Computer Science
Aston University
Birmingham B4 7ET
http://www.cs.aston.ac.uk/

Computer Graphics Lab Notes: OpenGL (in Java)

Dan Cornford, Remi Barillec

Module CS2150 September 29, 2011

Contents

Introduction 2

Lab 1: How to use OpenGL and Java; spatial awareness 4

Lab 2: Appearance in OpenGL 12

Lab 3: Construction in OpenGL 15

Lab 4: Lighting and materials in OpenGL 21

Lab 5: Quadrics and textures in OpenGL 25

Lab 6: Animation in OpenGL 29

Extra labs: useful examples in OpenGL 32

Summary 34

Introduction

This series of exercises teaches you how to implement graphics using the OpenGL API, in Java. You will learn largely by following and then extending examples that we have written for you. We will be using the Eclipse IDE for Java. The great functionality offered by Eclipse can be daunting at first, but you will get familiar with it as we go along.

It is highly recommended that you attempt the labs **before** the lab class, so that the lab time can be spent answering questions. The lab exercises build up in complexity, so you will want to attempt them in the correct order.

As you will appreciate, this is *not a programming module* – the aim is to use OpenGL, and programming using OpenGL, to reinforce your understanding of the material covered in the course and to give you skills as graphics programmers. This will also provide you with more experience of programming in Java (which may well be of benefit when you go on placement or when you are looking for employment).

OpenGL has bindings to just about every language that is used in computing. We'll use the Java bindings provided by the *Lightweight Java Game Library (LWJGL)*, see `http://lwjgl.org/`, but there are bindings to C and C++ as well as Visual Basic and many others. While you can use OpenGL with other languages I will expect coursework to be submitted written in Java. Installing the labs on the your computer at home is very simple; instructions can be found on the wiki (`http://wiki.aston.ac.uk/CS2150/LabSoftware`).

Scene Graphs

A scene graph is a hierarchical structure, commonly a tree, that represents the logical relationships between objects in a given virtual world. In the labs we will make use of simplified scene graphs to describe the three dimensional virtual worlds that are rendered by the lab code examples. Later in the course you will learn how scene graphs can be used to help you write your coursework.

Our simplified scene graphs will consist of two element types: nodes representing scene objects, and arcs representing spatial relationships between objects. Examples of scene objects that you will encounter in the labs include three dimensional objects such as houses, planets, and the various parts of a person.

In a given virtual world, spatial relationships exist between parent and child scene objects. Example spatial relationships include relative positioning (for example, a given child object is below its parent object), relative orientation (for example, a given child object is oriented 90 degrees clockwise to its parent), relative scaling (for example, a given child object is half the size of its parent), plus combinations of these. We will use OpenGL transformation matrices to implement the spatial relationships in our scene graphs.

Each node in the scene graph represents a new coordinate system relative to the cen-

troid of the scene object at that node. Each scene graph has a scene root element, which represents the origin of the virtual world that is being described.

An example scene graph, taken from Lab5, is given below.

```
Scene origin
|
+-- [S(20,1,20) T(0,-1,-10)] Ground plane
|
+-- [S(20,1,10) Rx(90) T(0,4,-20)] Sky plane
|
+-- [T(4,7,-19)] Sun
|
+-- [Ry(35) S(2,2,2) T(-2.5,0,-10)] HouseBase
    |
    +-- [S(1,0.5,1) T(0,0.75,0)] Roof
```

This scene graph describes the world model in Lab5; this has a ground plane and a sky plane and the sun, hanging from the scene origin. A house, rotated, scaled and translated is then drawn, and the roof of the house is scaled and translated to sit on top of the house, and moves with the house too, since this is a child element of the HouseBase (note that all transformations that are applied to the parent will also be applied to the children). Each scene object is prefixed by the spatial relationship between the scene object and its parent in square brackets, which corresponds to an OpenGL matrix (empty square brackets denote the identity matrix). Each spatial relationship is written as a series of translations (indicated by a **T** plus the x, y, and z units), rotations (indicated by an **R** plus the axis and angle of rotation) and scalings (indicated by an **S** plus the x, y and z units).

Make sure you look at the scene graphs for each example you run; the animated person is particularly important to understand.

Getting Started

The code for the labs is available as a zipped workspace file that includes one project per example application. The zipped file containing all Java bits and pieces can be obtained from wiki. Note that instructions for how to use Eclipse with the code are on the wiki site too.

Lab 1: How to use OpenGL and Java; spatial awareness

In the labs, and in your coursework, you will use the LWJGL library, and extend an abstract base class we wrote to hide some of the implementation details and focus on what is more important.

➥ The general structure of a LWJGL based OpenGL program can be seen in the file `Lab1.java`. Open the `Lab1.java` in Eclipse and select the 'Lab1' project as the active project so that you can run the program.

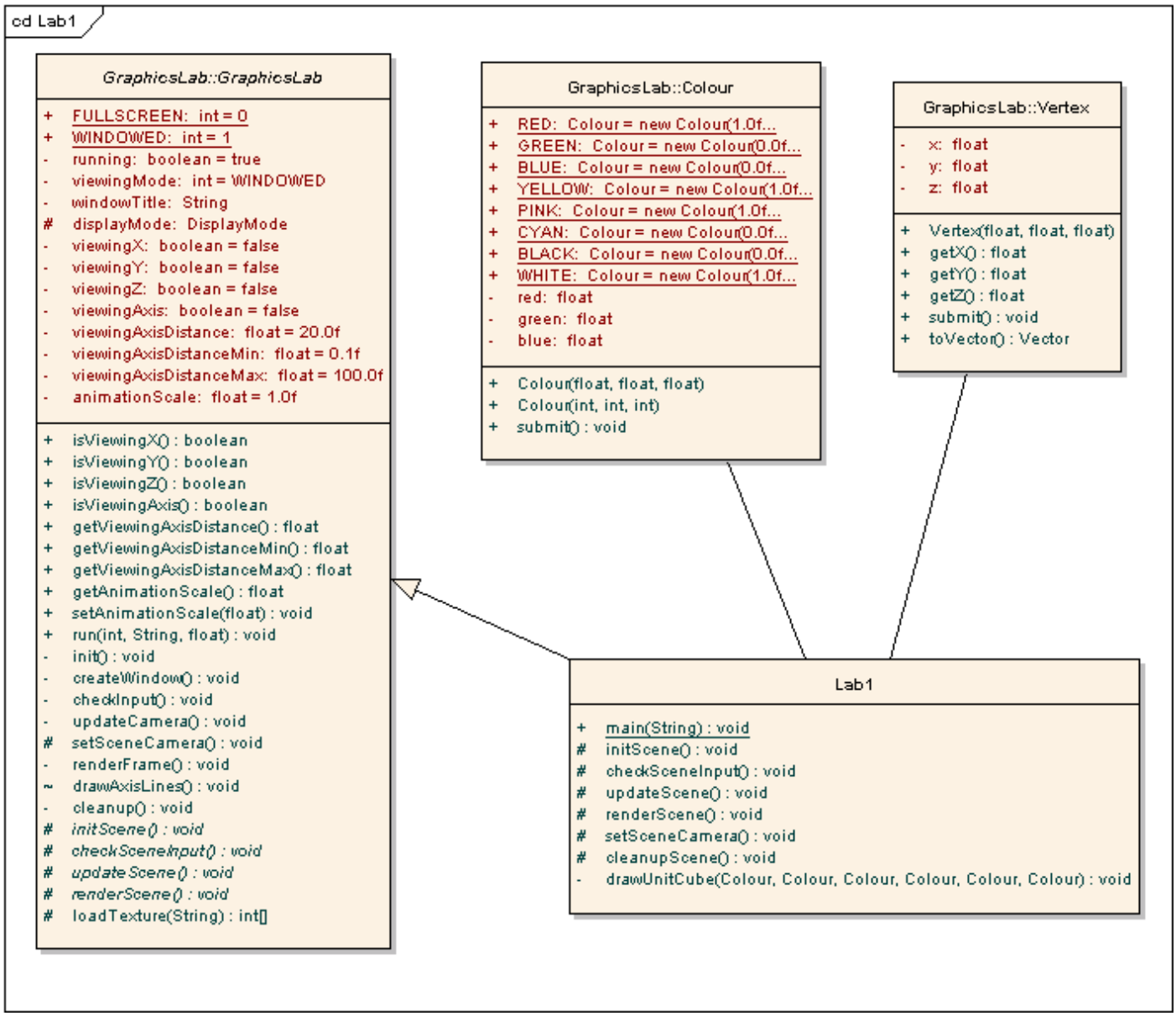


Figure 1: An overview of the Lab1 class and associated classes.

