# Dire Dawa Institute of Technology
# Department of Computer Science
# Chapter 5 Handout – The Viewing Pipeline (part I)
# Compiled by Gaddisa Olani, 2016

## 1. Introduction

In this handout we introduce the concept of a viewing pipeline. First we talk in general terms, i.e. not specific to any particular graphics package. The different transformations involved in the pipeline are described and the different coordinate systems involved are explained. Next, we describe in detail the operation of the OpenGL viewing pipeline for 3-D graphics. The OpenGL functions used to manipulate the pipeline are introduced and example code provided.

## 2. The 3-D Graphics Pipeline

Every 3-D graphics package features some kind of *viewing pipeline*. A viewing pipeline is just a sequence of transformations that every primitive has to undergo before it is displayed. Although the details of these transformations can vary slightly from package to package, they are generally very similar. In this section we will outline the different component transformations of the general 3-D graphics viewing pipeline.

Figure 1 to the right shows a breakdown of the different transformations and coordinate systems involved in the pipeline.
Note that we refer to a different *coordinate system* after each transformation. Every transformation can be thought of as defining a new coordinate system. For example, if we apply a translation we can think of this transformation as translating the origin of the coordinate system to define a new (shifted) coordinate system. The same concept applies to rotations, scalings and other transformations.

In the following sections we examine each of the transformations in the pipeline in turn.

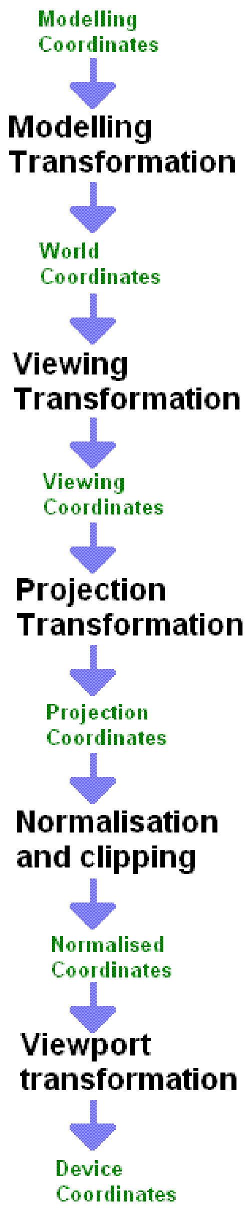


**Figure 1 - The 3-D Graphics Viewing Pipeline**

## 2.1 **The Modelling Transformation**

All primitives start off in their own private coordinate system. We call this the *modelling coordinate system*. *Modelling* refers to the process of building our 3-D scene from a number of basic primitives. Typically these primitives must be transformed to get them in the correct position, scale and orientation to construct our scene. For example, Figure 2 shows a *wheel* primitive being transformed in four different ways to form the wheels of a car.

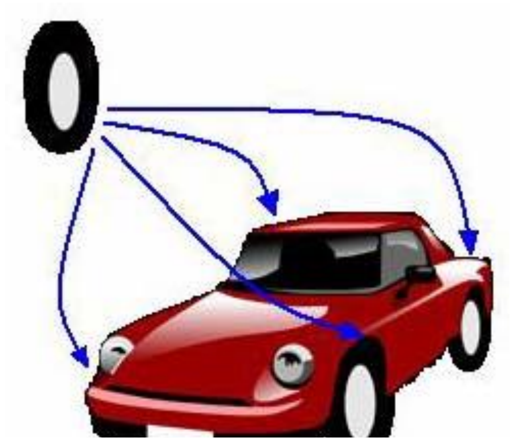


**Figure 2 - The Modelling Transformation**

Typically our primitives will be more basic than wheels (for example, they may be lines or polygons) but the same principle applies: the *modelling transformation* is used to transform primitives so that they construct the 3-D scene we want. Modelling transformations are normally different for each primitive.

Once we have transformed all primitives to their correct positions we say that our scene is in *world coordinates* (See Figure 3). This coordinate system will have an origin somewhere in the scene, and its axes will have a specific orientation. All primitives are now defined in this world coordinate system.

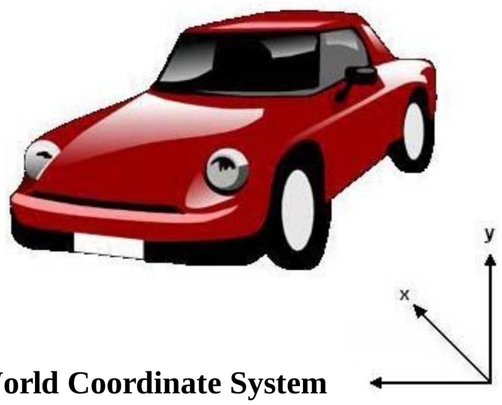


**Figure 3 - The World Coordinate System**

## 2.2 The Viewing Transformation

In order to be able to render an image of a scene we need to define a *virtual camera* to take the picture. Just like cameras in real life, virtual cameras must have a position in space, a direction (which way is it pointing?) and an orientation (which direction is 'up'?). These parameters are typically represented by a camera position, a 'look' vector and an 'up' vector (see Figure 4).

The *viewing transformation* represents the positioning, direction and orientation of the virtual camera. This transformation defines a new coordinate system known as the *viewing coordinate system*. In this coordinate system the origin is normally at the position of the virtual camera, the camera 'looks' along the z-axis, and the y-axis is 'up'. This is illustrated in Figure 5.

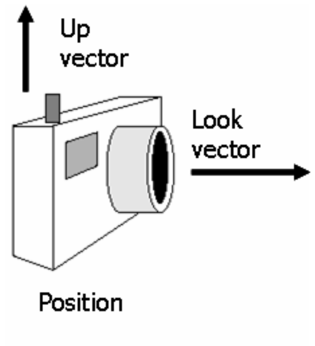


**Figure 4 - The Virtual Camera**

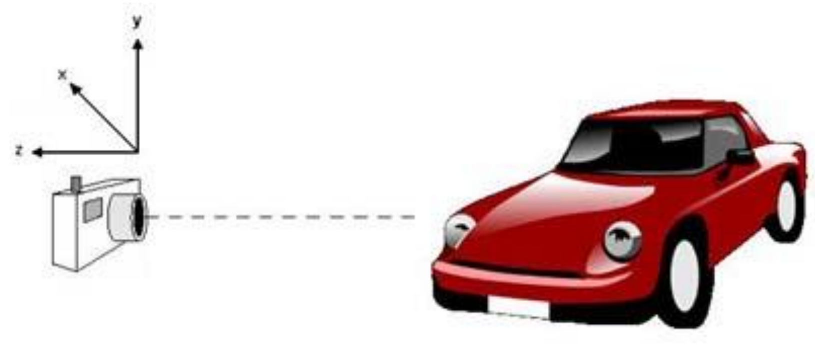


**Figure 5 - The Viewing Coordinate System: the virtual camera 'looks' along the z-axis, and the y-axis is 'up.**

### 2.3 The Projection Transformation

Now we know where our primitives are in relation to the virtual camera, we are in a position to 'take' our picture. Taking a picture involves 'projecting' the 3-D primitives onto a 2-D image plane. To be able to do this, we need to know something about the type of camera we have. For example, does it have a wide-angle or zoom lens fitted? The transformation that handles all of this is known as the *projection transformation*.

