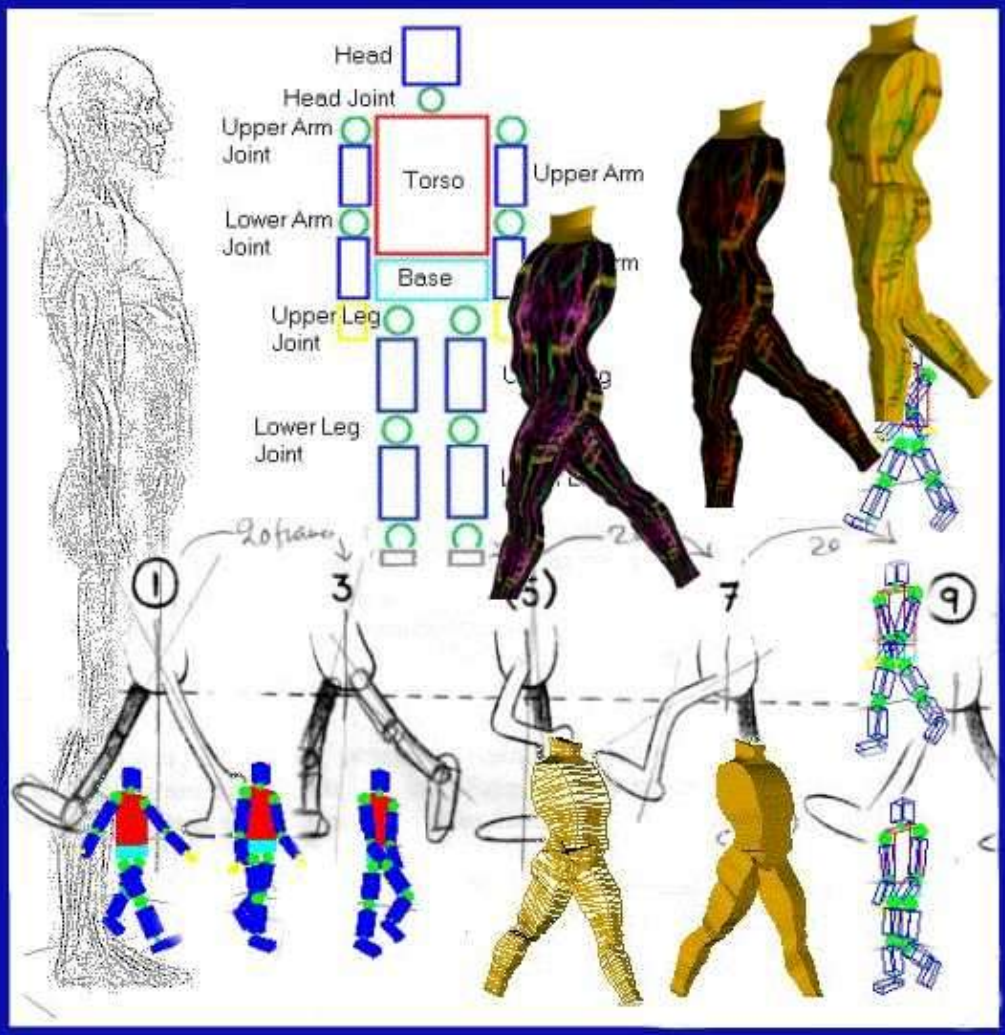
The Developers Gallery

www.dev-gallery.com



A 3D Case Study Using OpenGL

Copyright (C) Fotis Chatzinikos 1999

Table of Contents

Disclaimer

This document named *"A 3D Case Study using OpenGL"*, was initially written as part of a third year project in the University of Hull, by Fotis Chatzinikos.

This document is free for personal use. For commercial or academic use please contact the Developers Gallery Webmaster at: WebMaster@dev−gallery.com

For updates and the code that accompanies this tutorial please visit the Developers Gallery (www.dev−gallery.com)

Feel free to join our newsletter, so that you will be kept up to date with any additions to the site.

Document Version 1.0

Fotis Chatzinikos, August 29th, 2000

Copyright (C) Fotis Chatzinikos 1999

Chapter 1 – Introduction

1.1 About this report

This report was written as part of the final year project with the title *"A 3D Case study using OpenGL"*. In the following pages of this paper, a great deal of information can be found about several different aspects of this project.

Firstly, some information about the style conventions used during the development of this project report is provided. Some background work done mainly in the summer follows. The aims and objectives of this project are the following topic.

Further on, is a short report on what is OpenGL and why it was chosen for the development of this project. The discussions with the supervisor and the time plan of the two semesters also appears here.

