COM3503/4503/6503: 3D Computer Graphics

Dr Steve Maddock

Exercise sheet 5: Lighting

You should read through Joey de Vries's tutorial sections on 'Lighting': https://learnopengl.com/ as I will loosely follow this.

1 Introduction

At this stage, it is worth summarising where we are at. Figure 1.1 shows the data that we have been transferring to the GPU, via vertex and fragment attributes. We have used this structure to render textures on triangles and also to create a camera system. So far, we have drawn a single kind of object in each of our programs, for example, one cube or 100 cubes. Today we will look at shading objects according to the position of a light. We'll also look at using different objects in the same scene.

CPU

Main program	**Render loop**
Create textures from bitmaps in files	Transfer values to uniforms
Setup Shaders	Make texture units active
Fill buffers with vertex and index data	Select vertex/index data to render

GPU

Vertex buffers (for different meshes)	Index buffers (for different meshes)
Texture data – multiple textures can be stored	Shaders – we can have multiple shaders

Uniforms e.g. uniform mat4 mvpMatrix;		Uniforms e.g. uniform sampler2D first_texture;
Vertex shader	Rasterisation process	Fragment shader
Lots of copies of the vertex shader each dealing with a different vertex		Lots of copies of the fragment shader each dealing with a different fragment

Figure 1.1: Transferring data from CPU to GPU.

1.1 gmaths

I've made some minor changes to the gmaths package, so you will need to recompile that when you download it:

```
javac gmaths\*.java
```

1.2 Compiling

Now that the programs are becoming a little more complicated, it is worth compiling all the java classes when you make changes:

```
javac *.java
```

and then running the specific program, e.g. java L01.

1.3 Separate folders

In the examples below, I've used separate folders as we progress through the programs. It is important to keep the code in separate folders. This is because some of the classes are updated from folder to folder – a version in one folder will not necessarily work in another folder.

2 A light

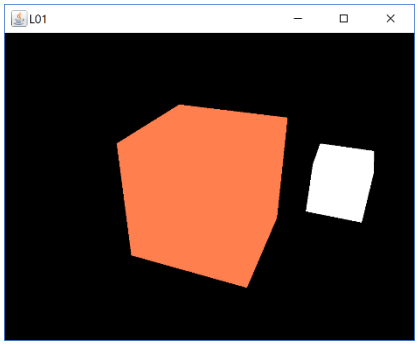

Figure 2.1: Output from Week5_2_start\L01.java.

Program: Week5_2_start\L01.java. Figure 2.1 shows the output.

This makes use of the cube used in previous examples as the object to render. This is composed of 12 triangles and 24 vertices (since each 'corner' vertex is repeated three times, once per surface of the cube, so that each has different normal). Each vertex has an x,y,z position, an x,y,z normal (normalised), and texture coordinates, (s,t). We won't use the texture coordinates in early examples even though we will still transfer them to the GPU as part of the vertex attributes. We'll just ignore them in the shaders – for our early examples, the effects of the light will be clearer on surfaces without texture.

The light is represented a simplified version of the cube data. We could have added a light to our scene without physically representing the light. However, this makes it difficult to understand where the light is and what effect it should be having on the scene. So, it is represented as an object. The 'simplified cube' is composed of 12 triangles, but only uses 8 vertices, with x,y,z position attributes. Vertex normals are not required, since we are not shading the light; we are just displaying it. Also, we are not interested in texturing the light, so texture coordinates are not required.

Program L01.java achieves its aim at the expense of some repetitive code, which we'll clear up in later examples. For now, I've again decided to limit the scope of the program code that needs to be focussed on, at the expense of efficiency. Thus two sets of vertex and index data are included in L01_GLEventListener.java , as well as two different fillBuffers() methods.