Here it is necessary to introduce some extra theory about projection transformations. Broadly speaking, there are two different types of projection transformation: parallel and perspective projections.

#### 2.3.1 Parallel Projections

For all projection transformations we can imagine the 2D image plane as being positioned somewhere in 3-D space (in the viewing coordinate system). We can draw *projection lines* that connect points in 3-D space to their corresponding points on the 2-D image plane. The difference between the two basic types of projection lies in what happens to these projection lines after they pass through the image plane. Consider Figure 6(a). Here we can see a line primitive in 3-D viewing coordinates being projected onto the image plane. We can see that the projection lines of the two end-points are *parallel*: after they have passed through the image plane, they continue to infinity without ever meeting. This type of projection is known as a *parallel projection*.

Parallel projections can be divided into two subtypes: *orthographic* (or *orthogonal*) projections and *oblique* projections. The difference between these two types is illustrated in Figure 6(b): in an orthographic projection then projection lines intersect with the image plane at right-angles (they are *orthogonal*), whereas in an oblique projection they intersect at an angle (they are *oblique*).

Parallel projections have the property that objects that are further away from the virtual camera do not appear smaller in the projected image. We can see this intuitively from Figure 6(a). If the projection lines are parallel they are always the same distance apart, so whenever they intersect with the image plane their distance apart will be identical. Therefore the size of the image of the line will be the same, however far away the line is.

Both orthographic and oblique projections are commonly used in engineering applications, particularly CAD software. Engineers often want to visualise designs in 3-D, but want to preserve the relative proportions of the objects. Parallel projections do this.

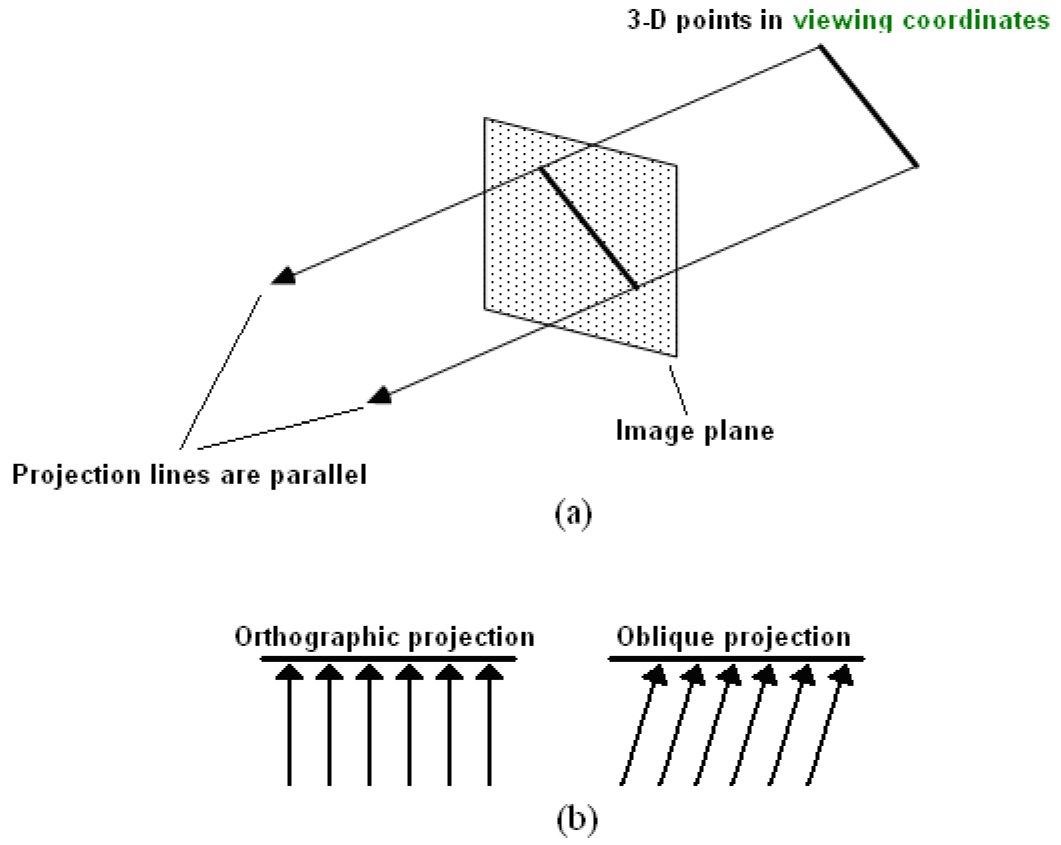
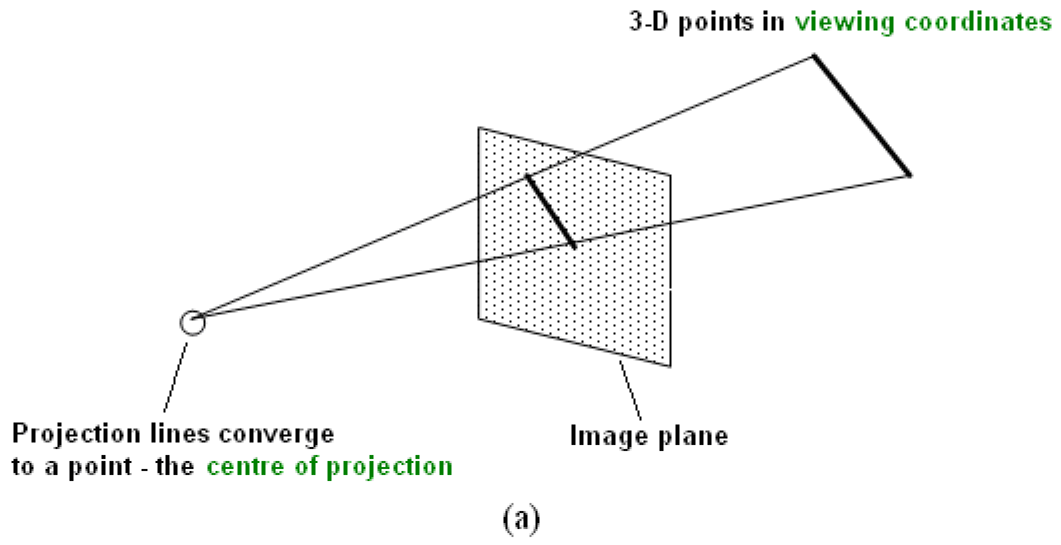

3-D points in viewing coordinates
Image plane
Projection lines are parallel
(a)
Orthographic projection
Oblique projection
(b)

**Figure 6 - Orthogonal and Oblique Parallel Projections**

### 2.3.2 Perspective Projections

The alternative to parallel projections is the *perspective projection*. Perspective projections are probably more familiar to you, even if you don't know it. In a perspective projection all projection lines converge to a point, the *centre of projection*. Figure 7(a) shows the same line primitive being projected onto the image plane using a perspective projection. Here we can see that after passing through the image plane the projection lines of the two end points intersect at a point.

