proceeds to compute the pixel colour, which in this example is simply Phong lighting.

Vertex Shader

```
#version 420

uniform mat4 modelMatrix;

in vec4 position;
in vec2 texCoord;

out vec2 texCoordV;

void main () {
      texCoordV = texCoord;
      gl_Position = modelMatrix * position;
}
```

Fragment Shader

```
#version 420

uniform vec4 diffuse, specular;
uniform sampler2D normalTex, lightDirTex, eyeVectorTex;

in vec2 texCoordV;

out vec4 cout;

void main() {

      vec4 dif, spec;
      vec3 n, l, h, e;

      n = texture(normalTex, texCoordV).rgb * 2.0 - 1.0 ;
      e =texture(eyeVectorTex, texCoordV).rgb * 2.0 - 1.0 ;
      l = texture(lightDirTex, texCoordV).rgb ;

      float intensity = max(dot(n, l), 0.0);
```

```
    h = normalize(l + e);
    float intSpec = max(dot(h,n), 0.0);
    spec = specular * pow(intSpec,100);
    dif = diffuse ;

    cout = (intensity + 0.2) * dif + spec ;
}
```