The first thing to do is to look at the code (resist the temptation to run the program). The header of the files will always look something like the below:

```
/* Lab1.java
 * A simple scene consisting of three boxes
 * Scene Graph:
 ...
 */
package Lab1;
import org.lwjgl.opengl.GL11;
import org.lwjgl.util.glu.GLU;
import GraphicsLab.*;

/**
 * Program comment in Javadoc style
 */
public class Lab1 extends GraphicsLab
{
  ...
}
```

All the labs are documented and use the Javadoc conventions; try to use this in your coursework submission too. Notice that the first thing you see in each example is the scene graph - this is very important (that's why it comes near the top) - it provides an abstraction of the contents of the scene in a very compact form. You must produce a scene graph for your coursework submission, so it is best to get familiar with them in the labs.

After the scene graph we tell the Java Virtual Machine to import the LWJGL library (at least the part of it relevant to OpenGL). This will appear in all the programs. Following this we see the main program comment. Finally we get to the class definition. In this case we define a single class (this will typically be the case) called `Lab1` which extends our abstract base utility class `GraphicsLab`, as shown in Figure 1. Before we go any further we'll quickly explain what is in the `GraphicsLab` class.

The `GraphicsLab` abstract class is designed to provide various utility functions and encapsulate behaviour that is needed across al the graphics labs. I do not suggest you undertake a deep analysis of it at this point, but once you are familiar with the graphics labs then you are welcome to take a look under the bonnet. For now I will describe the key methods in the class (and the associated helper classes that are also provided) shown in Figure 2.

The critical method to define in terms of producing computer graphics is the `renderScene` method. This is where you put the code that defines what should be drawn; if we look at the `Lab1.java` files we can see that the `protected abstract void renderScene();` from the `GraphicsLab` abstract class is overridden with:

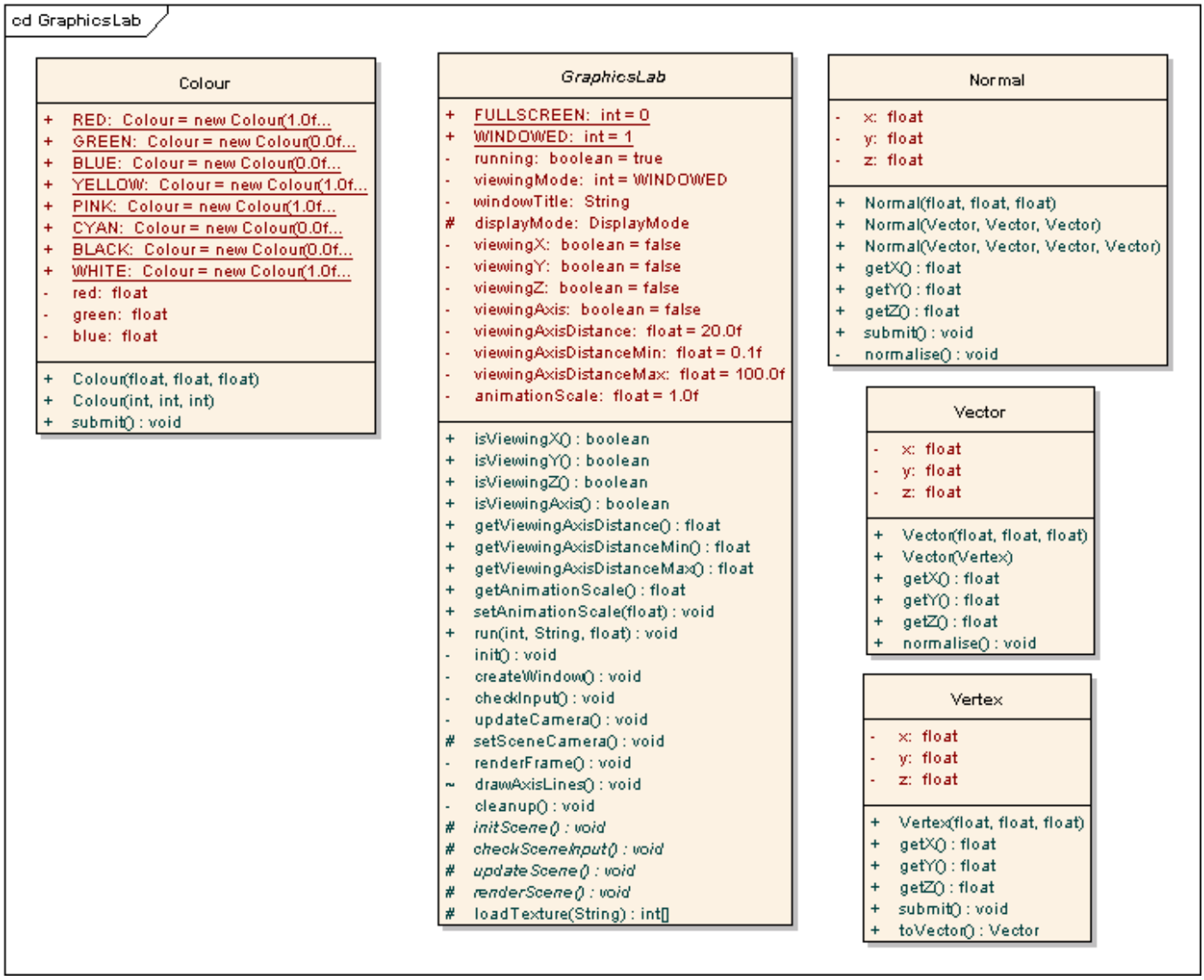


Figure 2: An overview of the GraphicsLab and associated classes.

```
protected void renderScene()
{
  // position and draw the first cube
  GL11.glPushMatrix();
  {
    GL11.glTranslatef(0.0f, -1.0f, -2.0f);
    drawUnitCube(Colour.BLUE,Colour.BLUE,Colour.RED,Colour.RED,
                 Colour.GREEN,Colour.GREEN);
  }
  GL11.glPopMatrix();
  ...
}
```

This code draws three cubes on the screen.

➥ Run the program, `Lab1` to see the effect (in the Package manager, right-click on `Lab1.java` and select “Run as” and “Java Application”), select. Can you understand

the code in the `renderScene` method?

The code is not too complex, but introduces a lot of features. We'll walk through the code first. The first thing you notice is the command `GL11.glPushMatrix()`. This command has a structure you'll see over and over again. It starts with the `GL11.` - this identifies the command as being related to the OpenGL v1.1 API. All OpenGL commands you will see and use start with `GL11.` and then are followed by the actual OpenGL function call, in this case to `glPushMatrix()` which takes no parameters. If you are ever unsure about what the different OpenGL functions are doing then you can check the online manual, which can be accessed from the CS2150 wiki page – but just look for the part following `GL11.` not the whole string.

`glPushMatrix()` is an OpenGL command that allows you to store the current value of the composite model-view matrix (typically) onto the *matrix stack*. This is a very important concept and allows you to isolate the effect of subsequent transformations by using `glPopMatrix()` to recover the value from the top of the stack. So in the example code above, the composite model-view matrix is stored on the stack, then a translation is applied using `GL11.glTranslatef(0.0f, -1.0f, -6.0f)`. This translation call modifies the composite model-view matrix, to reflect a translation of 0.0 in the x-direction, -1.0 in the y-direction and -6.0 in the z-direction. Remember x is across, y is up, and z is out of the screen in the world coordinate system, so this means that anything that is now draw is moved zero units across, one unit down, and 6 units away from the viewer.

The next command is one we have written for you: `drawUnitCube(...)` which draws a unit cube on the screen, with the parameters providing information on what colour each face is to be drawn in. We'll look at the code in a minute. For now just assume you know how the cube is drawn; remember that the translation that comes above this has modified the composite model-view matrix so this cube will be drawn zero units across, one unit down, and 6 units away from where it is originally defined. After the cube has been drawn the composite model-view matrix is then popped off the matrix stack using `GL11.glPopMatrix()`, so that the composite model-view matrix reverts back to what it was before the call `GL11.glPushMatrix()`. This means that the above translation only affects the single instance of the cube drawn in the above code fragment. Note that the { } brackets around the translation and draw method are not strictly needed, however their presence emphasises that the enclosed code is isolated (in terms of the effect on the composite model-view matrix).

Now we'll look at how the cube is drawn:

```
private void drawUnitCube(Colour near, ... )
{
  // the vertices for the cube (note that all sides have a
  // length of 1)
  Vertex v1 = new Vertex(-0.5f, -0.5f,  0.5f);
  Vertex v2 = new Vertex(-0.5f,  0.5f,  0.5f);
  ...
  // draw the near face:
  near.submit();
  GL11.glBegin(GL11.GL_POLYGON);
  {
    v3.submit();
```

```
      v2.submit();
      v1.submit();
      v4.submit();
    }
    GL11.glEnd();
    ...
}
```

In the above code you see several important things. Firstly we define the vertices (that is points that define the corners) of the cube we want to draw. This is centred about the origin (0,0,0). The vertices of the cube are thus the eight points that make up all the corners. When creating 3D shapes we need to be able to define the vertices.