Separate vertex and fragment shaders are used for the cube object and for the light objects. Program listings 2.1 and 2.2 give the vertex and fragment shaders for the cube object, and

```
#version 330 core

layout (location = 0) in vec3 position;
layout (location = 1) in vec3 normal;
layout (location = 2) in vec2 texCoord;

out vec3 fragPos;
out vec3 ourNormal;

uniform mat4 model;
uniform mat4 mvpMatrix;

void main() {
  gl_Position = mvpMatrix * vec4(position, 1.0);
  fragPos = vec3(model*vec4(position, 1.0f));
  ourNormal = mat3(transpose(inverse(model))) * normal;
}
```

Program listing 2.1: vs_cube_01.txt.

```
#version 330 core

in vec3 fragPos;
in vec3 ourNormal;

out vec4 fragColor;

uniform vec3 objectColor;
uniform vec3 lightColor;
uniform vec3 lightPos;
uniform vec3 viewPos;

void main() {
  fragColor = vec4(objectColor*lightColor, 1.0f);
}
```

Program listing 2.2: fs_cube_01.txt.

Program listings 2.3 and 2.4 give the vertex and fragment shaders for the light object, respectively.

The vertex shader for the cube (Program listing 2.1) contains a lot of information that we don't need for the first example, but that will be useful in subsequent examples. For example, fragPos and ourNormal are not used in the fragment shader (Program listing 2.2). Instead, this just computes a colour for the fragment using the uniforms objectColor and lightColor. However, we will update this fragment shader in subsequent examples to make use of fragPos and ourNormal so as to do shading based on the light's position. For now, every face of the cube will be a single colour. fragPos is the position of a fragment in world space (computed by multiplying the vertex position with the model matrix). ourNormal is computed by multiplying the vertex normal by the transpose inverse of the model matrix (which is referred to elsewhere as the 'normal matrix').

The vertex shader for the light (Program listing 2.3) only needs to calculate the position of a vertex and pass this down the pipeline. The fragment shader (Program listing 2.3) colours

the fragment white. Thus the light will appear as a white object. The next example will make changes to the cube's fragment shader.

```
#version 330 core

layout (location = 0) in vec3 position;

uniform mat4 mvpMatrix;

void main() {
  gl_Position = mvpMatrix * vec4(position, 1.0);
}
```

Program listing 2.3: vs_light_01.txt.

```
#version 330 core

out vec4 fragColor;

void main() {
  fragColor = vec4(1.0f);
}
```

Program listing 2.4: fs_light_01.txt.

2.1 Ambient light

The cube is coloured using ambient light. In L01_GLEventListener.initialise(), change the line:

```
shaderCube = new Shader(gl, "vs_cube_01.txt", "fs_cube_01.txt");
```

to

```
shaderCube = new Shader(gl, "vs_cube_01.txt", "fs_cube_01_a mbient.txt");
```

The new fragment shader is given in Program listing 2.5. Essentially, all we have done is multiply the result in Program listing 2.2 by 0.25f.

```
#version 330 core

in vec3 fragPos;
in vec3 ourNormal;

out vec4 fragColor;

uniform vec3 objectColor;
uniform vec3 lightColor;
uniform vec3 lightPos;
uniform vec3 viewPos;

void main() {
  float ambientStrength = 0.25f;
  vec3 ambient = ambientStrength * lightColor;
  vec3 result = ambient * objectColor;
  fragColor = vec4(result, 1.0);
}
```

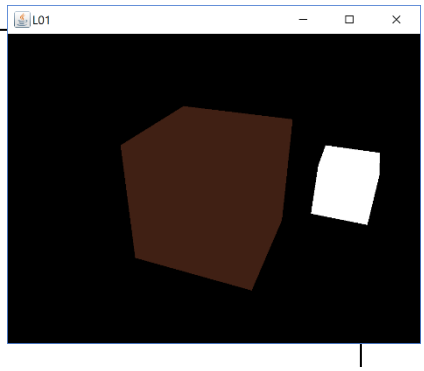

Figure 2.2: Using the ambient shader.

Program listing 2.5: fs_cube_01_ambient.txt.