Perspective projections have the property that objects that are further away from the camera <u>do</u> appear smaller. This is illustrated in Figure 7(b). The same sized line is projected twice, using two different distances from the virtual camera (centre of projection). The projected images are shown in red. We can clearly see that the bottom line (i.e. the furthest away) appears smaller on the image plane.


3-D points in viewing coordinates
Projection lines converge to a point - the centre of projection
Image plane
(a)

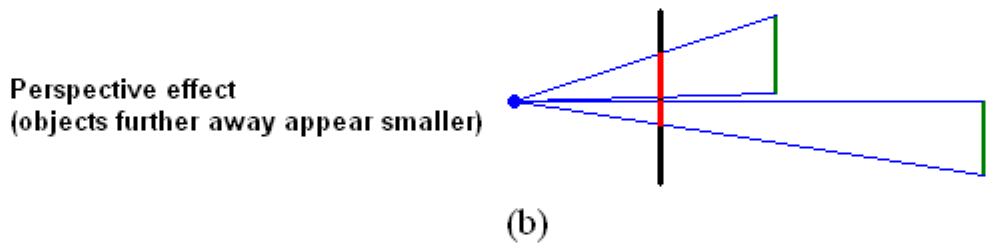


**Figure 7 - Perspective Projection**

The human eye uses a perspective projection, as do all commercial cameras. Parallel projections are a theoretical model of projection that can be useful in some computer graphics applications. Parallel projections are also faster to compute, and so are used in some real-time applications such as computer games.

Whether we use a parallel or perspective projection, afterwards our points are in *projection coordinates*. The next process in the pipeline is to eliminate all points that cannot be seen by the virtual camera.

**Note**: Although in theory a projection is a 3-D to 2-D transformation, in practise most graphics packages preserve the 3rd coordinate (the z-coordinate) in projection coordinates. That is, the x and y coordinates represent the projected coordinate on the 2-D image plane, but the z-coordinate is preserved and left unchanged. The reason for this is so that hidden surface elimination techniques can access the 'depth' of a projected point to determine whether or not it is occluded by other points.

## 2.4 Normalisation and Clipping

Before our final image can be generated, we need to decide which primitives (or parts of primitives) are 'inside' the picture and which ones are 'outside'. In computer graphics, this process is known as *clipping*. Clipping will be dealt with in detail in the next chapter, but we need to introduce a few basic concepts here so that we can understand the nature of the normalisation/clipping transformation.

Clipping is performed by defining a number of clipping planes. All points that are on one side of a clipping plane will be *clipped*, or removed from the scene. The image itself effectively defines 4 clipping planes: any point whose projection line intersects the image plane outside the image bounds (left, right, bottom or top) will not be visible by the camera. However, in addition we define 2 extra clipping planes: *near* and *far*. Therefore in total there are 6 clipping planes: *near*, *far*, *left*, *right*, *bottom* and *top*. These 6 planes together form a bounded volume, inside which points will be visible by the virtual camera, and outside which points will not be seen. This volume is known as a *view volume*, and is illustrated in Figure 8.

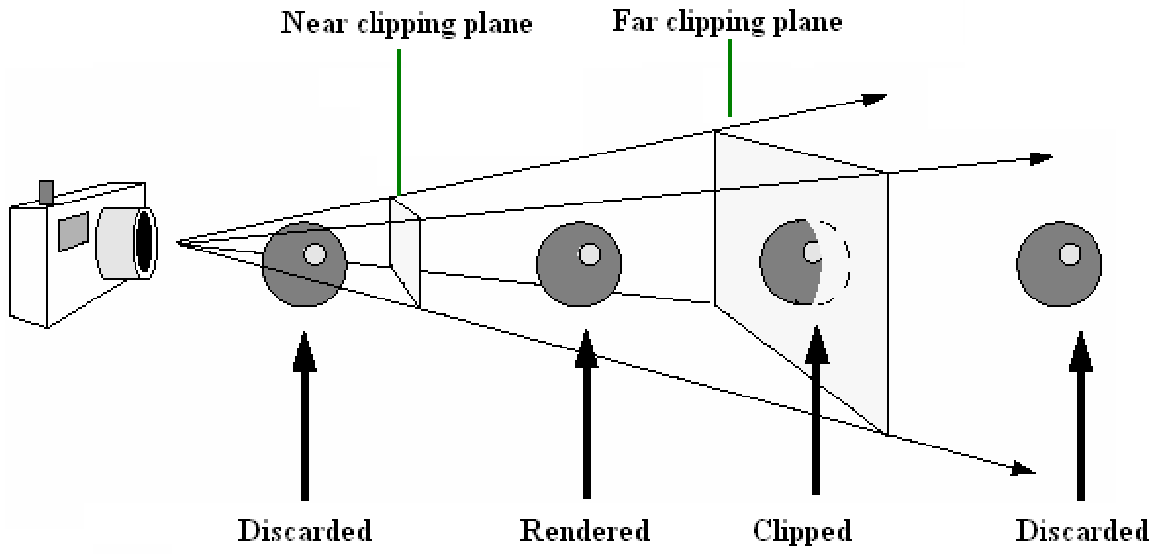


**Figure 8 - Near and Far Clipping Planes**

Different types of projection transformation lead to different shapes of view volume. For example, Figure 9 shows that using a perspective projection forms a view volume in the shape of a *frustum*, whereas Figure 10 shows that a parallel (orthographic) projection leads to a view volume that is a *cuboid* (a 3-D rectangle).