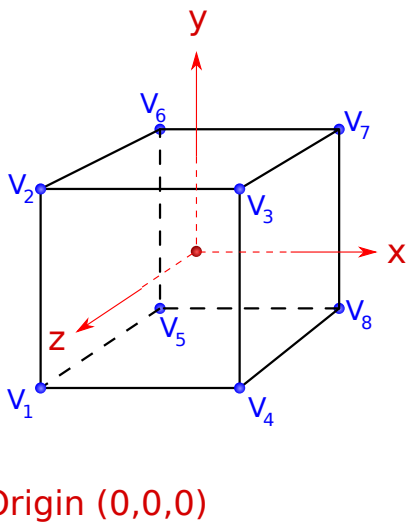


Figure 3: The cube, as defined in the code in Lab1.

Figure 3 shows the manner in which the cube is constructed. The vertices each have 3 coordinates, the x, y, and z coordinate respectively.

➥ On Figure 3 write down the coordinates for all the cube vertices – do they make sense?

The vertices are defined to be of type `Vertex` which is it's own class. The class enables us to store a 3 component vector, as shown below:

```
public class Vertex
{
  private float x; /** the x component of this vertex */
  private float y; /** the y component of this vertex */
  private float z; /** the z component of this vertex */

  /**
   * Constructs a Vertex object from its x, y and z components
   */
  public Vertex(float x, float y, float z)
  {
      this.x = x;
      this.y = y;
```

```
        this.z = z;
    }

    /**
     * Submits this Vertex to OpenGL using an immediate mode call
     */
    public void submit()
    {
        GL11.glVertex3f(x,y,z);
    }
    ...
}
```

Most of the code is quite clear; the class essentially stores the vertex coordinates, and has one significant method, called `submit`. This code passes the vertex to OpenGL using the command `glVertex3f(x,y,z)`, so that it can be 'drawn'. Looking back at the code to draw the square we see that that the vertices are submitted in the part of the code that draws the faces. OpenGL works by defining objects to be drawn between a `glBegin(.)` and `glEnd()` call. In the call to `glBegin(.)` we need to specify what is being drawn; here we draw a polygon by calling `GL11.glBegin(GL11.GL_POLYGON)`. There are a range of permissible drawing modes, as you'll see in Lab2.

You can't just pass anything in between the `glBegin(.)` and `glEnd()` calls. In general you should pass vertices (and normals) within these calls. Note the ordering of the vertices in the call *is* important; here we pass in v3, v2, v1, v4.

➥ Make sure you understand the order of the vertices for all faces – in what sense of rotation are the vertices given (clockwise or anti-clockwise)?

In OpenGL vertices must always be given is anti-clockwise order, as viewed from the front (as in the outside of the solid) of the face (so this is quite tricky for the rear face of the cube, since we need to put ourselves 'behind' the cube and imagine looking at this back face from the front!).

➥ In the Lab1 Java code change the order of the front face vertices: what happens?

The results will depend on what exactly you changed, but in general the object will not be drawn correctly. One of the most common problems in using OpenGL is that the vertices are not in the correct order, or they are not all on a plane. When OpenGL draws polygons or other shapes, all the vertices of these must lie in a 2 dimension plane (i.e. on a flat thing!). This is most easily achieved by designing your objects to be aligned with the major x, y, z axes. If this is the case, then all the values of one of the x, or y, or z coordinates must be the same for each face (but not the whole 3D object).

In the code to draw the squares you will also notice that there is a command `near.submit()` just before the call to `GL11.glBegin(GL11.GL_POLYGON)`. This sets the colour of the near face - in the call to `drawUnitCube(Colour near, ...)`, near is of type `Colour` which is a class in it's own right. Take a look at the Java for this class (it is in the GraphicsLabs directory). You can see this stores information on

the colour, which is again a 3 component vector of floats, this time the components being the amount of red, green and blue light the colour should contain. Again there is really only one method of significance in the `Colour` class; the `submit` method, which passes the colour to OpenGL using `GL11.glColor3f(red, green, blue)`. Note that OpenGL is a state machine; when you set the colour in one place in your code, it will stay set, and all objects drawn will use those colour properties until you change it. In the call `near.submit()` the colour defined for the near face (in this case blue, using a pre-set colour from the `Colour class`) is used.

The final part of the Lab1 code to look at is the method that defines the way we look at the scene, that is the method that sets the base projection and model-view matrices. The code is shown below:

```
protected void setSceneCamera()
{
  // call the default behaviour in GraphicsLab. Set the default
  // perspective projection and default camera settings ready
  // for some custom camera positioning below...
  super.setSceneCamera();

  // Set the viewpoint using gluLookAt. This specifies the
  // viewers (x,y,z) position, the point the viewer is looking
  // at (x,y,z) and the view-up direction (x,y,z), normally
  // left at (0,1,0) - i.e. the y-axis defines the up direction
  GLU.gluLookAt(0.0f, 0.0f, 10.0f,   // viewer location
                0.0f, 0.0f, 0.0f,    // view point loc.
                0.0f, 1.0f, 0.0f);   // view-up vector
}
```

Note that the `GraphicsLab` base class already has a complete definition of the projection and model-view matrix setup (which is called by `super.setSceneCamera()`). If you want to you can take a look at this in the `GraphicsLab` class. In general I would recommend you leave the projection matrix as it is; recall the projection matrix (`GL_PROJECTION`) is used to define the lens of the camera, i.e. it is used to zoom in and zoom out of the scene. The projection matrix is best changed using the `GLU.gluPerspective` command – to see how this works look it up using the OpenGL online manual.

The modelview matrix (`GL_MODELVIEW`) is used to position and point the camera, i.e. to say where it is in space, and where it is looking at (and also what direction is up). In general you will always be working with the model-view matrix. The `GLU.gluLookAt` command, from the OpenGL utilities library (GLU), is a very easy way to position and point the camera. You simple specify the 3D (x,y,z) coordinates of the viewer (where the camera is), the view point (where the camera is looking) and the view-up vector (what direction is up).

➥ Move the camera around the boxes to view them from the right (hint change only the viewer location), left, above and behind. How do the views differ from the plan views?

You should now understand the `renderScene` method in the Lab1 code. We now

need to see if you understand where these cubes that are drawn are in space. To help you keep track of where objects are in the code, we have written some helper functions.

➥ Run the Lab1 code, then press the 'x' key. This shows you a view of the scene, using an orthographic (i.e. like are used in plan drawings; not 3D, no perspective) view of the objects in the scene, as if you were looking directly down the x-axis (i.e. from the right hand side of your screen). The coloured lines show the positive y-axis (green) – that is up, and the positive z-axis (blue) – that is coming out of the screen.

➥ Now press the 'y' key; what do you see? This is like looking from directly above. The red line is the positive x-axis.

➥ Now press the 'z' key - this time the axes drawn are the x- and y-axes, and the view is from directly in front of the scene.

To help you orient yourself the tops and bottoms of the boxes are coloured green, the sides red and the front and back faces blue, and these colours are maintained on the plan views. These 3 plan views are very useful when trying to put the objects where you want them to go, and in checking they are where you think they are. The code to plot the plan views is hidden in the `GraphicsLab` base class – I advise you not to look at it yet; save it for when we look at viewing in 3D in the lectures.

➥ In the plan views you can also zoom in and out, pressing the arrow keys (up to zoom out, down to zoom in) at the same time as holding down the x, y or z keys.

➥ Using the plan views you have, the code and *a piece of paper* work out how to change the translations of the cubes so they can be stacked on top of each other.

It is important that you do this on paper; you should always draw what you want to do, then work it out on paper, before coding in graphics – you can't hack graphics stuff together easily!

➥ Now change the code – you'll need to modify the `GL11.glTranslatef(.)` commands to get the cubes to stack on top of one another.

➥ Once the cubes are stacked move the view again to check they really stack from all angles.

You might be wondering how such a small amount of code can produce the Lab1 results. The answer is that there is quite a lot of functionality in the `GraphicsLab` base class. For now I suggest you don't look at this, but if you are curious the code is all commented and most parts are quite obvious. As we develop more complex programs in later labs you will see how to extend other parts of `GraphicsLab`. To quit the Lab1 program, just press the Escape key, or kill the window.

Lab 2: Appearance in OpenGL

In Lab1 we looked at the basic operation of an OpenGL program, focussing on making sure you understood how the spatial location of objects was controlled and how viewing worked. We did not really talk about how to change the appearance of the objects.