2.2 Diffuse lighting

In L01_GLEventListener.initialise(), change the line:

```
shaderCube = new Shader(gl, "vs_cube_01.txt", "fs_cube_01.txt");
```

to

```
shaderCube = new Shader(gl, "vs_cube_01.txt", "fs_cube_01_diffuse.txt");
```

The new fragment shader is given in Program listing 2.6. The fragment position in world space (fragPos) and the lighting vector between the light's position and the fragment position (lightDir) are used to calculate the diffuse lighting.

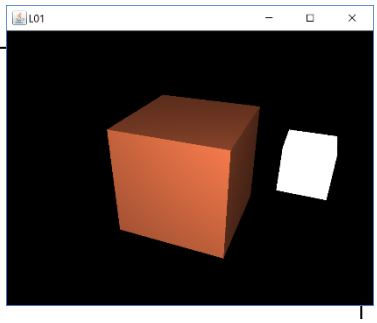

Figure 2.3: Using the diffuse shader.

```
#version 330 core

in vec3 fragPos;
in vec3 ourNormal;

out vec4 fragColor;

uniform vec3 objectColor;
uniform vec3 lightColor;
uniform vec3 lightPos;
uniform vec3 viewPos;

void main() {
  // ambient
  float ambientStrength = 0.1;
  vec3 ambient = ambientStrength * lightColor;

  // diffuse
  vec3 norm = normalize(ourNormal);
  vec3 lightDir = normalize(lightPos - fragPos);
  float diff = max(dot(norm, lightDir), 0.0);
  vec3 diffuse = diff * lightColor;

  vec3 result = (ambient + diffuse) * objectColor;
  fragColor = vec4(result, 1.0);
}
```

Program listing 2.6: fs_cube_01_diffuse.txt.

2.3 Specular lighting

In L01_GLEventListener.initialise(), change the line:

```
shaderCube = new Shader(gl, "vs_cube_01.txt", "fs_cube_01.txt");
```

to

```
shaderCube = new Shader(gl, "vs_cube_01.txt", "fs_cube_01_specular.txt");
```

The new fragment shader is given in Program listing 2.7. The vector from the fragment position to the camera view point is calculated. The mirror reflection vector (reflectDir) is calculated from the light direction and the fragment's normal. These terms are then used to calculate the specular term. We have thus calculated all three components of the Phong equation that was covered in lectures.

Given the current light and view positions, you will not see a specular highlight on the cube.

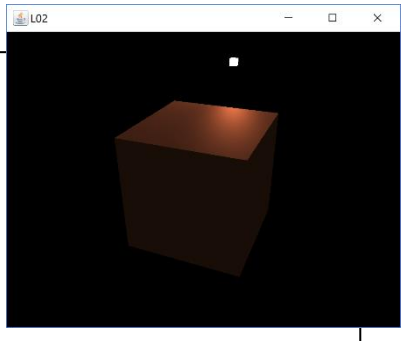

Figure 2.3: Using the specular shader with a rotating light source.

```
#version 330 core

in vec3 fragPos;
in vec3 ourNormal;

out vec4 fragColor;

uniform vec3 objectColor;
uniform vec3 lightColor;
uniform vec3 lightPos;
uniform vec3 viewPos;

void main() {
  // ambient
  float ambientStrength = 0.1;
  vec3 ambient = ambientStrength * lightColor;

  // diffuse
  vec3 norm = normalize(ourNormal);
  vec3 lightDir = normalize(lightPos - fragPos);
  float diff = max(dot(norm, lightDir), 0.0);
  vec3 diffuse = diff * lightColor;

  // specular
  float specularStrength = 0.5;
  vec3 viewDir = normalize(viewPos - fragPos);
  vec3 reflectDir = reflect(-lightDir, norm);
  float spec = pow(max(dot(viewDir, reflectDir), 0.0), 32);
  vec3 specular = specularStrength * spec * lightColor;

  vec3 result = (ambient + diffuse + specular) * objectColor;
  fragColor = vec4(result, 1.0);
}
```

Program listing 2.7: fs_cube_01_specular.txt.