Following, a discussion is done on the structure of this project and finally some comments are done on the structure of the accompanying compact disc, which contains all the work done, including this report.

1.2 Style Conventions

In this project report the following style conventions are used :

· The actual text of the report is written in Arial, size 12. Chapter headings are use Arial , size 20 and secondary heading are of size 16.

· Code is written in Courier New, size 10.

· OpenGL and glut command summaries are shaded with light blue boxes.

· Variables, arguments, parameter names, etceteras are in *Italics*.

· OpenGL functions start with 'gl', GLUT functions start with 'glut'.

· Constants of type GL_* or GLUT_* are predefined (OpenGL and GLUT specific).

Note: In the online version of the tutorial not all of the previous apply.

1.3 Background Material

During the summer (before the 5th Semester) some background work was done. This work included searching several Internet sites for information about OpenGL. There are quite a few sites with information on OpenGL but the most useful one proved to be *www/opengl.org*. At the particular site information is held about OpenGL documentation, specification definitions, developers, example programs, etceteras.

Some information was also needed on human walking in order to make the human model to walk. From Tony White's book on animation [1], the walking cycle of the human model was retrieved.

Some experimentation with OpenGL was done also before the beginning of the 5th semester in order to be familial with the particular graphics system.

Copyright (C) Fotis Chatzinikos 1999

1.4 Aims And Objectives

The title of this project is *"A 3D Case Study Using OpenGL"*, so one of the most defined aims of this project is to learn to use OpenGL. What is OpenGL and why it was chosen for the development of this problem are discussed later on. This may be the easiest identifiable aim of the project but is not the most important one.

More important aims are to understand the concept of 3D graphics. People may leave in a three dimensional world but building three dimensional applications is not the easiest thing somebody can do.

Another aim of this project is to learn how to model three dimensional hierarchical objects such as cars, articulated robot arms, humans etceteras. In few words objects with multiple moving parts that are related in some order.

In software engineering terms, a combination of the incremental and prototyping models was used in order to design and work with this project (divide and conquer). This model is based on the idea of constructing a simple and small system as soon as possible, as such a system is probably not complicated; and a simple (not complicated) system is probably correct. As the development of the project is continuing more parts are added to the initial system (Incremental Model). At any points that there is an uncertainty about which algorithm or technique should be used, different solutions can be tried out (Prototyping).

This software engineering model suits the particular project as one of the main objectives of this project is to produce an OpenGL tutorial, something that is clearly incremental.

This technique suits also the developer of this project as he prefers to have something working during the whole development time. Following this technique there was always the drawback of spending more time than designing the whole system and then implementing it, but there was no possibility of reaching the deadline without a working system.

Several Appendices are included with this report. The reason behind these appendices is to keep the length of the main report relatively short, without any loss of information. A short description of the appendices now follows.

· Appendix I : Using Borland C/C++ 5.02 to build an OpenGL DOS console WINDOWS program.

· Appendix II: Using the FLTK (Fast Light Tool Kit) GUI (Graphical User Interface) to construct buttons, dialogs, menu etceteras in C.

· Appendix III: Using Paint Shop Pro to retrieve the model data

· Appendix IV: Bibliography

1.5 What is OpenGL?

According to the OpenGL data sheet, OpenGL is an industry standard, stable, reliable and portable, evolving, scalable, easy to use and well−documented API.

But lets explain all these buzzwords. OpenGL is an industry standard (by now) as it has been available from 1992. The OpenGL specification is managed by an independent consortium, the OpenGL Architecture Review Board, some of its members being SGI (Silicon Graphics) and Microsoft.

OpenGL is available for more than seven years in a variety of systems. Additions to the specification (through extensions) are well controlled by the consortium and proposed updates are announced in time for developers to adopt changes. Backwards compatibility is also ensured.

Copyright (C) Fotis Chatzinikos 1999

OpenGL is reliable as all applications based on OpenGL produce consistent visual display results on any OpenGL API compliant hardware. Portability is also a fact as OpenGL is available in a variety of systems, such as PCs, Macintoshes, Silicon Graphics and UNIX based machines and so on. OpenGL is available also in different bindings, some of them being C and C++, Java and FORTAN.

OpenGL is evolving through its extensions mechanism that allows new hardware innovations to be accessible to the API, as soon as the developers have the hardware (and the extension) ready.

OpenGL is also scalable as it can run in a variety of computers, from ‘simple’ home systems to workstations and supercomputers. This is achieved through OpenGL’s hardware capabilities inquiry mechanism.

OpenGL is well structured with logical commands (a few hundred). OpenGL also encapsulates information about the underlying hardware, freeing the application developer from having to design hardware specific code.