➥ Run the Lab2 code; what do you see? Take a look at the code; what is different from Lab1?

Note that in Lab2 we do not set the camera matrices (the projection and modelview matrices) rather we rely on the default implementation in the `GraphicsLab` base class. This puts the camera at the origin, looking down the negative z-axis, with the y-axis being defined as up.

➥ Change the colours on the front faces of the cube to be the colours of a traffic light. You need to check the `Colour` class to see what colours are available.

Making the red and green lights is pretty easy, but what about orange? There are two ways you could do this; either by adding `ORANGE` as a hard coded colour to the `Colour` class, or by directly defining the colour yourself. The `Colour` class has two constructors; one that takes floats in the range zero to one, and one that takes integers in the range zero to 255. If you check on the wiki site you will see a link to a web page that defines the colours, in terms of their integer values in the range 0-255. Looking for orange, there are several options; I like R = 255, G = 127, B = 0. We can pass the new colour into the call to `drawUnitCube` by creating a new instance of the `Colour` class using `new Colour(255,127,0)`.

Alternatively if you wanted to use the orange colour a lot you could define an appropriate constant in the `Colour` class using `public static final Colour ORANGE = new Colour(255,127,0)` – this is actually already there, so you can just uncomment it.

➥ Also make sure you can also define your own colour in `Lab2.java` directly.

In OpenGL colours, and other properties can be set per vertex. The colours are then *interpolated* across the faces of the objects.

➥ Try modifying the draw cube method so that the first four colours passed in are applied one after another to the four vertices of the front face. What is the effect?

The code should look something like the below:

```
// draw the near face:
GL11.glBegin(GL11.GL_POLYGON);
{
  near.submit();
  v3.submit();
  left.submit();
  v2.submit();
```

```
    right.submit();
    v1.submit();
    top.submit();
    v4.submit();
  }
  GL11.glEnd();
```

Make sure you understand why you get the results you see. You might want to refer back to the list of vertices in Figure 3.

In addition to the appearance of the object in terms of its colour (which as we shall shortly see is better defined using materials in any case) we can also control the style in which the points, lines and polygons are drawn. Like the colour properties the style properties are also treated as a state machine, so set the once and they will apply to everything that is drawn afterwards until you reset them.

➥ The first decision we can make is whether to draw polygons as filled areas, or lines (wireframe). To make a wireframe view we simply need to specify that the polygons should be drawn as lines.

Change the `initScene` method in Lab2:

```
protected void initScene()
{
  GL11.glDisable(GL11.GL_CULL_FACE);
  GL11.glPolygonMode(GL11.GL_FRONT_AND_BACK, GL11.GL_LINE);
}
```

We need to switch off back-face culling so that both the front and the back faces are drawn, otherwise we will only see the front faces, and the wireframe view won't look right. In specifying the `glPolygonMode` we can refer to which face (the first argument) which can be `GL_FRONT`, `GL_BACK` or `GL_FRONT_AND_BACK`. We can also specify the style; commonly used options are `GL_FILL` (the default), `GL_LINE` or `GL_POINT`.

➥ Change the style so that the polygons are drawn as points.

Notice that the lines are thin and the points very small. These styles can also be changed using `glLineWidth( float )` and `glPointSize( float )`.

➥ Change the size of the points and the thickness of the lines and see the effect.

At this point you might be wondering where the blue front face has gone, but note the order in which the faces are drawn; disabling back-face culling means all faces are drawn; the top, bottom and side ones are drawn over the top of the front face. This illustrates the immediate mode nature of many OpenGL commands; the effects are drawn directly to the framebuffer (i.e. the pixels are set to the given colour once the `glEnd()` command is called), so anything that draws onto the the same pixel later in the frame just removes what was first there. Since we make all these changes in the initialisation, the effect will persist for all objects drawn. If we want to make changes

that apply only to one object we need to change the drawing state for one object and the reset it for the next one.

➥ Try to make the top cube be drawn in a wireframe manner, with all other cubes as solid objects. Hint: you will need to re-enable back-face culling and you need to modify the `renderScene` method.

Drawing your objects as wireframe, especially when you use different colours on different faces can be very useful when you are trying to understand why a solid object does not render correctly – a sort of visual debugging.

Lab 3: Construction in OpenGL

Building 3D models is an important part of graphics programming. Typically in a real-life situation you would use a range of tools to help you build your 3D models, using existing models (aka code re-use) and a carefully designed GUI. But here you will learn the hard way (I believe this is an important skill if you are to master the more advanced tools and understand their limitations!), using pencil and paper.

➥ Open Lab3, and run the program. What does it do? Look at the code - do you notice any differences from Labs 1 and 2?

The graphical changes are quite obvious; only a single cube is created and it is quite red, and a bit rotated.

But there is a more significant change to the code. We have started to use display lists. Display lists are a feature of OpenGL that allow objects to be pre-defined and pre-compiled. In many systems this means the graphical objects are then stored on the GPU memory rather than in system memory, and thus their use is very much faster since there is no interaction with the system bus when they are rendered. Display lists are also quite reminiscent of retained mode graphics packages, since they are kept in memory and drawn only on rendering. In general it is a good idea to use display lists; the main reason is speed. However they should only be used for rigid bodies - if there are transformations within a display list these will only be evaluated at compile time (which should happen in the `init` method), otherwise all benefits are lost, indeed there are significant disadvantages. So use display lists for the rigid parts of objects - they can still be used in a larger object that will be animated, but the display list can only contain rigid sub-elements. You will see more examples later in the labs.

➥ Look at the code that defines the display list, and make sure you can follow it.

Now we want to add a simple triangular roof to the cube, to make it into a house. We will proceed as follows:

1. We will draw the roof on paper, at the origin
2. We will work out the roof's vertices (relative to the origin)
3. We will then add the necessary code to draw the roof (at the origin)
4. Prior to drawing the roof (in OpenGL), we will move to the top of the cube, so that it is drawn above, not inside the cube.

Guidelines for drawing objects

We could draw the house as a whole, by adding new vertices to the cube. However, this is not recommended in practice. The reason for this is that you might want to draw several houses, with different sized roofs. Having 2 objects, one for the base of the house (a cube) and one for the roof (a triangular prism, see image below), makes it a lot easier to create different looking houses. In general, when you want to draw a complex object, you should try and split it into simple, reusable 3D objects and

design each of these independently. You can then put the simple objects together using transformations.

➥ Take a pen and a piece of paper. We will make the roof a triangular prism rather than a pyramid (left shape below).

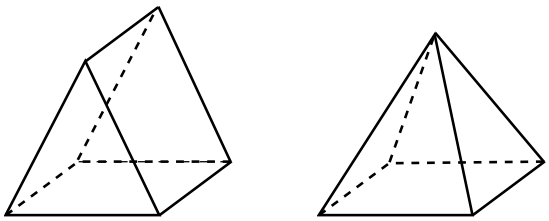


Figure 4: A triangular prism (left) and a pyramid (right)

➥ Draw the roof. Make it big enough, so you can add names for the vertices later.

Choosing the origin

This is a very important step, as all transformations you apply to an object are relative to the origin. When choosing the origin of an object, ask yourself whether the object presents any symmetry. If so, you probably want to make the origin coincide with the centers (or axes) of symmetry. Does our roof present any symmetries? To find symmetries, ask yourself the question: can the object be split in 2 so that both parts are the mirror of each other? Or is there an axis around which the object can be rotated by and still look exactly the same?

The roof presents 2 reflection (i.e. mirror) symmetries, shown below (figures a and b). It makes sense to put the origin on the line where the 2 symmetry planes (in gray) meet (figure c). Where on that line does not really matter. However, since we will need to translate the roof so that its *bottom* face lies on the top face of the cube, setting the origin on the *bottom* face will make it easier to work out the required translation later.

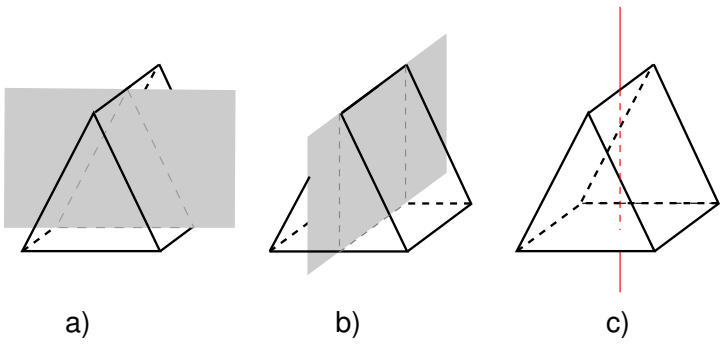
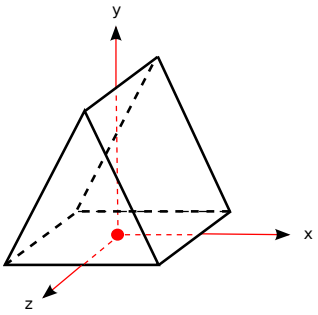


Figure 5: Reflection symmetries and their intersecting line

➥ Set the origin of the roof so that it lies at the intersection of the vertical symmetry line and the bottom face.