Program: Week5_2_start\L02.java. This program uses a moving light position – the light circles around the world y axis. You will now see the effect of the specular highlight.

Exercise

1. You can alter the light's position by changing the following lines in method L02_GLEventListener. getLightModelMatrix():

   ```
   lightPosition.x = 5.0f*(float)(Math.sin(Math.toRadians(elapsedTime*50)));
   lightPosition.y = 3.0f;
   lightPosition.z = 5.0f*(float)(Math.cos(Math.toRadians(elapsedTime*50)));
   ```

   For example, you can change the vertical position by changing the y coordinate, or alter the radius of the circle that is being used by changing the 5.0f values into something else.

3 Meshes and Materials

Program:
Week5_3_mesh_and_material\L03.java

This program covers the section on Materials in Joey's tutorial. The result is a scene that contains three planes, a cube and a light, as illustrated in Figure 3.1. The planes are each made out of two triangles. Two of the planes are rotated and translate to make the vertical planes shown in Figure 3.1

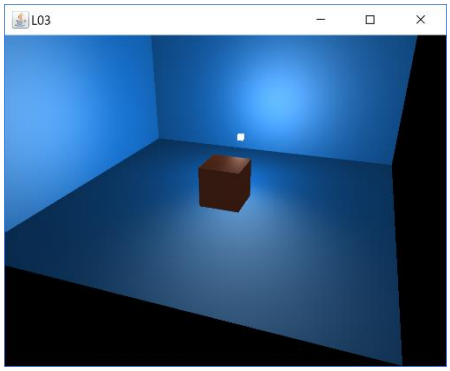

Figure 3.1: Output from Week5_3_mesh_and_material\L03.java.

I've also cleaned up the previous program a little and introduced a general Mesh class. Two types of mesh are included in the program, a Cube mesh and a TwoTriangle mesh. Figure 3.2 shows the structure. A Mesh contains a Material, a Shader and a Mat4 to represent the model matrix. A Cube class extends the Mesh class. ATwoTriangles class extends the Mesh class. The light has also been separated out into a Light class, which repeats some of the contents of a Mesh class, but changes the fillBuffers() method as only vertex position data is required (as described in Section 2 above).

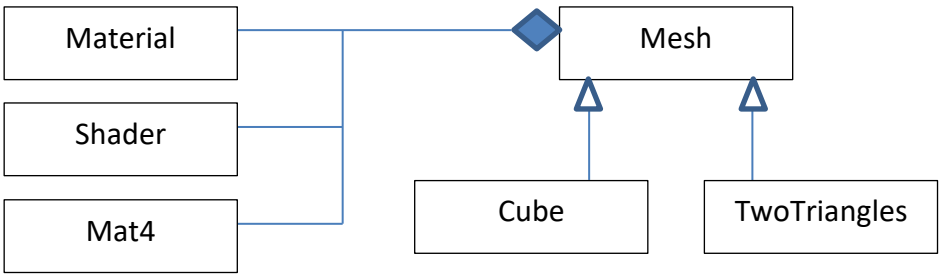

Figure 3.2: The Mesh structure.

The Mesh class takes care of creating a default Material and a default model matrix (stored as a Mat4) and implements the fillBuffers() method which is the same for both a Cube and a TwoTriangles. The render method is declared as abstract in the Mesh class. This method is then implemented in each of Cube and TwoTriangles. The reason for doing this is so that each could then be rendered in a different way. In L03.java the render methods for the two are identical. Later examples will change this.

