# COM3503/4503/6503: 3D Computer Graphics

Dr Steve Maddock

# Exercise sheet 4: Transformations, Coordinate systems and a camera

You should read through Joey de Vries's tutorial sections on 'Transformations', 'Coordinate Systems', and 'Camera': https://learnopengl.com/#!Getting-started/Transformations.

## 1 Introduction

We have already covered the material in this week's exercise sheet in lectures. Again, I'm going to loosely follow Joey's tutorials.

## 2 Coordinate Systems

Many of the examples in Joey's tutorial and in other online tutorials on OpenGL make use of the 'glm' maths library, which includes classes for Vec2, Vec3, Vec4, Mat4, etc. There have been some efforts at producing a Java version of this: try searching for Java and glm and you'll find a few examples on github. For example, JOML appears to be used in a number of universities. However, these are more than we need. I have created a simple maths library that covers some of what glm includes and will suffice for what we need. This is supplied as a package called gmaths. This should be copied into your working folder for your JOGL programs and can then be imported into your programs using:

```
import gmaths.*;
```

at the top of any relevant Java file.

It includes classes for Vec2, Vec3 and Vec4, which are a 2D vector, a 3D vector and a 4D vector, respectively. I have also included a class called Mat4 which is a 4x4 matrix, and a class called Mat4Transform, which is used to set up translate, scale and three rotate matrices, one for each axis, as well as including methods to set up a view matrix and a perspective matrix. (Note: the parameters for my rotate methods in Mat4Transform use angles in degrees, whereas the Java Math.sin(), Math.cos(), etc. require radians. Inside the rotate methods, I convert the supplied parameters into radians before use with Math.sin() and Math.cos().) Program 2.1 gives an example of using the 2D vector class, Vec2, showing some of the methods available. You should examine the methods in each of the classes in the gmaths package before continuing further.

You will need to compile the package before using it. Make sure the gmaths folder is in the folder you are working in. Then, type:

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

This will produce the package and you can now import it into your programs. (On a Mac you may have to use javac gmaths/*.java.)

To produce some java documentation for the classes in the gmaths package, type:

```
javadoc gmaths\*.java
```

This will create an index.html file in the current folder. Opening this will show the documentation. I have only documented the Vec2 class and the Mat4Transform class, but this should be enough to explain how things are working.

Before proceeding, make sure you read through Joey's tutorial: https://learnopengl.com/#!Getting-started/Transformations. This tutorial explains some of the types available in GLSL, in particular vec3 and mat4, which, unsurprisingly, are similar to the C functions available in glm. We'll make use of mat4 in a later program, so do read through Joey's tutorial.

```java
import gmaths.*;

public class TestVec2b {

  public static void main(String[] args) {
    Vec2 v1 = new Vec2();
    Vec2 v2 = new Vec2();
    System.out.println("Vec2 v1 = " + v1);
    System.out.println("Vec2 v2 = " + v2);

    v1.x = 1;
    v2.y = 1;
    System.out.println("v1 = " + v1);
    System.out.println("v2 = " + v2);

    v1.add(v2);
    System.out.println("v1.add(v2);");
    System.out.println("v1 = " + v1);
    System.out.println("v2 = " + v2);

    Vec2 v3 = Vec2.add(new Vec2(-1f,0), v2);
    System.out.println("v3 = v1+(-1f,0) = " + v3);

    float m = v3.magnitude();
    System.out.println("v3.magnitude = " + m);

    Vec2 v4 = Vec2.normalize(v3);
    System.out.println("v4 = normalize(v3) = " + v4);

    v3.normalize();
    System.out.println("v3.normalize();");
    System.out.println("v3 = " + v3);

    Vec2 v5 = new Vec2(3.2f,5f);
    float d = v4.dotProduct(v5);
    System.out.println("v5 = new Vec2(3.2f,5f) = " + v5);
    System.out.println("d = v4.dotProduct(v5) = " + d);
  }
}
```

**Program listing 2.1:** Testing the Vec2 class in the gmaths package.

# 3 Coordinate Systems

Joey's tutorial https://learnopengl.com/#!Getting-started/Coordinate-Systems explains what we covered in the lecture on 'The viewing pipeline'. Make sure you read through Joey's tutorial before proceeding.