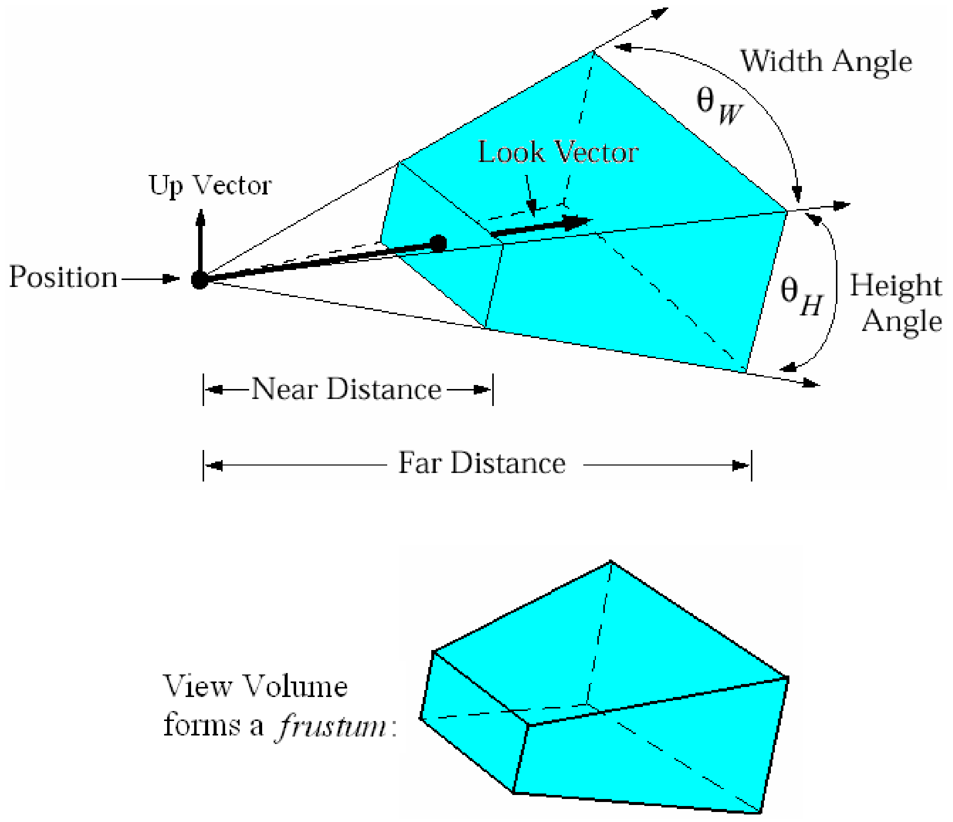


**Figure 9 - The View Volume of a Perspective Projection**

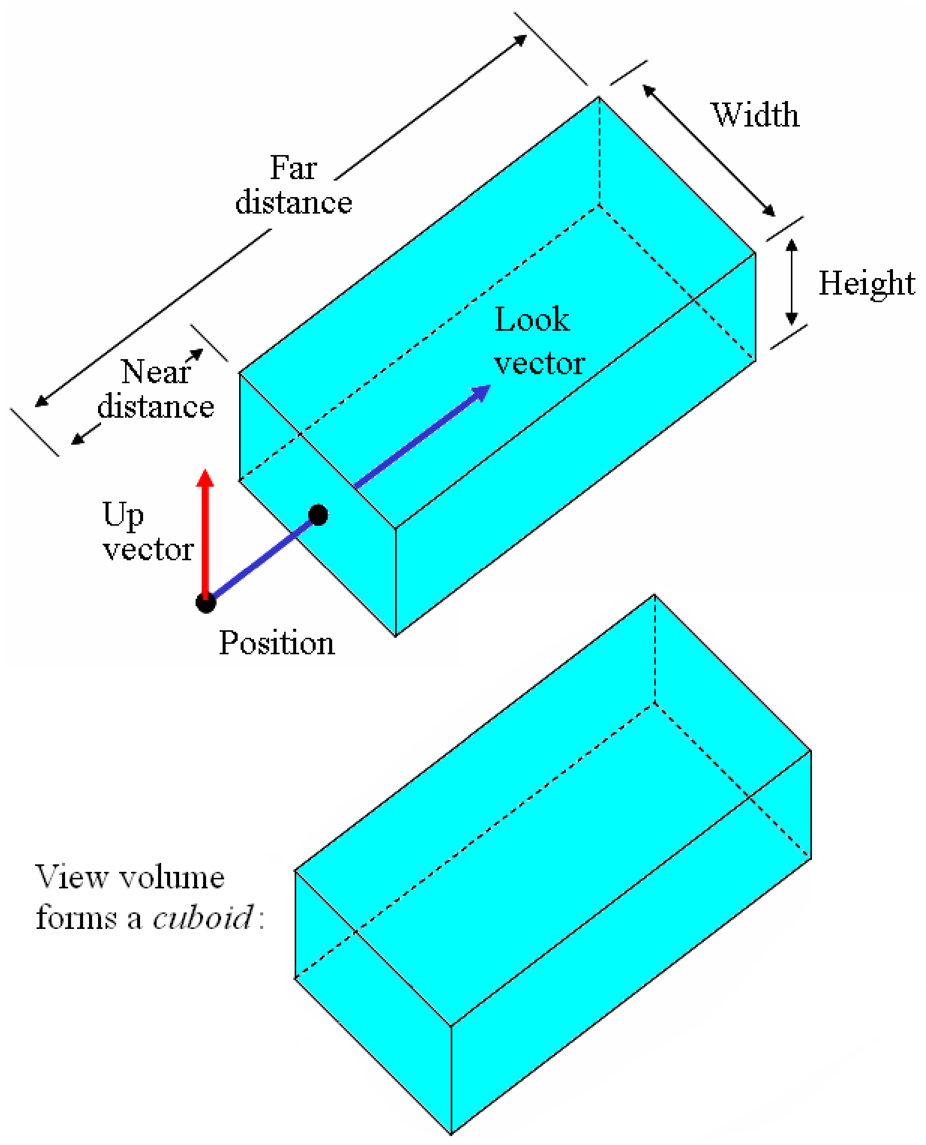


**Figure 10 - The View Volume of a Parallel Projection**

The process of clipping is more efficient if it takes place in a *normalised* coordinate system. Normalising means scaling all coordinates so that they range between 0 and 1 (or occasionally -1 and +1). Therefore the normalisation and clipping transformation is responsible for both removing points that cannot be seen by the virtual camera, and for scaling the view volume so that all coordinates are normalised. For example, if we scale to the range [0,1] then points at the near clipping plane will have a z-coordinate of 0 whilst points at the far clipping plane will have a z-coordinate of 1. The same applies to the x-coordinate (left and right clipping planes) and the y-coordinate (bottom and top clipping planes). Once the view volume has been normalised, and clipping has taken place, our remaining points are in *normalised coordinates*. All that is left now is to draw our picture into the display window.

## 2.5 The Viewport Transformation

When drawing our image into the display window, we may not want it to fill up the entire display. For example, we may want to draw several different (or identical) images in the same display. Therefore, whenever we draw a picture, we need to specify which part of the display we want to use for drawing. This is known as the *viewport*. Figure 11 illustrates this concept: on our computer's monitor, OpenGL creates a display window, or *screen window*. If we wish, we can draw in the whole of the screen window, but we can also specify a sub region of it for drawing. Whatever region we specify, it is known as the *viewport*.