The origin is the point of coordinate (0,0,0). Note that when designing new objects, we imagine that we are working in an empty world. So the origin of the new object is not related to the origin of the previous scene (the scene with the cube).

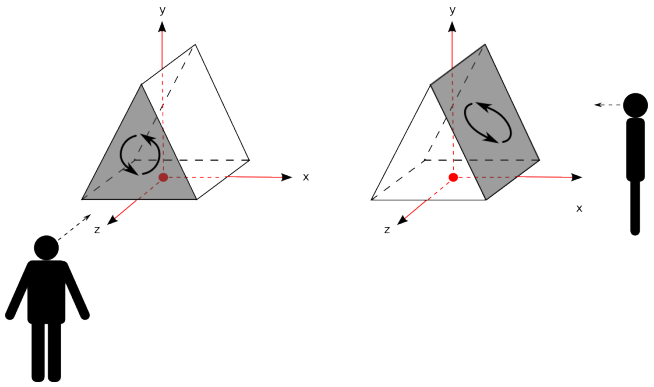
➥ Add the x, y and z axes to your drawing. These are represented as arrows starting from the origin and following the right hand coordinate system (x to the right, y to the top, and z towards us).



➥ Give each of the 6 vertices a name, e.g. V_1, ..., V_6.

➥ Given that the origin is the point (0,0,0) and that the roof fits within a unit cube (i.e. width = height = depth = 1), work out the coordinates of each vertex.

➥ Work out the order in which to pass the vertices for each face of the roof. Remember that you must pass the vertices in counter-clockwise order, *from the point of view of someone looking at the face from the outside of the object*.



Now that you have computed the vertices and the faces, you will want to add a method `drawHouseRoof` to the code, which declares the vertices and draws the faces of the roof.

Using the ShapeDesigner

Because implementing new objects can be sometimes a bit tricky, I have made a little "utility" to help you in the process: the ShapeDesigner. The ShapeDesigner allows you to work with a single shape (i.e. 3D object) and visualise it easily. With the ShapeDesigner, you can rotate or move the camera around and change the way things are drawn, by using simple keyboard shortcuts.

In the Package Explorer, open the `Designer` package. This package contains 2 classes: `AbstractDesigner` and `ShapeDesigner`. The `AbstractDesigner` class you need not worry about – it is an abstract class (you had probably worked that one out) which provides most of the functionality for the ShapeDesigner. Open the `ShapeDesigner` class. It is a very basic class, with only 2 methods:

- the `main(...)` method, which just runs the ShapeDesigner,
- the `drawUnitShape()`, which draws a single shape (the one we are designing).

For now, I've simply copied over the code from the `drawUnitCube` method so something gets drawn when you run the ShapeDesigner for the first time.

➥ Run the `ShapeDesigner`. You should see our good old unit cube in white wireframe. You can change the way the object is viewed using the following controls:

Key	Action
Tab	Switch between point / wireframe / polygon drawing modes
A	Toggle colours*
Up	Move the camera away from the shape
Down	Move the camera closer to the shape
X	Rotate about the world's X axis
Y or C	Rotate about the rotated Y axis (explication below)
Z	Rotate about the object's Z axis
Shift	Invert the direction of rotation (hold key while rotating)
Space	Reset the view to its initial state

** The colour toggle only works if you submit colours for the faces using the `submitNextColour()` method. See below.*

➥ Experiment with the controls. Make sure you can rotate the object, change its appearance (drawing mode and colour) and reset the view. The rotation system is a bit complicated. It essentially follows a gimbal (see http://en.wikipedia.org/wiki/Gimbal). The X rotation is about the world's X-axis (which doesn't move) and the Z rotation is about the object's Z axis (blue line). However, the Y rotation is about an axis which depends on both the others and is hard to visualise. You might find it easier to not use the Y rotation and stick to the other two, which are more intuitive. These two rotations are sufficient anyway to show the object from any angle.

To create a new shape in the ShapeDesigner, all you need to do is delete the body of the `drawUnitShape()` method (leave its signature) and start coding your own vertices and shapes. When declaring the faces, you can submit your own colours as usual, e.g. `Colour.RED.submit()`, or you can use `submitNextColour()` instead, which cycles through a list of predefined colours and also allows you to toggle between white and colour modes.

➥ In the ShapeDesigner, delete the body of the `drawUnitShape()` method and add the declarations for the roof's vertices, as done in `drawUnitCube()`, but using the coordinates you have worked out. You can test the vertices as you go by drawing them with the following statement, e.g. for vertex v_1:

```
GL11.glBegin(GL11.GL_POINTS);
v1.submit();
GL11.glEnd();
```

You can add the other vertices between the `glBegin(...)` and `glEnd()` statements.

➥ Once you are happy with your vertices, remove the code above so they don't get

drawn anymore and add the statements to draw the 5 faces. To take advantage of the automatic colouring, remember to use `submitNextColour()` before declaring the face. When designing the faces of an object, it is usually recommended to work on one face at a time. So once you are happy with a given face, you can comment out its code and move on to the next face. When all faces look ok, you can uncomment them all and draw the full object.

Common implementation problems are to do with getting the order of the points about the face correct (anti-clockwise) and the points not all lying in a plane (a polygon is flat, so all its vertices must lie on a same plane!).

➥ Once you have the code to draw the full roof, copy the `drawUnitShape()` method over to Lab 3, rename it to something more meaningful like `drawUnitRoof()` and remove any call to `submitNextColor()` (this only works in the ShapeDesigner). Create a display list for your roof, which will call `drawUnitRoof()`.

Depending on how you chose the origin for the roof object, you might well need to call a translation after you draw the house base (the cube) to put the roof on top of the house, but remember the other transformations that are applied before the base is drawn will also be applied to the roof – which is what you want! This means if you move the base of the house (e.g. rotate it, as is done here) then the same thing will be applied to the roof, so the house won't fly apart.

➥ Maybe you didn't sort the colour of the roof; make it something appropriate.

You might feel your house is too big or too small. This is very easy to change in OpenGL.

➥ The most simple way to make it smaller here is to move it further away from us: change the z-value of the translation in the `renderScene` method to a larger negative value, say -50. What does this do?

Of course this only makes the house look smaller, it is clearly really the same size! One important point to note here is that the order of the transformations in the `renderScene` method is critical.

➥ Change the order of the rotation and translation in `renderScene`. Can you explain what happens - remember that pressing the x,y and z keys gives the plan views in all labs.

OpenGL applies the transformations to the composite transformation matrix, by multiplication on the right (that is the the composite transformation matrix C gets updated by the current transformation matrix, T, in the following manner: $C \longleftarrow T \times C$) – this means the transformations are applied in the opposite order to which they appear in the code. In the original code we have:

```
protected void renderScene()
{
  // position the house
  GL11.glTranslatef(0.0f, -0.5f, -5.0f);
```

```
    // rotate the house a little so that we can see more of it
    GL11.glRotatef(35.0f, 0.0f, 1.0f, 0.0f);

    // draw the base of the house by calling the appropriate
    // display list
    GL11.glCallList(cubeList);
}
```

In the above code it is the rotation that is applied to the house (cube) vertices first, and then the translation to move the house away and down a little. Changing the order means the house is translated first, then rotated, which is probably not what we wanted to do! In general, we want to rotate objects about their centres (not always of course) and in most of the code we will show you, the objects are defined so that their centres are at the origin, thus rotation (which is always about the origin) should be applied before translations in general. Clearly if you want to rotate an object about another point, then you need to translate the object first. We will come back to this when we look at animation.

➥ The other way to make the house smaller is to scale it. There is a command `glScalef(sx,sy,sz)` which allow you to scale an object by a factor sx, sy, sz about the x, y and z-axes respectively. Apply this to house.

➥ What effect does the positioning of the scale command in the `renderScene` method have? Can you explain this?

The most important part of this lab is that you have understood how to define a new object in OpenGL.

Lab 4: Lighting and materials in OpenGL

To make 3D graphics look realistic you need to be able to define lighting. OpenGL has support for a range of lighting models. The models we discuss in the lectures; ambient (coming from all around equally), diffuse (from a direction but scattered in all directions) and specular (shining off an object, like reflection) are all quite easy to implement. When you set up the lighting properties, using up to 8 separate lights, you also need to set the material properties of the objects in the scene, to determine how they respond to light.