Whilst Joey composes a model view projection matrix in the vertex shader, we're going to do it on the CPU instead and then pass this to the vertex shader using a uniform. The reason for doing this is so that we only compose it once per frame rather than once per vertex. (Think carefully about that last sentence.)

**Program:** V01.java

Program Listing 3.1 lists the data structure and Program Listing 3.2 gives the render method. Two triangles are drawn, and texture mapped with the front cover of Watt's book, as shown in Figure 3.1. The two triangles have been rotated away from the viewer and are thus seen in perspective. We'll now examine Program Listing 3.2 in more detail.

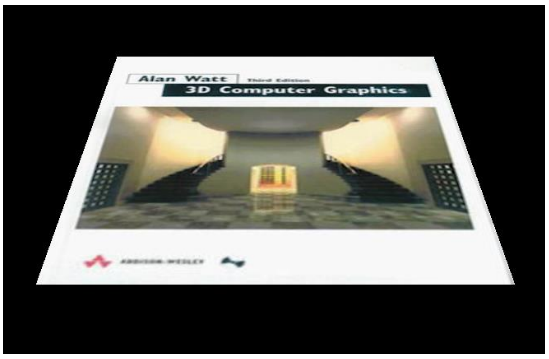

**Figure 3.1:** Two triangles shown in perspective, texture mapped with the front cover of Watt's book.

```
// ***************************************************
/* THE DATA
 */
// anticlockwise/counterclockwise ordering
private float[] vertices = {       // position, colour, tex coords
  -0.5f,  0.5f, 0.0f,   1.0f, 0.0f, 0.0f,   0.0f, 1.0f,   // top left
  -0.5f, -0.5f, 0.0f,   0.0f, 1.0f, 0.0f,   0.0f, 0.0f,   // bottom left
   0.5f, -0.5f, 0.0f,   0.0f, 0.0f, 1.0f,   1.0f, 0.0f,   // bottom right
   0.5f,  0.5f, 0.0f,   1.0f, 1.0f, 1.0f,   1.0f, 1.0f    // top right
};

private int vertexStride = 8;
private int vertexXYZFloats = 3;
private int vertexColourFloats = 3;
private int vertexTexFloats = 2;

private int[] indices = {          // Note that we start from 0!
    0, 1, 2,
    0, 2, 3
};
```

**Program listing 3.1:** The data.

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

  double elapsedTime = getSeconds()-startTime;

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

  float zposition = 2f
  //float zposition = 2f+(float)(Math.sin(Math.toRadians(elapsedTime*50)));
  Vec3 position = new Vec3(0,0,zposition);
  Mat4 view = Mat4Transform.lookAt(position, new Vec3(0,0,0), new Vec3(0,1,0));

  float angle = -55f;
  //float angle = (float)(-115*Math.sin(Math.toRadians(elapsedTime*50)));
  Mat4 model = Mat4Transform.rotateAroundX(angle);

  Mat4 mvpMatrix = Mat4.multiply(view, model);
  mvpMatrix = Mat4.multiply(perspective, mvpMatrix);

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

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

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

**Program listing 3.2:** The render() method.

A perspective matrix is created using the static method in Mat4Transform. The static method Mat4Transform.lookat() is then used to create the view matrix. This takes three parameters: the camera position, the target that the camera is looking at, and a nominal up vector for the world; in this example the camera is at position (0,0,10), looking at the world origin, with an initial up vector (0,1,0). The Gram–Schmidt process (as described in lectures) is used to create a coordinate frame for the view (i.e. the camera).