OpenGL’s data sheet says that numerous books are available on the subject. Actually at the start of 1998 only two of them were widely available, the *OpenGL programming guide* [3], and *the OpenGL Super Bible* [4]. The truth is that at the end of 1998 a few more books appeared about OpenGL and that the World Wide Web has enough resources available for free.

Lets see now the programmers view of Open. To the programmer OpenGL is a set of commands. Firstly he opens a window in the frame buffer into which the program will draw. After this some calls are made to establish a GL context. When this is done, the programmer is free to use the OpenGL commands to describe 2D and 3D objects and alter their appearance but changing their attributes (or state). The programmer is also free to manipulate directly the frame buffer with calls like read and write pixels.

So why was OpenGL chosen for the development of this project. Why OpenGL instead of DirectX, GKS or XGKS ?

The answer is a combination of the just mentioned OpenGL advantages and the fact that DirectX is quicker (only at the moment) but OpenGL is more precise and that GKS (and X−GKS) are years in the market but are not as platform independent as OpenGL (OpenGL is also free). And for scientific visualisation, virtual environments, CAD/CAM/CAE, medical imaging and so on (not just games) precision and platform independence are the key features.

1.6 Discussions With The Supervisor−Time Plan

Supervisor meeting took place every Monday during the first semester. At each meeting the supervisor was told of new developments during the previous week. These developments were discussed and on one occasion a change in the program was made (the program made to read data from a file). Another decision that may be of some importance is, that it was agreed that after finish the modelling of the human (chapter 4) no further improvements were to be done, until texture mapping (the next chapter) was finished.

1.7 The structure of this project

The structure of this project is such that a newcomer to three−dimensional graphics and OpenGL can follow easily, building up knowledge before moving on to more complicated concepts, in other words this project is written in the form of an OpenGL tutorial.

The topic of the second chapter is simple window construction, as OpenGL needs a graphical (windowing) operating system, and the introduction of modelling and projection transformations. The reader first learns how to open windows using OpenGL, and then a discussion follows on modelling transformations like rotation, scaling and translation and projection transformations like orthographic and perspective viewing.

Copyright (C) Fotis Chatzinikos 1999

In the third chapter a first attempt is made to create a simple model of a man and the appropriate animation cycle, which resulted in a model constructed from basic geometrical shapes (spheres and cubes). Previously acquired knowledge of modelling and projection transformations is used in order to construct this simple model. At first the example programs use data that are ‘hard−wired’ in the program, but latter on the examples become data−driven.

The model of a man was chosen because it is an interesting case. Firstly, it is a hierarchical model, meaning that there are many interrelations between certain parts of the body such that cause the rotation and movement of body parts when other, higher in the hierarchy parts are moving. Secondly, the animation of a human walk cycle is a very interesting topic that is still a research topic. In this project a simple, but effective animation technique was chosen, based on the idea of ‘key−framing’.

The fourth chapter introduces OpenGL’s lighting model and continues with a discussion on materials and their properties. A material is an object property approximating real−life materials. When proper material components are chosen, material like wood, glass, steel, etc. can be constructed. A program is constructed where a user can experiment with the light and material properties in order to familiarise with the concept.

A more elaborate geometrical example is presented in the fifth chapter, as its discussion topic is the improvement of the basic, model of a man. A three−dimensional model of a man is created, using a technique that is able to construct a three−dimensional model from several two−dimensional images.

The sixth chapter introduces texture mapping, a technique that enables the use of images as parts of objects, making OpenGL programs more attractive and ‘real’. The topic of texture mapping is not going to be throughoutly exhausted, as the subject is quite complicated and the applications of texture mapping are inexhaustible. An example program is created that can load a bitmap image, select a part of it, in order to create a texture and then this texture is applied on a rotating cube and then on the improved model of a man itself. The user can interactively set different texture properties like texture filters and so on. A function that is able of saving a texture as a bitmap file is also available.

The seventh, and final chapter contains the conclusions of this report and future possibilities that arise from this project.

Copyright (C) Fotis Chatzinikos 1999

Chapter 2 − Opening a window and drawing simple graphics with OpenGL

As mentioned in the first chapter, OpenGL programs need a graphical window interface in order to work, possibilities include Microsoft's Windows systems, Silicon Graphics systems and X−Windows systems. In this chapter introductory material of OpenGL will be discussed, things like opening and naming a window, clearing the window and drawing simple graphics like a cube.

The first example demonstrates how to open a window by using the *GL Utility Toolkit* named *GLUT*. This library will be used quite often as it contains many functions that without them simple *OpenGL* programs would be quite tedious to write.