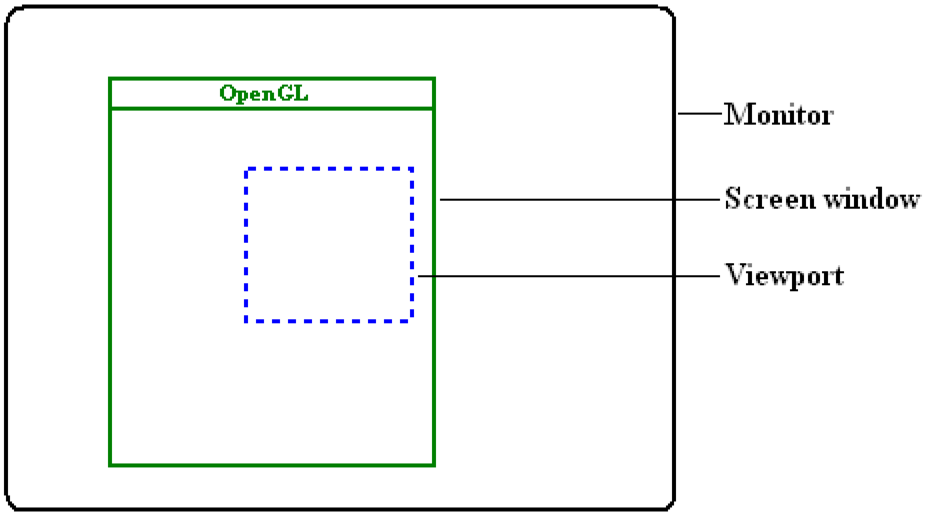


**Figure 11 - The Viewport**

The *viewport transformation* defines the mapping from our normalised coordinate system onto the viewport in the display window. After undergoing this transformation, our final rendered image is in *device coordinates* (i.e. the coordinate system of the display device).

## 3. The OpenGL Viewing Pipeline

Now we will examine the viewing pipeline as it is defined and implemented by the OpenGL graphics package. OpenGL defines 3 matrices in its viewing pipeline:

- The *modelview* matrix: This matrix combines the effects of the modelling and viewing transformations in the general graphics pipeline. Before this transformation, each primitive is defined in its own coordinate system. After it, all primitives should be in the viewing coordinate system.
- The *projection* matrix: This represents the projection of 3-D viewing coordinate onto the image plane. Like most graphics packages, OpenGL preserves the z-coordinate after projection and normalises the resulting coordinate system. Clipping is performed automatically based on the bounds we specify for the view volume.
- The *viewport* matrix: This matrix defines the part of the display that will be used for drawing.

In the subsequent sections we will examine how we can define each of these matrices using C++/OpenGL code. Remember that OpenGL will *postmultiply* any matrices we specify by the current matrix (i.e. the one on top of the matrix stack). This means that if we want to specify a sequence of transformations for any of the viewing pipeline matrices, we must define them in reverse order.

## 3.1 The Modelview Matrix

We mentioned in the last chapter that OpenGL maintains a matrix stack that we can use to store matrices that we may need later. In fact, OpenGL maintains one matrix stack *for each matrix in the viewing pipeline*. In other words, there is one *modelview* matrix stack, one *projection* matrix stack and one *viewport* matrix stack. Therefore, before making any modifications to our current matrix (the top one on the stack) we must tell OpenGL which matrix we are talking about. The following line of code specifies to OpenGL that any subsequent transformations that we specify will apply to the modelview matrix:

```
glMatrixMode(GL_MODELVIEW);
```

Now we can use any of the following OpenGL functions to modify the current modelview matrix:

- *glTranslate**
- *glRotate**
- *glScale**
- *glLoadMatrix**
- *glMultMatrix**
- *gluLookAt*

The first five of these we have already seen in the last chapter. These are typically used to define the *modelling* part of the modelview matrix. That is, they are used to transform the primitives to form our 3-D scene in world coordinates. However, the last, *gluLookAt*, needs further explanation. This function is used to define the *viewing* part of the transformation. Recall that the viewing transformation defines the position, direction and orientation of a virtual camera. The basic format of *gluLookAt* is as follows:

```
gluLookAt(x0, y0, z0, xref, yref, zref, Vx, Vy, Vz)
```

The first three arguments define a position *(x0, y0, z0)* in 3-D world coordinates. This represents the position of the virtual camera. The next three arguments define another position in 3-D world coordinates: *(xref, yref, zref)*. This is the position that the camera is 'looking' at. For example, if we specified *(0,0,-10)* at the camera position and *(0,0,0)* as the 'look' position, our virtual camera would be looking along the positive *z*-axis. Finally we have the last three arguments *(Vx, Vy, Vz)*. These three values form a 3-D 'up' vector that defines the orientation of the camera. If the vector is not perpendicular to the 'look' direction then it is projected so that it is perpendicular.

For example, the following 3 lines define a virtual camera positioned at *(8, 0, 8)* in world coordinates, looking at the origin, and with the *y*-axis as 'up'.

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(8,0,8,0,0,0,0,1,0);
```

Note that we have initialised the current modelview matrix to the identity matrix before defining the viewing matrix. It is always necessary to do this, because OpenGL matrix functions *postmultiply* by the current matrix, so if this current matrix is not initialised the results will be undefined.

If you do not specify a viewing matrix in OpenGL the default parameters are:

- *(x0, y0, z0) = (0, 0, 0)*
- *(xref, yref, zref) = (0, 0, -1)*
- *(Vx, Vy, Vz) = (0, 1, 0)*

That is, the camera is positioned at the origin, pointing along the negative *z*-axis, with the *y*-axis as 'up'. For many applications this may be sufficient, so sometimes OpenGL programs will not define a viewing matrix.

Recall from Figure 1 that the viewing transformation occurs <u>after</u> the modelling transformation. However, because OpenGL postmultiplies all matrices, it must be specified <u>before</u> the modelling transformation. The viewing transformation, if specified, will normally be common to all primitives in the scene. However, the modelling transformation is normally different for each primitive. Therefore, it is normal to define a single viewing transformation to begin with, and then to modify it with individual modelling transformations for each primitive. But because we only have a single current matrix, whenever we define modelling transformations for a given primitive it will overwrite the common viewing transformation, which we need to keep for subsequent primitives. How can we 'remember' this viewing transformation so that we don't need to redefine it for each primitive?

The answer to this question is to use the *matrix stack*. We can remember the viewing transformation by *pushing* it onto the modelview matrix stack, and then when we need it for the next primitive, we *pop* it back again. The standard sequence of events is therefore:

- *Define the viewing matrix*
- *For each primitive we want to draw:*
    - *Push the viewing matrix onto the stack*
    - *Postmultiply the viewing matrix by the modelling transformations for this primitive*
    - *Draw the primitive(s)*
    - *Pop the viewing matrix back from the stack*

For example, the code example shown below displays two primitives (both 2-D rectangles) using the same viewing transformation but different modelling transformations.