An angle of -55 is chosen to rotate the two triangles around the x axis, which will rotate the top away from the viewer. The static method Mat4Transform.rotateAroundX() is used to set up the required model transformation matrix.

In the next step, the model view projection matrix is calculated. Remember that the matrices precede the vertices they are transforming. Thus they are composed in the order perspective, view, model, so that the model matrix is the first applied to a vertex.

The mvpMatrix is then passed to the vertex shader using a uniform. In order to do thisthe 4x4 matrix must first be converted into an array of floats. The method Mat4.toFloatArrayForGLSL() is used to do this. It converts the row-column ordered matrix into a column-row ordered matrix stored in an array of floats which is what is required in the data type used in the vertex shader. A new method has also been added to the shader class to set the relevant uniform in the vertex shader.

There are two commented out lines in the render method. These are for experimentation in the exercises below.

```
#version 330 core

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

out vec3 ourColor;
out vec2 ourTexCoord;

uniform mat4 mvpMatrix;

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

**Program listing 3.3:** v01_vs.txt.

The next thing to look at is the vertex shader, which is given in Program Listing 3.3. Most of this should be familiar to you. The new bit is the uniform mat4 mvpMatrix, which contains the model view projection matrix sent from the main program. This is used to multiply the position of a vertex to convert it into 3D screen space (or clip space as it is also known). Each instance of the vertex shader does the same thing on different vertices. All the vertices are sent to the next stage of the pipeline, where they are connected back together to make triangles and are then rasterised to produce a set of fragments which are handled by the fragment shader. The fragment shader does not need to change from our previous programs. It just works out the colour of a fragment using the relevant texture coordinate. The colour attribute that we have used in the triangle data structure is ignored.

We now have a working 3D world.

## Exercises

1. Uncomment each of the lines in the render method (corresponding to zposition and angle) and see what effect they each have.
2. Experiment with each of the rotate methods available in Mat4Transform for the model matrix, i.e. Mat4Transform.rotateAroundX(float angle), Mat4Transform.rotateAroundY(float angle), Mat4Transform.rotateAroundZ(float angle).

**Program:** V02.java

Next, we're going to replace the data for two triangles with a cube. Program listing 3.4 gives the data. This may look slightly daunting, but it is just an extension of the data for two triangles, with two triangles on each face of the cube. (The colour data is actually superfluous, as we are not currently using it. However, it can be used in the fragment shader, so is left for now.) Each face of the cube has texture coordinates in the range 0.0,0.0 to 1.0,1.0, so the Watt book cover is mapped to each face of the cube, as illustrated in Figure 3.2.

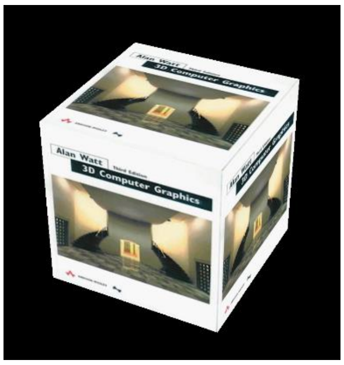

**Figure 3.2:** The front cover of Watt's book is mapped onto each face of a cube.

```
// ***************************************************
/* THE DATA
 */
// anticlockwise/counterclockwise ordering

 private float[] vertices = new float[] {  // x,y,z, colour, s,t
    -0.5f, -0.5f, -0.5f,  1.0f, 0.0f, 0.0f,  0.0f, 0.0f,  // 0
    -0.5f, -0.5f,  0.5f,  1.0f, 0.0f, 0.0f,  1.0f, 0.0f,  // 1
    -0.5f,  0.5f, -0.5f,  1.0f, 0.0f, 0.0f,  0.0f, 1.0f,  // 2
    -0.5f,  0.5f,  0.5f,  1.0f, 0.0f, 0.0f,  1.0f, 1.0f,  // 3
     0.5f, -0.5f, -0.5f,  1.0f, 0.0f, 0.0f,  1.0f, 0.0f,  // 4
     0.5f, -0.5f,  0.5f,  1.0f, 0.0f, 0.0f,  0.0f, 0.0f,  // 5
     0.5f,  0.5f, -0.5f,  1.0f, 0.0f, 0.0f,  1.0f, 1.0f,  // 6
     0.5f,  0.5f,  0.5f,  1.0f, 0.0f, 0.0f,  0.0f, 1.0f,  // 7