➥ Open up `Lab4.java` - this now has a house, but it looks all wrong if you run the code; note we have added the roof for you, so you can check if your roof is correct by comparing it with the roof in this code.

Notice there are some new elements to the `initScene` method (for now ignore the commented part):

```
protected void initScene()
{
  // global ambient light level
  float globalAmbient[]   = {0.2f,  0.2f,  0.2f, 1.0f};
  // set the global ambient lighting
  GL11.glLightModel(GL11.GL_LIGHT_MODEL_AMBIENT,
                    FloatBuffer.wrap(globalAmbient));
  ...
  // enable lighting calculations
  GL11.glEnable(GL11.GL_LIGHTING);
  // ensure that all normals are automatically re-normalised
  // after transformations
  GL11.glEnable(GL11.GL_NORMALIZE);
  ...
}
```

We typically set the lighting in the initialisation (lighting is also part of the OpenGL state machine, so once set it remains that way until you alter it). Of course we can also include lighting within our rendering methods (normally we just switch them on and off here in fact); this is only sensible when we want to change lighting during the course of the execution of the program (for example we might want to switch off a light if a key is pressed). In general you should try to avoid unnecessary changes of OpenGL state; so if you want to use the same lighting model all the time, as here, make sure this is set up in the initialisation so it is not reset each time the render process is called.

In the above code two things are done. Firstly an ambient lighting model is defined globally (i.e. this ambient model applies everywhere). In general specific lights in OpenGL have a position, so their ambient lighting value does not always make sense, although they will add some illumination to the global ambient light field. The ambient lighting is set quite low, since values for lights should be in the range of 0.0 – 1.0; this is done so the house object is visible, but so that later the ambient light won't dominate. Note the rather unpleasant necessity of having the use the `FloatBuffer.wrap`

method to pass the arrays to OpenGL; this is just something you'll have to remember to do for all lighting related commands.

The other OpenGL commands in the above code, enable the lighting model - this can be switched on and off at will, but I suggest that in general you leave it switched on. The automatic re-normalisation (to unit length) of surface normals is also switched on; again I suggest you always enable this.

There have also been changes in the `renderScene` method:

```
...
// how shiny are the front faces of the house
// (specular exponent between 0 and 128)
float houseFrontShininess  = 2.0f; // Not at all shiny!
// specular reflection of the front faces of the house
float houseFrontSpecular[] = {0.1f, 0.0f, 0.0f, 1.0f};
// diffuse reflection of the front faces of the house
float houseFrontDiffuse[]  = {0.6f, 0.2f, 0.2f, 1.0f};

// set the material properties for the house using OpenGL
GL11.glMaterialf(GL11.GL_FRONT, GL11.GL_SHININESS,
                 houseFrontShininess);
GL11.glMaterial(GL11.GL_FRONT, GL11.GL_SPECULAR,
                FloatBuffer.wrap(houseFrontSpecular));
GL11.glMaterial(GL11.GL_FRONT, GL11.GL_DIFFUSE,
                FloatBuffer.wrap(houseFrontDiffuse));
GL11.glMaterial(GL11.GL_FRONT, GL11.GL_AMBIENT,
                FloatBuffer.wrap(houseFrontDiffuse));
...
```

There is quite a lot happening in the above code; in particular the material properties for the front (that means outward facing) faces of the house cube are being set. Normally 3 main properties must be set for the materials; the ambient reflection coefficients, the diffuse reflection coefficients and the specular reflection coefficients and exponent. In general the ambient and diffuse reflection coefficients, set for red, green and blue light, should be the same (as they are in the above code) - using different values is not physically realistic. Remember that all reflection coefficients must also be in the range 0.0 – 1.0. This also applies to specular reflection, but here we also need to set the specular exponent, which determines how shiny the object looks – the higher the value the more shiny it will appear (the maximum value is 128). You might notice a fourth value is also used; this is the so called alpha value and provides a method for creating transparency; we will not cover this so always leave the final value at 1.0.

Every face that is to be rendered must be assigned a material value, but again materials are part of the OpenGL state machine; once set the material properties will be applied to all objects rendered until they are changed. This is why a new set of material properties are defined before the roof is drawn.

➥ Run the code and look at the results. It does not look very interesting or 3D – any idea why?

If you just get a black screen look closely - raise the ambient lighting to have values of 0.5 for each component and see what happens; then return these to their original values of 0.2. So far we have only set the ambient light, and this light has no directional preference, so it just looks like we have used colours again; to get the 3D effect we need to add some directional light. That is your next job.

➥ Uncomment the code in the initialisation that sets up the first light. The code is show below. Run the code and notice the effect. Can you explain what you see?

The code we have uncommented looks like:

```
// the first light for the scene is white...
float diffuse0[]  = { 0.6f,  0.6f, 0.6f, 1.0f};
// ...with a dim ambient contribution...
float ambient0[]  = { 0.1f,  0.1f, 0.1f, 1.0f};
// ...and is positioned above and behind the viewpoint
float position0[] = { 0.0f, 10.0f, 5.0f, 1.0f};

// supply OpenGL with the properties for the first light
GL11.glLight(GL11.GL_LIGHT0, GL11.GL_AMBIENT,
             FloatBuffer.wrap(ambient0));
GL11.glLight(GL11.GL_LIGHT0, GL11.GL_DIFFUSE,
             FloatBuffer.wrap(diffuse0));
GL11.glLight(GL11.GL_LIGHT0, GL11.GL_SPECULAR,
             FloatBuffer.wrap(diffuse0));
GL11.glLight(GL11.GL_LIGHT0, GL11.GL_POSITION,
             FloatBuffer.wrap(position0));
// enable the first light
GL11.glEnable(GL11.GL_LIGHT0);
```

This code sets the properties of the lights that are applied to the scene. Here we set a low level of ambient light from `GL_LIGHT0` - note we can have up to seven other lights in the scene as well. We also set the diffuse and specular components of the light; physically these ought to be the same, as they are set in this example, but sometimes they can be set differently to achieve strange looking effects. I do not recommend this. We also have to position the lights in the scene; this is quite important since it will determine where the illumination falls, although by default distance is ignored in OpenGL and only the vector direction is important. Finally we need to enable the light, so it is used. Individual lights can be set up in the initialisation and enabled and disabled within the program, to simulate switching lights on and off for example.

➥ Press (and hold down) the r key; this rotates the house and allows you to see the lighting effect much more clearly.

The rotation is the first example of animation; the code to perform the animation is quite simple. We add a field called `houseRotationAngle` to the `Lab4` class; this field is used in the rotation transformation that is applied to the house prior to drawing:

```
GL11.glRotatef(houseRotationAngle, 0.0f, 1.0f, 0.0f);
```

The code that modifies the rotation angle is in the `checkSceneInput` method:

```
protected void checkSceneInput()
{
  // Rotate if the r key is pressed
  if(Keyboard.isKeyDown(Keyboard.KEY_R))
  {
    houseRotationAngle += 1.0f * getAnimationScale();

    // Wrap the angle back around into 0-360 degrees
    if (houseRotationAngle > 360.0f) {
      houseRotationAngle = 0.0f;
    }
  }
}
```

If the animation is too slow this can be modified by setting the animation scaling factor in the `main` method:

```
public static void main(String args[]) {
 new Lab4().run(WINDOWED,"Lab 4 - Lighting",0.01f);
}
```

where the value, `0.01f`, is the value of the `animationScale` field in the `GraphicsLab` base class. If the rotation is too slow on your computer make this number bigger; if it is too fast make it smaller. Try to remember to use this in your coursework, since then we have a simple method to speed up or slow down your animation so it runs at a sensible speed on the computers we mark your work on! More on animation later.

➥ Experiment with moving the light about - what happens if you put the light on the other side of shape (i.e. along the negative z axis)?

➥ Change the material properties of the house to simulate the walls being white-washed (hint: you should only change the material properties).

➥ Change the lighting so that it simulates a sodium (orange) street light at night; do not change the material properties.

➥ Change the lighting and materials so that the house seems to be by the sea (i.e. bright light) and is painted yellow.

➥ If you have time, try to add a door to the house (this will simply be a rectangle with different material properties; choose any colour you like. Remember to start with pencil and paper.

Lab 5: Quadrics and textures in OpenGL

In this lesson you will learn how to use textures (bitmaps that are applied to objects to enhance detail without adding greatly to the complexity of the geometry) and quadrics. Quadrics are widely used in OpenGL to define objects that are curved (in reality of course they are simply made up of many small polygons), so they are great for spheres, cylinders, cones and disks.

➥ Open the Lab5 project and run it. What do you notice?