```
// initialize modelview matrix to identity
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();

// virtual camera is at (8,0,8) looking at the origin
// with the y-axis as 'up'
gluLookAt(8,0,8,0,0,0,0,1,0);
glClear(GL_COLOR_BUFFER_BIT); // clear screen

// draw primitives
glPushMatrix();         // remember viewing matrix
glRotatef(45,0,0,1);    //rotate this primitive
glRecti(0,0,50,50);     // draw rectangle
glPopMatrix();          // restore viewing matrix

glPushMatrix();         // remember viewing matrix
glTranslatef(200,0,0); // translate this primitive
glRecti(0,0,50,50);     // draw rectangle
glPopMatrix();          // restore viewing matrix

glFlush();              // send all output to display
```

## 3.2 The Projection Matrix

Depending on the type of projection required, OpenGL provides 4 alternative functions for specifying the projection matrix:

- *gluOrtho2D*
- *glOrtho*
- *gluPerspective*
- *glFrustum*

<u>*gluOrtho2D*</u>

OpenGL provides two functions for specifying orthographic parallel projections (it does <u>not</u> support oblique projections). The *gluOrtho2D* function is intended for use with 2-D graphics only, as it does not allow you to specify near and far clipping planes. The basic format of *gluOrtho2D* is:

```
gluOrtho2D (xwmin, xwmax, ywmin, ywmax)
```

The four arguments specify the *x*-coordinates of the left and right clipping planes, and the *y*-coordinates of the bottom and top clipping planes. All coordinates are specified in the *viewing coordinate system* (i.e. the virtual camera is at the origin and looking along the negative z-axis, and the y-axis is 'up'). In 2-D graphics we just ignore the third coordinate (it is normally set to zero), so orthographic projections are ideal and no near and far clipping planes are required.

For example, the lines of code shown below define a 2-D orthographic projection that will display all primitives with *x*-coordinate between 0 and 640, and *y*-coordinate between 0 and 480. Note that, just as with the modelview matrix, the first thing we have to do it tell OpenGL which matrix we are referring to, and initialise it to the identity matrix.

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(0.0, 640.0, 0.0, 480.0);
```

## *glOrtho*

An alternative function for defining orthographic projections is *glOrtho*. This is very similar to *gluOrtho2D* except that it allows us to specify near and far clipping planes, in addition to the left, right, bottom and top ones. The basic format is:

```
glOrtho (xwmin, xwmax, ywmin, ywmax, dnear, dfar)
```

Here the six arguments represent the x-coordinates of the left and right clipping planes, the y-coordinates of the bottom and top clipping planes, and the z-coordinates of the near and far clipping planes. Again, all coordinate are specified in the viewing coordinate system.

The following code specifies an orthographic projection that defines a view volume that is a *24x24x30* cuboid. The left and right bounds of this view volume are at *x=-12* and *x=+12*, the bottom and top bounds are at *y=-12* and *y=+12*, the near clipping plane is at the virtual camera position (*z=0*) and the far clipping plane is *30* units away from the camera in the viewing direction. Since the viewing direction of the viewing coordinate system is the negative z-axis in OpenGL, this means that the far clipping plane is positioned at *z=-30* in this example.

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
// arguments are left, right, bottom, top, near, far
glOrtho(-12, 12, -12, 12, 0, 30);
```

## *gluPerspective*

If a perspective projection is required, OpenGL provides two alternative functions. The first is *gluPerspective*. The basic format is:

```
gluPerspective (theta, aspect, dnear, dfar)
```

Here, `theta` represents the *height angle* of the perspective projection. The height angle is the angle between the top and bottom bounds of the image (see Figure 9). The next argument, `aspect`, is the *aspect ratio* of the image (the ratio of the width to the height). If OpenGL knows the height angle and the aspect ratio it is straightforward to determine the width angle. The last two arguments, `dnear` and `dfar`, are the *z*-coordinates of the near and far clipping planes.

As an example, the following code defines a perspective projection with a height angle of $45^o$. The image has an aspect ratio of *1.0* – this means that the width of the image is the same as the height, and therefore the width angle must also be $45^o$. The near clipping plane is positioned at *z=0*, or the origin of the viewing coordinate system, and the far clipping plane is at a distance of *50* from the camera. For same reasons given above, this positions it at *z=-50*.

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(45, 1.0, 0.0, 50.0);
```

*glFrustum*

Recall that perspective projections lead to a view volume in the shape of a frustum. The final OpenGL perspective projection routine allows you to specify the vertices of this frustum directly. The basic format is:

```
glFrustum (xwmin, xwmax, ywmin, ywmax, dnear, dfar)
```

Here, the first 4 arguments are the *x* and *y* coordinates of the left, right, bottom and top clipping planes *at the near clipping plane*. The last two arguments, `dnear` and `dfar`, represent the *z*-coordinates of the near and far clipping planes. Therefore the positions of the four corners of the near clipping plane are (in viewing coordinates):

- *(xwmin, ywmin, dnear)*
- *(xwmin, ywmax, dnear)*
- *(xwmax, ywmin, dnear)*
- *(xwmax, ywmax, dnear)*

The positions of the corners of the far clipping plane can be determined from these coordinates, the centre of projection (which is the origin) and the value of `dfar`.

For example, the code example given below specifies a perspective projection in which the near clipping plane is defined by the four corners *(-10, -10, -5)*, *(-10, 10, -5)*, *(10, -10, -5)* and *(10, 10, -5)*. Note that the *z*-coordinates are negated. The `dnear` argument specifies the distance of the near clipping plane from the virtual camera, and the virtual camera looks along the negative *z*-axis of the viewing coordinate system. The far clipping plane is at a distance *50* from the camera so it has a *z*-coordinate of *-50*.

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluFrustum(-10,10,-10,10,5,50);
```

## 3.3 The Viewport Matrix

The *viewport* matrix is defined slightly differently to the *modelview* and *projection* matrices. There is only one function for defining the viewport matrix so we do not need to use *glMatrixMode* to tell OpenGL which matrix we are referring to – it is unambiguous. The function is *glViewport* and its basic format is:

```
glViewport(xmin, ymin, xsize, ysize);
```

Referring to Figure 11, *(xmin, ymin)* is the bottom-left corner of the viewport, in the coordinate system of the screen window (whose origin is at the bottom-left). The arguments `xsize` and `ysize` are the size of the viewport in pixels.

For example, assuming that we have a screen window of size 640x480, the following line defines the viewport to be of size 300x300 and centred in the screen window.