    -0.5f, -0.5f, -0.5f,  0.0f, 1.0f, 0.0f,  1.0f, 0.0f,  // 8
    -0.5f, -0.5f,  0.5f,  0.0f, 1.0f, 0.0f,  0.0f, 0.0f,  // 9
    -0.5f,  0.5f, -0.5f,  0.0f, 1.0f, 0.0f,  1.0f, 1.0f,  // 10
    -0.5f,  0.5f,  0.5f,  0.0f, 1.0f, 0.0f,  0.0f, 1.0f,  // 11
     0.5f, -0.5f, -0.5f,  0.0f, 1.0f, 0.0f,  0.0f, 0.0f,  // 12
     0.5f, -0.5f,  0.5f,  0.0f, 1.0f, 0.0f,  1.0f, 0.0f,  // 13
     0.5f,  0.5f, -0.5f,  0.0f, 1.0f, 0.0f,  0.0f, 1.0f,  // 14
     0.5f,  0.5f,  0.5f,  0.0f, 1.0f, 0.0f,  1.0f, 1.0f,  // 15

    -0.5f, -0.5f, -0.5f,  0.0f, 0.0f, 1.0f,  0.0f, 0.0f,  // 16
    -0.5f, -0.5f,  0.5f,  0.0f, 0.0f, 1.0f,  0.0f, 1.0f,  // 17
    -0.5f,  0.5f, -0.5f,  0.0f, 0.0f, 1.0f,  0.0f, 1.0f,  // 18
    -0.5f,  0.5f,  0.5f,  0.0f, 0.0f, 1.0f,  0.0f, 0.0f,  // 19
     0.5f, -0.5f, -0.5f,  0.0f, 0.0f, 1.0f,  1.0f, 0.0f,  // 20
     0.5f, -0.5f,  0.5f,  0.0f, 0.0f, 1.0f,  1.0f, 1.0f,  // 21
     0.5f,  0.5f, -0.5f,  0.0f, 0.0f, 1.0f,  1.0f, 1.0f,  // 22
     0.5f,  0.5f,  0.5f,  0.0f, 0.0f, 1.0f,  1.0f, 0.0f   // 23
   };

   private int[] indices =  new int[] {
    0,1,3, // x -ve
    3,2,0, // x -ve
    4,6,7, // x +ve
    7,5,4, // x +ve
    9,13,15, // z +ve
    15,11,9, // z +ve
    8,10,14, // z -ve
    14,12,8, // z -ve
    16,20,21, // y -ve
    21,17,16, // y -ve
    23,22,18, // y +ve
    18,19,23  // y +ve
  };

private int vertexStride = 8;
private int vertexXYZFloats = 3;
private int vertexColourFloats = 3;
private int vertexTexFloats = 2;
```

**Program listing 3.4:** The data for a cube.

```
public void render(GL3 gl) {
  gl.glClear(GL.GL_COLOR_BUFFER_BIT | GL.GL_DEPTH_BUFFER_BIT);
  shader.use(gl);
  Mat4 perspective = Mat4Transform.perspective(45, aspect);
  Mat4 view = getViewMatrix();
  Mat4 model = getModelMatrix();
  Mat4 mvpMatrix = Mat4.multiply(perspective, Mat4.multiply(view, model));
  shader.setFloatArray(gl, "mvpMatrix", mvpMatrix.toFloatArrayForGLSL());
  gl.glActiveTexture(GL.GL_TEXTURE0);
  gl.glBindTexture(GL.GL_TEXTURE_2D, textureId1[0]);
  gl.glBindVertexArray(vertexArrayId[0]);
  gl.glDrawElements(GL.GL_TRIANGLES, indices.length, GL.GL_UNSIGNED_INT, 0);
  gl.glBindVertexArray(0);
}