We'll start with the most dramatic change; the use of textures. In practice we have hidden much of the complexity of using and applying textures in the `GraphicsLab` base class – you can take a look, but I don't advise you to change things here! The code to load the textures from file is in the `initScene` method:

```
// load the textures
groundTextures = loadTexture("Lab5/textures/grass.bmp");
```

This loads the texture from a bitmap called grass.bmp in the Lab5/textures directory.

➥ Take a look at the raw texture – i.e. open the bitmap in the default viewer (e.g. paint). What do you notice?

In general I suggest you stick to using bitmap formats for your textures and keeping them square with a dimension of $2^n \times 2^n$, e.g. 128 by 128 or 512 by 512. Other sizes *should* work with OpenGL but you cannot guarantee this for all hardware. The texture then needs to be applied to the polygon in the `renderScene` method:

```
// draw the ground plane
GL11.glPushMatrix();
{
  // disable lighting calculations so that they don't affect
  // the appearance of the texture
  GL11.glPushAttrib(GL11.GL_LIGHTING_BIT);
  GL11.glDisable(GL11.GL_LIGHTING);
  // change the geometry colour to white so that the texture
  // is bright and details can be seen clearly
  Colour.WHITE.submit();
  // enable texturing and bind an appropriate texture
  GL11.glEnable(GL11.GL_TEXTURE_2D);
  GL11.glBindTexture(GL11.GL_TEXTURE_2D,groundTextures.getTexture

  // position, scale and draw the ground plane using its display
  GL11.glTranslatef(0.0f,-1.0f,-10.0f);
  GL11.glScaled(25.0f, 1.0f, 20.0f);
  GL11.glCallList(planeList);

  // disable textures and reset any local lighting changes
  GL11.glDisable(GL11.GL_TEXTURE_2D);
  GL11.glPopAttrib();
}
```

```
GL11.glPopMatrix();
```

This code might seem quite complex at first glance; there is certainly a lot going on here. First and push-pop pair isolate any transformations applied to the ground plane.

Then we see a new use of push and pop, this time on the attribute stack. Using `GL11.glPushAttrib(GL11.GL_LIGHTING_BIT)` we are able to tell OpenGL to store (push) the current lighting settings on to the attribute stack. This enables us to save the settings, and recover them later. We then turn the lighting off, using `glDisable()` and set the background colour to white (so the texture is drawn cleanly). Only then do we apply the texture, first enabling the use of 2D textures and then finally binding the texture using `GL11.glBindTexture(GL11.GL_TEXTURE_2D, groundTextures.getTextureID())` – this is then applied to all polygons that follow and have defined texture coordinates, so after drawing the suitably transformed ground plane we have to remember to disable the texture and retrieve the old lighting settings from the attribute stack. In this example we have not used lighting and textures together, but we will show you how to do this later.

The only thing that remains is to link the textures to the geometry in the scene:

```
private void drawUnitPlane()
{
  Vertex v1 = new Vertex(-0.5f, 0.0f,-0.5f); // left,  back
  Vertex v2 = new Vertex( 0.5f, 0.0f,-0.5f); // right, back
  Vertex v3 = new Vertex( 0.5f, 0.0f, 0.5f); // right, front
  Vertex v4 = new Vertex(-0.5f, 0.0f, 0.5f); // left,  front

  // draw the plane geometry. order the vertices so that the plar
  // faces up
  GL11.glBegin(GL11.GL_POLYGON);
  {
    new Normal(v4.toVector(), v3.toVector(), v2.toVector(),
               v1.toVector()).submit();

    GL11.glTexCoord2f(0.0f,0.0f);
    v4.submit();

    GL11.glTexCoord2f(1.0f,0.0f);
    v3.submit();

    GL11.glTexCoord2f(1.0f,1.0f);
    v2.submit();

    GL11.glTexCoord2f(0.0f,1.0f);
    v1.submit();
  }
  GL11.glEnd();
  ...
```

Here we link the texture coordinates to the vertices of the ground plane object. We need to remember to do this for all objects that we want to draw with textures attached. Again you will need to get paper and pen out to check that the attachment is correct.

In general square textures are most easily applied to rectangular objects; we will not treat more complex cases here, although there are plenty of examples on the internet.

Using textures is really quite simple. Notice at the moment that the sky is rather bland (indeed it picks up the last material properties defined; in this case the materials applied to the roof. We have supplied a file in the Lab5/textures directory called daySky.bmp.

➥ Modify the code to load this and apply it to the sky plane.

➥ We have also supplied a file called nightSky.bmp - apply this to the sky plane instead and adjust the lighting to make the appearance more realistic (make sure you turn the sun into the moon - make it white, not yellow).

➥ To make the moon appear to glow you need to set the `GL_EMISSION` material property – this causes the object to appear to emit light, useful for objects you want to glow. Hint: don't forget that OpenGL is a state machine; once you set the `GL_EMISSION` property it will apply to all objects unless you turn it off.

If you are not sure how to use emission then you an check the manual, but it follows the pattern of other material properties.

The other new feature is the use of quadrics; there are several we can use, and LWJGL wraps these nicely. The sun in the scene which looks like a sphere is drawn using a quadric:

```
new Sphere().draw(0.5f,10,10);
```

The call to the draw method of the Sphere class takes three parameters; these are the radius of the sphere in world coordinate units (in this case 0.5 units), and then the number of slices (around the equator) and number of stacks (pole to pole) used to make up the sphere. LWJGL has reasonable documentation, so you can always explore the methods by accessing the Javadoc at `http://www.lwjgl.org/javadoc/`.

➥ Change the size, and the number of slices and stacks to see their effect. In general it is not a good idea to have too many stacks and slices since this produces very complex geometry and will slow down the application.

We'll explore other quadrics later, but for now this illustrates how easy they are to use.

➥ How could you change the sun to be rugby ball shaped? Hint: no need to change the quadric; use transformations.

One important point to note is that if you want to use quadrics they need to be imported at the top of the class file:

```
import org.lwjgl.util.glu.Sphere;
```

➥ Just to check you are following the placement of objects in the scene use the x,y and z keys to view the plan views of the scene. Do you understand these?

Lab 6: Animation in OpenGL

In this lab we will show you how to construct animations, using all the features we have seen thus far.

➥ Open and run the Lab6 project. There is quite a lot going on in the scene, indeed everything we have covered thus far is applied.

➥ Looks at the world model using the plan views.

The model now has several additional features; a tree has been created using quadrics:

➥ Look at the code – in particular look at the parts used to create the tree – make sure you understand this.

The code is much more complicated (at least in terms of the number of lines), but does not have anything substantially new. Note that we now have more fields; these are being used to store ID's for display lists, and variables used to control the animation.

The animation is quite simple; the moon can set and rise.

➥ Press the l key to lower the moon – can you see in the code how this happens?

In this example the animation is driven by user input: when the user presses the l key the boolean `risingSunMoon` field is set to false (it starts being set to true, i.e. at the start of the program the moon has aleady risen). The `checkSceneInput()` method is used to monitor the user interaction with the code:

```
protected void checkSceneInput()
{
  if(Keyboard.isKeyDown(Keyboard.KEY_R)) {
    risingSunMoon = true;
  }
  else if(Keyboard.isKeyDown(Keyboard.KEY_L)) {
    risingSunMoon = false;
  }
  else if(Keyboard.isKeyDown(Keyboard.KEY_SPACE)) {
     resetAnimations();
  }
}
```

The other options are to press the r key to make the moon rise, and the space bar to reset the animation. If the l key is pressed then in the `updateScene` method we check to see whether the moon should be rising or setting, and the setting (falling) option will be chosen:

```
protected void updateScene()
{
```

```
    // if the sun/moon is rising, and it isn't at its highest,
    // then increment the sun/moon's Y offset
    if(risingSunMoon && currentSunMoonY < highestSunMoonY) {
      currentSunMoonY += 1.0f * getAnimationScale();
    }
    // else if the sun/moon is falling, and it isn't at its lowest,
    // then decrement the sun/moon's Y offset
    else if(!risingSunMoon && currentSunMoonY > lowestSunMoonY) {
      currentSunMoonY -= 1.0f * getAnimationScale();
    }
  }
```

In the above code there are two possibilities; the moon should be rising, or falling. These are determined from the boolean value and whether the moon has yet reached its maximum or minimum elevation. If this is not the case the `currentSunMoonY` is incremented / decremented by an amount scaled by the `animationScale` to control the overall speed of the animation, as described in Lab4.

➥ Change the speed of the animation so it runs fast (e.g. change the scale in the `main` method `run` call to 1.0. Make it too slow. If you scale all the animations you script with a single scale then you can easily adjust the overall speed.

➥ Press the space bar to reset the animation.