The differences between Cube and TwoTriangles are in the constructors for each class. They each set slightly different Material properties, as Program listings 3.1 and 3.2 show. The material instance is declared in the Mesh class (as illustrated in Program listing 3.3), which they both extend.

```
  public Cube(GL3 gl) {
    super(gl);
    super.vertices = this.vertices;
    super.indices = this.indices;
    material.setAmbient(1.0f, 0.5f, 0.31f);
    material.setDiffuse(1.0f, 0.5f, 0.31f);
    material.setSpecular(0.5f, 0.5f, 0.5f);
    material.setShininess(32.0f);
    shader = new Shader(gl, "vs_cube_03.txt", "fs_cube_03.txt");
    fillBuffers(gl);
  }
```

Program listing 3.1: The Cube constructor.

```
  public TwoTriangles(GL3 gl) {
    super(gl);
    super.vertices = this.vertices;
    super.indices = this.indices;
    material.setAmbient(0.1f, 0.5f, 0.91f);
    material.setDiffuse(0.1f, 0.5f, 0.91f);
    material.setSpecular(0.3f, 0.3f, 0.3f);
    material.setShininess(4.0f);
    shader = new Shader(gl, "vs_tt_03.txt", "fs_tt_03.txt");
    fillBuffers(gl);
  }
```

Program listing 3.2: The TwoTriangles constructor.

```
public abstract class Mesh {

  protected float[] vertices;
  protected int[] indices;
  private int vertexStride = 8;
  private int vertexXYZFloats = 3;
  private int vertexNormalFloats = 3;
  private int vertexTexFloats = 2;
  private int[] vertexBufferId = new int[1];
  protected int[] vertexArrayId = new int[1];
  private int[] elementBufferId = new int[1];

  protected Material material;
  protected Shader shader;
  protected Mat4 model;

  public Mesh(GL3 gl) {
    material = new Material();
    model = new Mat4(1);
  }

...
```

Program listing 3.3: The start of the Mesh class.

```
...

public class L03_GLEventListener implements GLEventListener {

  private Mesh cube, tt1, tt2;
  private Light light;

  ...

  public void initialise(GL3 gl) {
    cube = new Cube(gl);
    tt1 = new TwoTriangles(gl);
    tt2 = new TwoTriangles(gl);
    light = new Light(gl);
  }

  public void render(GL3 gl) {
    gl.glClear(GL.GL_COLOR_BUFFER_BIT | GL.GL_DEPTH_BUFFER_BIT);

    Mat4 perspective = Mat4Transform.perspective(45, aspect);
    Mat4 view = getViewMatrix();

    light.setPosition(getLightPosition());
    light.render(gl, perspective, view);

    cube.setModelMatrix(getMforCube());
    cube.render(gl, light, viewPosition, perspective, view);

    tt1.setModelMatrix(getMforTT1());
    tt1.render(gl, light, viewPosition, perspective, view);
    tt2.setModelMatrix(getMforTT2());
    tt2.render(gl, light, viewPosition, perspective, view);
    tt1.setModelMatrix(getMforTT3());
    tt1.render(gl, light, viewPosition, perspective, view);
  }

  private Vec3 getLightPosition() {
    double elapsedTime = getSeconds()-startTime;
    float x = 3.0f*(float)(Math.sin(Math.toRadians(elapsedTime*50)));
    float y = 2.4f;
    float z = 3.0f*(float)(Math.cos(Math.toRadians(elapsedTime*50)));
    return new Vec3(x,y,z);
  }

  ...

  private Vec3 viewPosition = new Vec3(4f,6f,15f);

  private Mat4 getViewMatrix() {
    Mat4 view = Mat4Transform.lookAt(viewPosition,
                                     new Vec3(0,0,0), new Vec3(0,1,0));
    return view;
  }
```

Program listing 3.4: Parts of L03_GLEventListener.