private Mat4 getModelMatrix() {
  double elapsedTime = getSeconds()-startTime;
  //float angle = -55;
  //float angle = (float)(-115*Math.sin(Math.toRadians(elapsedTime*50)));
  Mat4 model = new Mat4(1);
  //model = Mat4.multiply(Mat4Transform.rotateAroundY(angle), model);
  //model = Mat4.multiply(Mat4Transform.rotateAroundX(angle), model);
  return model;
}

private Mat4 getViewMatrix() {
  double elapsedTime = getSeconds()-startTime;
  float xposition = 2;
  float yposition = 3;
  float zposition = 4;
  //float xposition = 3.0f*(float)(Math.sin(Math.toRadians(elapsedTime*50)));
  //float zposition = 3.0f*(float)(Math.cos(Math.toRadians(elapsedTime*50)));
  Mat4 view = Mat4Transform.lookAt(new Vec3(xposition,yposition,zposition),
              new Vec3(0,0,0), new Vec3(0,1,0));
  return view;
}
```

**Program listing 3.5:** Rendering the cube.

Program listing 3.5 gives the render method and accompanying methods, used to prevent the render method becoming cluttered. getModelMatrix() returns the model transform matrix. Here, it simply returns the identity matrix. getViewMatrix() sets the camera at position (2,3,4), looking at the origin (0,0,0), with a nominal up vector of (0,1,0).

## Exercises

1. Start by experimenting with the commented out lines in getViewMatrix(). These will rotate the camera position around the world y axis, so that the camera is always looking at the world origin. With only a single object centred at the world origin on screen, this can instead look like the object is rotating, rather than the camera rotating.
2. Set the view position back to the fixed values of 2,3,4 before changing getModelMatrix(). Now comment out some of the lines in getModelMatrix(). Try to guess the effect before commenting out the lines.

Joey's tutorial on coordinate systems discusses the z buffer.

We have already enabled the depth test in the initialise method, so we don't get the problematic depth effect he describes.

**Program:** V03.java. Program listing 3.5 uses a nested for loop to draw a grid of cubes, as illustrated in Figure 3.3. Figure 3.4[1] shows the texture used in Figure 3.3. The texture coordinates used in the cube data structure have been changed so that each face uses a part of the texture. You should examine this data structure in V03_GLEventListener.java.

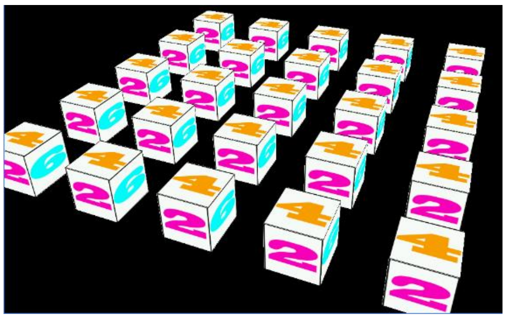

**Figure 3.3:** A grid of cubes.

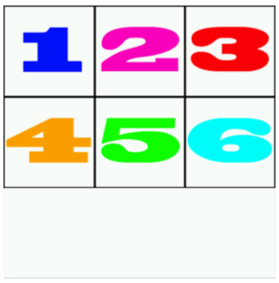

**Figure 3.4:** The texture used for Figure 3.3.