It is very useful to be able to reset the animations, so you should try and include this in your coursework. Here it is quite simple and achieved in the call to the `resetAnimations` method:

```
private void resetAnimations()
{
  // reset all attributes that are modified by user controls
  // or animations
  currentSunMoonY = highestSunMoonY;
  risingSunMoon = true;
}
```

which simply resets the animation variables to their initial values.

Most animations are achieved by changing transformations applied to objects. In this simple example the animation is directly controlled by the user. In general this method of scripting animations is good in terms of allowing the user to have control over what happens, and when. However,it is often better to have a timer running and script animations as a function of the time (this is show in the extension labs). It is important to realise that animation can also involve lighting, materials and other properties. Combining all these allows some sophisticated effects to be achieved, and this is the aim of the coursework.

➥ Modify the animation so that when the moon sets the lighting gets even darker. Hint: you'll need to change the lighting properties; there are several ways to do this – I'd probably create a `turnOffMoonLight` method that changes `GL_LIGHT0` to have

simply a very low ambient level.

➥ Now allow the user to get the sun to rise by pressing the s key (this should only work if the moon has set). Hint: again you'll need to change the lighting, the materials of the sun / moon, and ideally also the sky texture. This is quite complex and should take some planning and time.

➥ If you want a real challenge try to create a complete story by using a time variable to script the rise and fall of the moon and sun. To make it really realistic you should ideally use sine and cosine functions for the gradual changes in lighting levels as the various celestial bodies rise and fall. Only attempt this if you have time!

It is a significant challenge; come back here if you have looked at the extension labs and understood what is there first.

Extra labs: useful examples in OpenGL

These extension labs are designed to show you a variety of different aspects of using OpenGL; they should be very helpful in the coursework. There is less to do here, but you must spend some time looking at them (the comments should be helpful!), and it is worth attempting some of the exercises.

Solar System

➥ Select and run the **SolarSystem lab**.

This example shows the application of simple time scripted animation of the Earth and Mars (with their moons). Note the x,y, and z keys still work if you want to see what is going on.

Key features of this example are the use of `glPushMatrix` / `glPopMatrix` to construct a hierarchical scene graph (or equivalently isolate the transformations appropriately). For example we want the Earth's moon to orbit around it (thus this is a child element in the scene graph and the Earth's rotation is also applied to the Moon)

➥ Look at the scene graph for the code and the `renderScene` method - it is well commented; make sure you understand why the animation works as it does and why the `glPushMatrix` / `glPopMatrix` are sited as they are.

The other important aspect is the use of single time variable, `timeday`, to construct the animation. Note this variable is used in all the transformations in the `renderScene` method.

➥ Make sure you understand the various rotations in the `renderScene` method; can you explain the units of time?

➥ Add Mercury and Venus; I have no idea how fast these orbit (or whether they have moons) but a quick internet search should reveal the answers!

Borg Cube

➥ Select and run the **BorgCube lab**. Anthony is very proud of this so make sure you go WOW![1]

This example shows the use of lighting and textures and animation. The actual lighting that is changed in the animation is the emission property that makes the cube appear to glow with different intensities as it rotates.

➥ Take a look at the code and make sure you understand it; again the comments are pretty self explanatory.

[1] Anthony isn't teaching the labs anymore, but we'll pass the word on to him!

➥ Modify the code so that the Borg cube can spin off into the distance (it will disappear behind the sky plane quite fast but you can fix this). This is quite a simple exercise.

➥ If you are feeling adventurous try to make the cube accelerate away. Note that if an object has constant acceleration a, then the velocity at a time zero will be $v = 0$ and for times in the future $v = a * t$, so the distance travelled, d, (i.e. the offset) will be $d = a * t^2$.

Animated Person

➥ Select and run the **AnimatedPerson lab**.

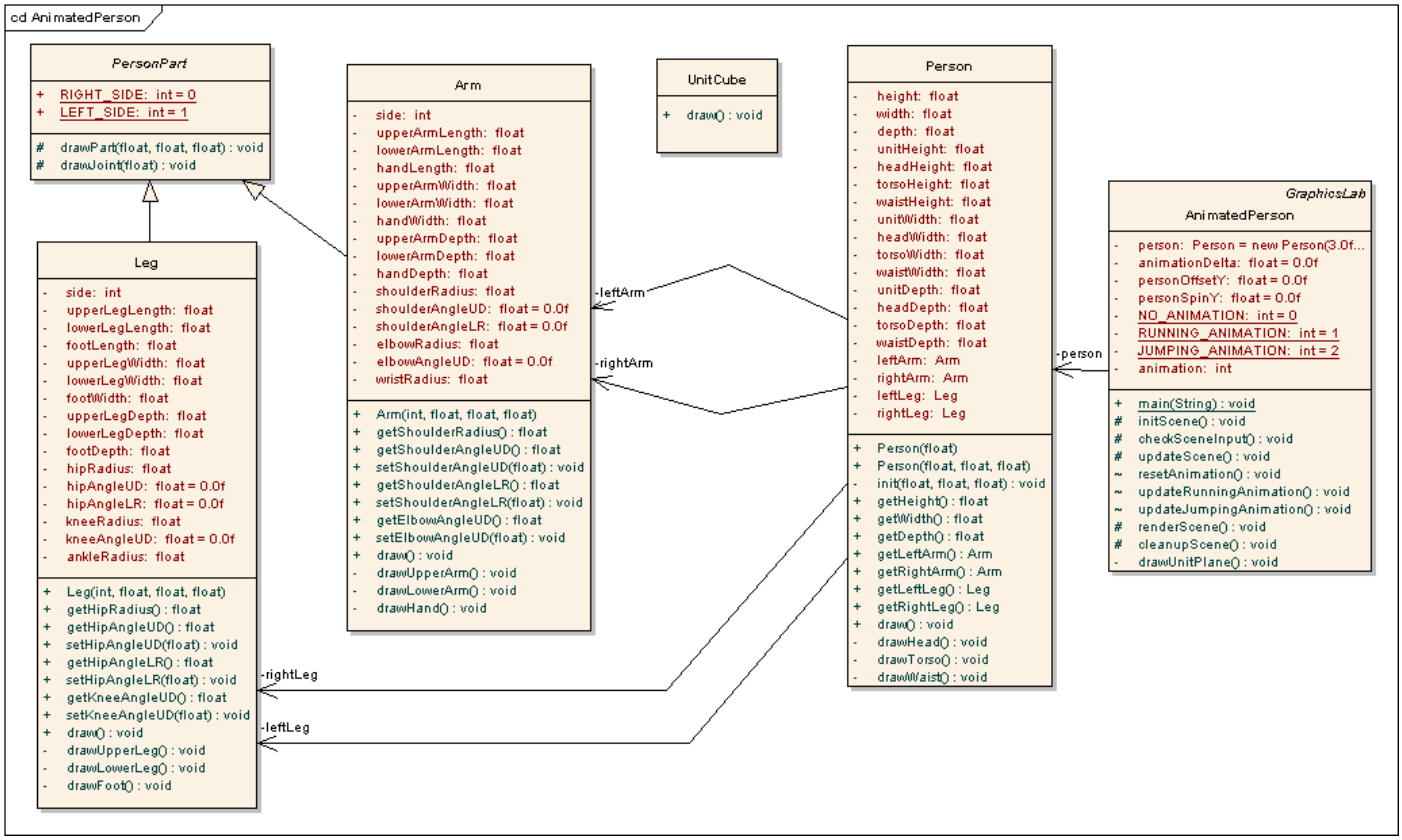


Figure 6: An overview of the Person and associated classes.

This example shows several features; for one there are several classes being used, shown in Figure 6. The example creates a `Person` object - note the constructor - take a look at the person class definition – there are two constructors available.

➥ Create a version of the person to mimic Dan (short and a bit wider than is ideal!).

The main benefit of using other classes for objects in your scene is the hiding of complexity in these objects; if all the code for this example were in a single class it would be very difficult to read; here the person is responsible for creating and rendering themselves. The animation is all done in the main class, so it is easier

to follow, but this could also have been delegated (indeed it might have been easier to follow if it had).

➥ Can you get the person to wave (or make some other gesture)? This is quite challenging, so don't try this unless you have time.

➥ Can you make the person look rather more realistic; either using more materials or maybe a texture or two? Again this is only for the dedicated!

Summary

These labs have taken you from a basic 3D Java application using OpenGL and LWJGL to a quite complex animation. We have covered a lot of material. I really want to emphasise two things:

1. Graphics requires that you use pen and paper; you need to start with the idea on paper; don't try to hack the code. This means a scene graph and a sketch of the world space and the objects therein.
2. Feel free to extend what we have provided for you in undertaking the coursework; these labs are meant to provide examples that you can learn from; make sure you read the code and comments – this should explain most things!

If you have any comments feed them back to us; that is the only way we can make things better!