We are now ready to look at the contents of L03_GLEventListener.java. Program listing 3.4 shows some of the class attributes and the initialise and render methods. Method initialise() creates new instances of the Cube and TwoTriangles classes. Method render() then makes use of these. A method is included in to get the light's position, which is continually updated to rotate around the y axis, as in Section 2 above. The Light class contains a method to set the light's position. The render method in Light requires the perspective and view matrices. Likewise the Cube class contains a method to set the model matrix for the cube and there is a method in Program listing 3.4 to get a value for this. The method getMforCube() is not shown – you will need to look at the contents of L03_GLEventListener.java.

You may be wondering why the next few lines of code render tt2 once and tt1 twice. I could have used tt1 three times, each time preceding it by a different model matrix. Or, I could have created a third instance of TwoTriangles, called tt3, and used that instead of using tt1 twice. There is a trade-off. Using tt1 multiple times means that only one data buffer needs to be created, but the model matrix must be continually reset to new values before drawing each instance of tt1. If three different instances of TwoTriangles are created, e.g. tt1, tt2, and tt3, then we could set their model matrices once and once only, not every frame. This would save on calculation time, but at the expense of more buffers of data. For the amount of data we are dealing with either approach will suffice. (There are also further possibilities involving packing three sets of data in a single buffer, and then using offsets to access different parts of the buffer. We won't look at this.)

The render method in each of Cube and TwoTriangles makes use of the Material class to set the ambient, diffuse and specular components used in the Phong equation. Program listing 3.5 shows the fragment shader for the Cube. The one for TwoTriangles (fs_tt_03.txt) is the same, but will change in the next section.

```
#version 330 core

in vec3 fragPos;
in vec3 ourNormal;
in vec2 ourTexCoord;

out vec4 fragColor;

uniform vec3 viewPos;

struct Light {
  vec3 position;
  vec3 ambient;
  vec3 diffuse;
  vec3 specular;
};

uniform Light light;

struct Material {
  vec3 ambient;
  vec3 diffuse;
  vec3 specular;
  float shininess;
};

uniform Material material;

void main() {
  // ambient
  vec3 ambient = light.ambient * material.ambient;

  // diffuse
  vec3 norm = normalize(ourNormal);
  vec3 lightDir = normalize(light.position - fragPos);
  float diff = max(dot(norm, lightDir), 0.0);
  vec3 diffuse = light.diffuse * (diff * material.diffuse);

  // specular
  vec3 viewDir = normalize(viewPos - fragPos);
  vec3 reflectDir = reflect(-lightDir, norm);
  float spec = pow(max(dot(viewDir, reflectDir), 0.0), material.shininess);
  vec3 specular = light.specular * (spec * material.specular);

  vec3 result = ambient + diffuse + specular;
  fragColor = vec4(result, 1.0);
}
```

Program listing 3.5: fs_cube_03.txt.

Program listing 3.5 is similar to Program listing 2.7, only it makes use of a Struct to collect together the different material properties for the cube and another to collect together the material properties for the light. This is described in Joey's tutorial, so I won't repeat it here.

Exercises

1. Experiment with changing the material properties of the Cube instance. This is done in the constructor for the Cube.

2. Experiment with changing the material properties of the instances of TwoTriangles and the Light instance.
3. There is a commented out line in the render method "//updateLightColour();" Uncomment this and then explain what happens.

4 Light maps

Program: Week5_4_light_maps\L04.java

This follows Joey's tutorial fairly closely. Figure 4.1 shows the output from the program. I've decreased the diffuse component in the Light class a little to make the specular effect clearer.

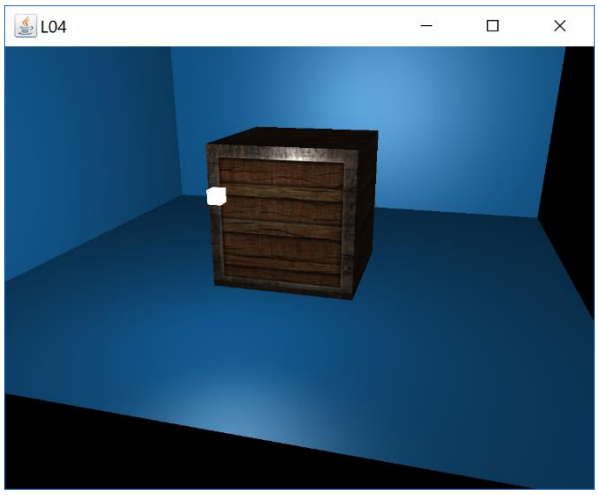

Figure 4.1: Output from Week5_4_light_maps\L04.java.