```
public void render(GL3 gl) {
  gl.glClear(GL.GL_COLOR_BUFFER_BIT | GL.GL_DEPTH_BUFFER_BIT);
  shader.use(gl);
  Mat4 perspective = Mat4Transform.perspective(45, aspect);
  Mat4 view = getViewMatrix();
  gl.glActiveTexture(GL.GL_TEXTURE0);
  gl.glBindTexture(GL.GL_TEXTURE_2D, textureId1[0]);

  for (int i=-2; i<3; ++i) {
    for (int j=-2; j<3; ++j) {
      Mat4 model = getModelMatrix(2f*i, 2f*j);
      Mat4 mvpMatrix = Mat4.multiply(perspective, Mat4.multiply(view, model));
      shader.setFloatArray(gl, "mvpMatrix", mvpMatrix.toFloatArrayForGLSL());
      gl.glBindVertexArray(vertexArrayId[0]);
      gl.glDrawElements(GL.GL_TRIANGLES, indices.length, GL.GL_UNSIGNED_INT, 0);
      gl.glBindVertexArray(0);
    }
  }
}

private Mat4 getModelMatrix(float i, float j) {
  double elapsedTime = getSeconds()-startTime;
  float angle = (float)(elapsedTime*100);
  Mat4 model = new Mat4(1);
  model = Mat4.multiply(model, Mat4Transform.translate(i, 0, j));
  model = Mat4.multiply(model, Mat4Transform.rotateAroundY(angle));
  return model;
}

private Mat4 getViewMatrix() {
  double elapsedTime = getSeconds()-startTime;
  Vec3 pos = new Vec3(4,6,10);
  Mat4 view = Mat4Transform.lookAt(pos, new Vec3(0,0,0), new Vec3(0,1,0));
  return view;
}
```

**Program listing 3.4:** Rendering a 5x5 grid of cubes.

[1] The texture is from http://www.opengl-tutorial.org/beginners-tutorials/tutorial-5-a-textured-cube/ – available under Creative-Common license: CC-BY-NC-ND.

## Exercises

In each of these exercises, try to guess what happens before you make the changes.

1. What happens if you reverse the order of the two lines:
   model = Mat4.multiply(model, Mat4Transform.translate(i, 0, j));
   model = Mat4.multiply(model, Mat4Transform.rotateAroundY(angle));
2. What happens if you add an extra line, so that the transforms become:
   model = Mat4.multiply(model, Mat4Transform.rotateAroundY(angle));
   model = Mat4.multiply(model, Mat4Transform.translate(i, 0, j));
   model = Mat4.multiply(model, Mat4Transform.rotateAroundX(angle));
3. Try using different transformations on different cubes. You'll need to use an if test, e.g. if (i%3 == 0) would select cubes 0, 3, 6, 9, etc.

**Program:** V04.java

Program listing 3.6 gives the render method for this program. Figure 3.5 shows the output. This program is similar to V03.java, with one main difference. The cubes are positioned at random positions, rather than as part of a regular grid. The same random positions have to be generated every time the scene is rendered. This is done using a global array of random numbers created once when the program is run. Program listings 3.7 and 3.8 give further details.

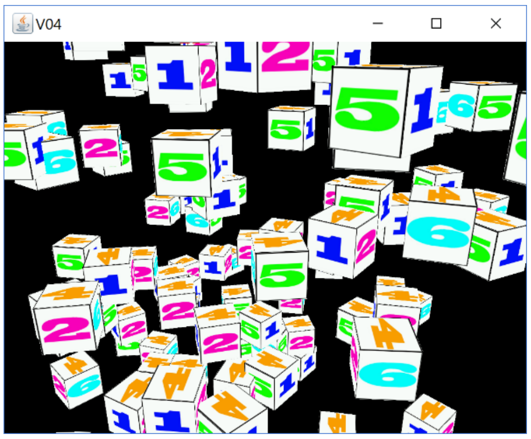

**Figure 3.5:** 100 cubes randomly positioned.

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

    shader.use(gl);

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

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

    for (int i=0; i<100; ++i) {
      Mat4 model = getModelMatrix(i);    // get a random position
      Mat4 mvpMatrix = Mat4.multiply(perspective, Mat4.multiply(view, model));
      shader.setFloatArray(gl, "mvpMatrix", mvpMatrix.toFloatArrayForGLSL());

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

**Program listing 3.6:** The render method.