```
glViewport(160, 90, 300, 300);
```

**Summary**

The following points summarise the key concepts of this chapter:

- The viewing pipeline is a sequence of transformations that every primitive must undergo in order to be displayed.
- In general terms, the viewing pipeline consists of the following transformations:
    - *Modelling transformation*
    - *Viewing transformation*
    - *Projection*
    - *Normalisation and clipping*
    - *Viewport transformation*
- The *modelling transformation* is used to transform primitives relative to each other to form a 3-D scene. After the modelling transformation all primitives are in *world coordinates*.
- The *viewing transformation* represents the positioning, direction and orientation of the *virtual camera* that will take the picture of the scene. After the viewing transformation all primitives are in *viewing coordinates*.
- The projection transformation projects 3-D points onto a 2-D image plane. After projection the primitives are in *projection coordinates*.
- Projection transformations can be either *parallel projections* or *perspective projections*.
- Parallel projections can be either *orthogonal* (*orthographic*) or *oblique* parallel projections.
- Most graphics packages preserve the *z*-coordinate during projection for use by *hidden surface elimination* techniques later in the pipeline.
- The *normalisation and clipping* transformation *normalises* the *view volume* of the projection, i.e. scales coordinates to the range [0,1] or [-1,1]. After this transformation the primitives are in *normalised coordinates*.
- The *viewport transformation* maps *normalised coordinates* to *device coordinates*. This involves drawing the projected primitives onto a *viewport* of the *screen window*.
- In OpenGL, the *modelling* and *viewing* transformations are combined into a single matrix: the *modelview* matrix.
- In OpenGL, the *projection* transformation is represented by the *projection matrix*. Clipping and normalisation occur automatically.]
- In OpenGL, the *viewport* transformation is represented by the *viewport matrix*.
- In OpenGL, the *glMatrixMode* function is used to tell OpenGL which *current matrix* will be affected by subsequent matrix functions (e.g. the modelview matrix or the projection matrix).

- The OpenGL *modelview matrix* can be manipulated by the following functions:
    - *glLoadIdentity*

    - *glLoadMatrix*
    - *glMultMatrix*
    - *glTranslate**
    - *glRotate**
    - *glScale**
    - *gluLookAt*
- The OpenGL *projection matrix* can be defined using the following functions;
    - *gluOrtho2D*
    - *glOrtho*
    - *gluPerspective*
    - *glFrustum*
- The OpenGL *viewport matrix* can be defined using the *glViewport* function.

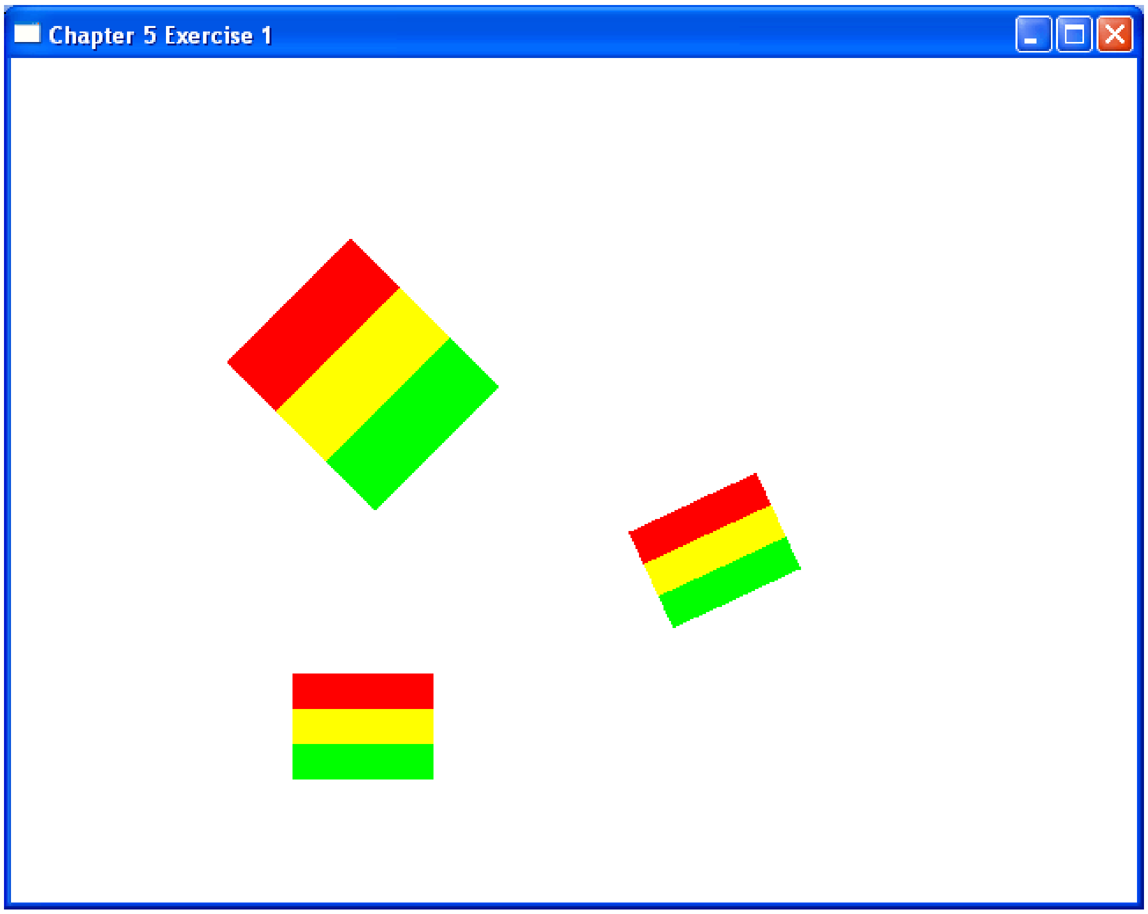
**Exercises**

1) Assuming we have a routine called *drawFlag()* that draws a 2-D Ethiopian flag at the origin, write OpenGL/C++ code for a display event callback function to produce the image shown below. (You can assume that the projection and viewing transformations have already been defined – you only need to define modelling transformations and draw the flags.)

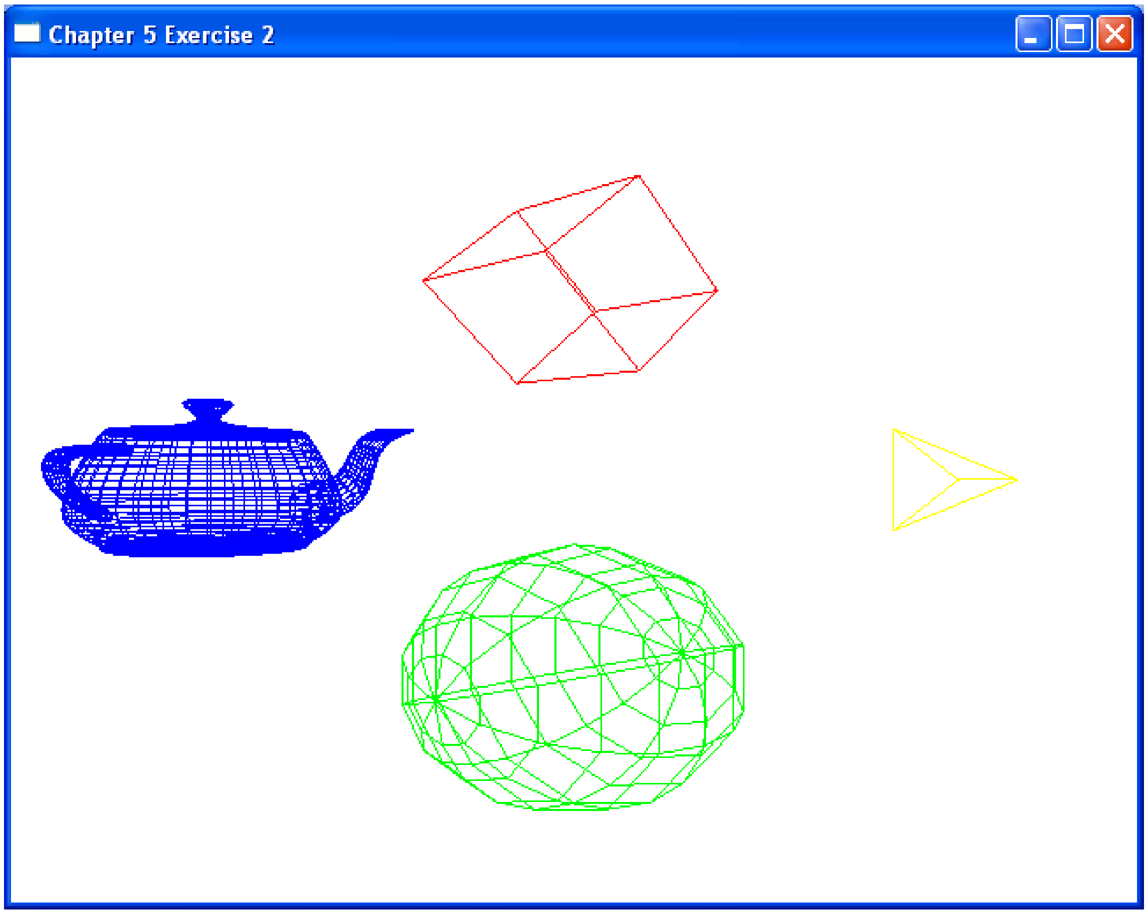
   The image consists of three flags. All should be drawn by calling *drawFlag()*. The 3 flags are transformed as described below:

   - Flag 1 is translated by (200,100) from the origin.
   - Flag 2 is rotated 25$^o$ anti-clockwise about the z-axis and then translated by (400,200).
   - Flag 3 is scaled by (1.25, 2.0), rotated 45$^o$ anti-clockwise and then translated by (200,300).

   You should use the matrix stack in your program.



2) This exercise is similar to the first one except that we will do 3-D graphics. You should use the OpenGL viewing pipeline and the glut routines for displaying 3-D objects to reproduce as closely as possible the scene shown below.



The following information will help you in reproducing the scene:

- You should use a perspective projection with height and width angles of 60$^o$. The near and far clipping planes should be at distances of 5 and 50 units from the virtual camera position.
- The virtual camera should be positioned at (8,0,8) and be directed at the origin with the y-axis being 'up'.
- The wireframe cube should be drawn in red, with a size of 2, rotated by 45$^o$ about the x-axis, and translated by (0,3,0) from the origin.
- The wireframe sphere should be drawn in green, with a radius of 2, and translated by (0,-3,0) from the origin.
- The wireframe teapot should be drawn in blue, with a size of 1.5, rotated by 30$^o$ about the y-axis, and translated by (-3,0,3) from the origin.
- The wireframe tetrahedron should be drawn in yellow, rotated by 45$^o$ about the x-axis, and translated by (3,0,-3) from the origin.

You should use the matrix stack in your program.

## Exercise Solutions

1) The display event callback function code is as follows:

```
void display (void) {
    glClear(GL_COLOR_BUFFER_BIT);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity

    // draw flag 1
    glPushMatrix();
    glTranslatef(200.0, 100.0,0.0);
    drawFlag();
    glPopMatrix();

    // draw flag 2
    glPushMatrix();
    glTranslatef(400.0,200.0,0.0
    glRotatef(25.0, 0.0, 0.0, 1.0;
    drawFlag();
    glPopMatrix();

    // draw flag 3
    glPushMatrix();
    glTranslatef(200.0,300.0,0.0);
    glRotatef(45.0, 0.0, 0.0, 1.0);
    glScalef(1.25, 2.0, 1.0);
    drawFlag();
    glPopMatrix();

    glFlush();
}
```

2) The complete code listing is as follows:

```
#include <GL/glut.h>
#include <stdlib.h>

#define SCREEN_WIDTH 640
#define SCREEN_HEIGHT 480

void myInit(void) {
    glClearColor(1.0, 1.0, 1.0, 0.0); // white background
    glColor3f(0,0,0); // black foreground

    // projection transformation
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(60.0, 1.0, 5.0, 50.0);

    // modelview transformation
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();

    // set up camera
    gluLookAt(8,0,8,0,0,0,0,1,0);
}

/* GLUT callback Handlers */

void drawObjects()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glMatrixMode(GL_MODELVIEW);

    // draw a wireframe cube
    glPushMatrix();         // save viewing matrix
    glColor3f(1.0,0.0,0.0);
    glTranslated(0,3,0);
    glRotated(45.0,1,0,0);
    glutWireCube(2.0);
    glPopMatrix();          // restore viewing matrix

    // draw a wireframe sphere
    glPushMatrix();         // save viewing matrix
    glColor3f(0.0,1.0,0.0);
    glTranslated(0,-3,0);
    glutWireSphere(2.0,10,10);
    glPopMatrix();          // restore viewing matrix
     // draw a wireframe teapot
    glPushMatrix();        // save viewing matrix
    glColor3f(0.0,0.0,1.0);
    glTranslated(-3,0,3);
    glRotated(30,0,1,0);
    glutWireTeapot(1.5);
    glPopMatrix();         // restore viewing matrix

    // draw a wireframe tetrahedron
    glPushMatrix();        // save viewing matrix
    glColor3f(1.0,1.0,0.0);
    glTranslated(3,0,-3);
    glRotated(45,1,0,0);
    glutWireTetrahedron();
    glPopMatrix();         // restore viewing matrix

    glFlush(); // send all output to the display
}

/* Program entry point */

int main(int argc, char *argv[])
{
    // normal OpenGL/glut initialisation
    glutInit(&argc, argv);
    glutInitWindowSize(SCREEN_WIDTH,SCREEN_HEIGHT);
    glutInitWindowPosition(10,10);
    glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB|GLUT_DEPTH);
    glutCreateWindow("Chapter 5 Exercise 2");
    glEnable(GL_DEPTH_TEST);

    glutDisplayFunc(drawObjects);
    myInit();

    glutMainLoop(); // enter the GL event loop
    return EXIT_SUCCESS;
}
```

Source: Computer Graphics with OpenGL,
Donald Hearn and M. Pauline Baker,
Prentice-Hall, 2004 (Third Edition)