The Cube class is changed to accept two textures to represent the diffuse texture and the specular texture. We have to be careful when we are using multiple texture maps when we have split the program across multiple classes. There are limits to the number of texture image units, although we are unlikely to breach these in our programs[1]. The Cube class makes use of texture units 0 and 1. Program listing 4.1 shows the method that creates the Cube instance. Program listing 4.2 shows the Cube class.

```
...

  public void initialise(GL3 gl) {
    createRandomNumbers();
    int[] textureId0 = TextureLibrary.loadTexture(gl, "container2.jpg");
    int[] textureId1 = TextureLibrary.loadTexture(gl, "container2_specular.jpg");
    cube = new Cube(gl, textureId0, textureId1);
    tt1 = new TwoTriangles(gl);
    tt2 = new TwoTriangles(gl);
    light = new Light(gl);
  }

...
```

Program listing 4.1: L04_GLEventListener.initialise().

[1] GL_MAX_COMBINED_TEXT_IMAGE_UNITS is the maximum number of texture units allowed and GL_MAX_TEXTURE IMAGE_UNITS is the maximum number of texture units accessible from one fragment shader. This website has a database of reported values across alot of GPUs, http://feedback.wildfiregames.com/report/opengl

```
public class Cube extends Mesh {
  private int[] textureId1;
  private int[] textureId2;

  public Cube(GL3 gl, int[] textureId1, int[] textureId2) {
    super(gl);
    super.vertices = this.vertices;
    super.indices = this.indices;
    this.textureId1 = textureId1;
    this.textureId2 = textureId2;
    material.setAmbient(1.0f, 0.5f, 0.31f);
    material.setDiffuse(1.0f, 0.5f, 0.31f);
    material.setSpecular(0.5f, 0.5f, 0.5f);
    material.setShininess(32.0f);
    shader = new Shader(gl, "vs_cube_04.txt", "fs_cube_04.txt");
    fillBuffers(gl);
  }

  public void render(GL3 gl, Light light, Vec3 viewPosition, Mat4 perspective, Mat4 view) {
    Mat4 mvpMatrix = Mat4.multiply(perspective, Mat4.multiply(view, model));

    shader.use(gl);
    shader.setFloatArray(gl, "model", model.toFloatArrayForGLSL());
    shader.setFloatArray(gl, "mvpMatrix", mvpMatrix.toFloatArrayForGLSL());

    shader.setVec3(gl, "viewPos", viewPosition);

    shader.setVec3(gl, "light.position", light.getPosition());
    shader.setVec3(gl, "light.ambient", light.getMaterial().getAmbient());
    shader.setVec3(gl, "light.diffuse", light.getMaterial().getDiffuse());
    shader.setVec3(gl, "light.specular", light.getMaterial().getSpecular());

    // shader.setVec3(gl, "material.ambient", material.getAmbient());
    // shader.setVec3(gl, "material.diffuse", material.getDiffuse());
    // shader.setVec3(gl, "material.specular", material.getSpecular());
    shader.setFloat(gl, "material.shininess", material.getShininess());

    shader.setInt(gl, "material.diffuse", 0);
    shader.setInt(gl, "material.specular", 1);

    gl.glActiveTexture(GL.GL_TEXTURE0);
    gl.glBindTexture(GL.GL_TEXTURE_2D, textureId1[0]);
    gl.glActiveTexture(GL.GL_TEXTURE1);
    gl.glBindTexture(GL.GL_TEXTURE_2D, textureId2[0]);

    gl.glBindVertexArray(vertexArrayId[0]);
    gl.glDrawElements(GL.GL_TRIANGLES, indices.length, GL.GL_UNSIGNED_INT, 0);
    gl.glBindVertexArray(0);
  }

...
```

Program listing 4.2: Part of Cube.java. (Notice that most of the material properties are now commented out, since the texture maps are used to supply this.)