```
private Mat4 getModelMatrix(int i) {
  double elapsedTime = getSeconds()-startTime;
  Mat4 model = new Mat4(1);
  float yAngle = (float)(elapsedTime*100*randoms[(i+637)%NUM_RANDOMS]);
  float multiplier = 12.0f;
  float x = multiplier*randoms[i%NUM_RANDOMS] - multiplier*0.5f;
  float y = multiplier*randoms[(i+137)%NUM_RANDOMS] - multiplier*0.5f;
  float z = multiplier*randoms[(i+563)%NUM_RANDOMS] - multiplier*0.5f;
  model = Mat4.multiply(model, Mat4Transform.translate(x,y,z));
  model = Mat4.multiply(model, Mat4Transform.rotateAroundY(yAngle));
  return model;
}

private Mat4 getViewMatrix() {
  double elapsedTime = getSeconds()-startTime;
  float xposition = 4;
  float yposition = 5;
  float zposition = 12;
  Mat4 view = Mat4Transform.lookAt(new Vec3(xposition,yposition,zposition),
                new Vec3(0,0,0), new Vec3(0,1,0));
  return view;
}
```

**Program listing 3.7:** Helper methods for the render method.

```
// ***************************************************
/* An array of random numbers
 */

private int NUM_RANDOMS = 1000;
private float[] randoms;

private void createRandomNumbers() {
  randoms = new float[NUM_RANDOMS];
  for (int i=0; i<NUM_RANDOMS; ++i) {
    randoms[i] = (float)Math.random();
  }
}

public void initialise(GL3 gl) {
  //...
  createRandomNumbers();
}
```

**Program listing 3.8:** An array of random numbers.

Program listing 3.7 gives the helper methods for method render(). getModelMatrix() makes use of a global array of random numbers, so that the same values are used every time the method is called for a particular value of the parameter i. The array of random numbers is initialised once in the method initialise().

## 4 Camera

**Program:** V05.java

This program creates a camera that is under mouse and keyboard control. This is similar to the camera described in Joey's tutorial: https://learnopengl.com/#!Getting-started/Camera. When you run the program, you will be able to fly through the scene of 100 cubes.

An instance of the Camera class is created in V05.java. V05.java also contains classes for Keyboard input and Mouse input, each of which can set attributes in the Camera class based on user input. Holding down the left button and moving the mouse will move the direction that the camera is looking in. The A and Z keys can be used to move in and out of the scene in the direction the camera is looking in. The arrow keys can be used to move up, down, left and right.

The Camera instance is passed to the V05_GLEventListener constructor. Thereafter the render method in this class uses the Camera instance to retrieve the current view matrix. The rest of the render method is the same as previous programs.

Take some time to read through Joey's tutorial and look at the new classes that I have created. It is not necessary that you understand the full details to be able to use the Camera class. You can just use it as it is in your programs. We will use it on subsequent exercise sheets and it can also be used in the assignment.

## 5 Further thoughts

There are a number of inefficiencies in the programs we have looked at so far. A lot of the data is reset in each frame, even if it has not changed. However, this allows me to keep all the relevant code close together for you to study it. Longer term, we need a better data structure to store information in. We also need to consider lighting and shading of objects.

**Lighting and shading**

So far, the scenes we have created do not have lights in them, nor do the objects consider those lights and shade the surfaces accordingly. Exercise sheet 5 will look at lighting and shading.

**Multiple different objects**

So far, we have considered a single object type in each example, e.g. a cube in the last few examples. We have not mixed different kinds of object. Exercise sheet 6 will consider this. A Mesh class will be developed, which will be used to represent different kinds of objects, e.g. cube, sphere, cylinder, etc. The Mesh class will feature a render method that can be used to render each specific Mesh instance. Thus the single render method we have used in examples so far will then make use of a separate render method for each Mesh instance. Relevant model transformations can then be included with each Mesh instance and a scene can be built.