5 Adding a Camera and also some textures on other surfaces

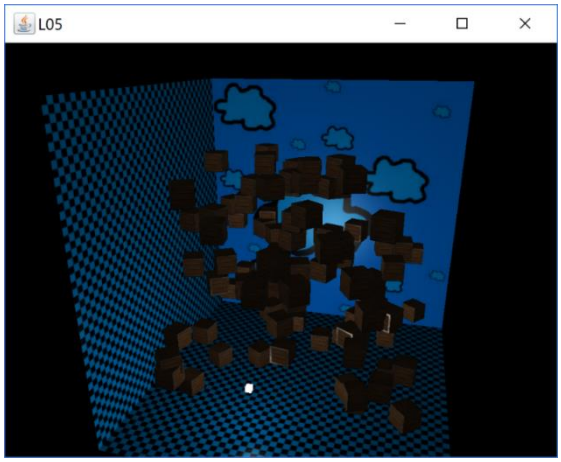

Figure 5.1: Output from Week5_5_with_camera\L05.java.

Program: Week5_5_with_camera\L05.java

The output is illustrated in Figure 5.1. A texture has been included in the constructor for the TwoTriangles class. The back wall uses one texture (cloud.jpg) and the floor and side wall use a different texture (chequerboard.jpg). Notice how the chequerboard still looks good at an angle even when the texture starts to compress further back into the perspective view. This is because the TextureLibrary class has turned on mip-mapping for the textures that it loads – we'll cover mip-mapping in lectures.

You should examine the program in detail (L05_GLEventListener.initialise() and the classes for Cube and TwoTriangles) to see how the texture units are used in rendering. Texture units 0 and 1 are used for the diffuse and specular maps on the cube, as in Section 4, texture unit 0 is used for the chequerboard on the floor and side wall and texture unit 0 is used for the cloud texture on the back wall. This is possible because the textures are loaded into the GPU, with links to these memory areas stored in variables: `textureId0`, `textureId1`, `textureId2` and `textureId3`. Inside the Cube and TwoTriangles classes these are then bound to specific texture units, e.g. GL_TEXTURE0 and GL_TEXTURE1. The analogy would be real pictures (`textureId0`, `textureId1`, `textureId2` and `textureId3`) which can be bound to a minimum of 16 picture frames (GL_TEXTURE0, GL_TEXTURE1, etc.).

Exercises

1. The cubes in Figure 5.1 are slowly rotating over time. Examine the method Lo5_GLEventListener.getModelMatrix(). Change elapsedTime*10 to elapsedTime*100 to speed up the rotation.
2. Try adding a different texture to the side wall.

6 Further thoughts

Joey's tutorial goes on to discuss light casters and multiple lights. These are left as an exercise.

So far we have not concerned ourselves with running out of resources, e.g. too many textures or too much vertex data. This is because we are unlikely to breach these limits in the examples we are looking at. However, you should be aware that, depending on the specific GPU, there are limits that need to be addressed in commercial systems. As an example, there are shader resource limitations: see https://www.khronos.org/opengl/wiki/Shader. Another example is the limit for the maximum number of uniforms in a fragment shader: GL_MAX_FRAGMENT_UNIFORM_BLOCKS. Documents say a minimum of 12 (https://www.khronos.org/registry/OpenGL-Refpages/es3.0/html/glGet.xhtml). If you use a lot of uniforms, you could hit this limit. Uniform buffer objects would then be required.

The next exercise sheet will look at more examples of objects beyond a cube and a two-triangle plane. We'll also look at scene graphs.