The second program goes a bit further and demonstrates how to create and show a cube using *OpenGL*. At this point the cube looks two−dimensional as the projection used is orthographic (projections are described in detail later).

The third example expands the second one, in order to show the difference between flat and smooth shading. A square is drawn either by using flat or smooth shading. The user is able to change back and forth interactively in order to see the difference.

The fourth program draws four cubes. Two of them are displayed with orthographic projection and two with perspective projection. Different order of translation and rotation is also applied in order to demonstrate the different effect.

2.1 Opening a window using OpenGL

The goal of this section is to create an *OpenGL*−based window. There are many ways in which a window can be created and shown under the various windowing systems, but *OpenGL's Utility Toolkit, GLUT*, provides some functions that can create a window in an Operating System independent way. This means that programs created with *GLUT* will operate under different windowing systems without having to change the code manually.

In order to use *OpenGL* and *GLUT* the header file glut.h is needed. This file contains references also to the header files opengl.h and glu.h. These three files are all that is needed at the moment in order to construct some simple *OpenGL* programs. The file windows.h is also need to be included before the inclusion of the *OpenGL* header files, otherwise the compiler will give quite a few errors. In order to make the program portable, the following piece of code can be written (as the file windows.h will not be needed for example with a Silicon Graphics machine).

Example 2.1 Checking for execution platform type

```
#ifdef __FLAT__
#include windows.h
#endif
```

This will check at compile time if the environment is a Microsoft's Win32 environment (Windows 95/98/NT) and if the check is true the file will be included, otherwise the file will not be included (in X−Windows for example).

A window has several properties, like dimensions, name, buffers and so on. These properties must be initialised before the actual window is created and shown. *GLUT* provides several functions for this particular reason. In this example the calls to these functions can be found inside the body of the **main** function. The initialisation of the window is the topic of the next paragraph.

Copyright (C) Fotis Chatzinikos 1999

Before using any *GLUT* functions the *OpenGL Utility Toolkit*, *GLUT* must be initialised. This is done by calling the function **glutInit** inside the main function of the program. After *GLUT* is initialised the display mode of the window must be initialised, too. Calling the function **glutInitDisplayMode** will do the last. This function accepts quite a few arguments as the display mode of a window can be double or single buffered, RGB or indexed colour table, with or without a depth buffer etc.

The next thing to do is to call the function **glutCreateWindow** in order to create the actual window, but prior to that, the two functions **glutInitWindowSize** and **glutInitWindowPosition** must be called. The first one as its name implies is responsible for setting the size of the window and the second one for setting the window's initial position. Both size and position can change later on. The last two functions accept two integer arguments, each specifying pixel dimensions. In the case of **glutInitWindowSize** the arguments are its width and height. In the second case the two arguments are the horizontal and vertical distance from the upper left corner of the monitor, where the window in creation should appear (if possible). The function **glutCreateWindow** accepts a string as its argument. This string will be used as the window's name.

Now that the window is actually created, only a few steps remain before the window is ready and visible.

In the following lines, the code that is responsible for doing all the previous operations can be seen (Example 2.2).

Example 2.2 Code to initialise and create a window

```
int main(int argc, char** argv)
{
   glutInit (&argc, argv) ;
   glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB) ;
   glutInitWindowSize (400, 100) ;
   glutInitWindowPosition (100, 100) ;
   glutCreateWindow ("First Chapter − Opening an OpenGL Window") ;

   init() ;
   glutDisplayFunc (display) ;

   glutMainLoop () ;

   return 0 ;
}
```

In this piece of code the actual calls to the *GL* functions can be seen, in order to create a RGB, double buffered window that has 400 pixels width, 100 pixels height and is named *"First Chapter – Opening an OpenGL Window"*. This window will be positioned (if possible) 100 pixels from the upper left corner of the screen (both horizontally and vertically). Some other functions are visible here that have not been mentioned before.

The function **init** is responsible for any initialisation needed prior to the window construction and/or visualisation. Its structure can be seen here.

Example 2.3 The init function

```
void init(void)
{
   glClearColor(1.0, 1.0, 1.0, 0.0) ;
   glShadeModel(GL_FLAT) ;
}
```

This function contains just two OpenGL calls. The first one, named **glClearColor** is responsible for setting the initial clearing colour. The clearing (background) colour in this occasion is set to white (all colour values,

Copyright (C) Fotis Chatzinikos 1999

Red, Green and Blue are set to one). The fourth value (0.0) is the one called alpha value and is normally used for blending. At this point the alpha value is of no importance.

Next the function **glShadeModel** is called in order to set the shading model. The shading model can be either GL_SMOOTH or GL_FLAT. When the shading model is GL_FLAT only one colour per polygon is used, whereas when the shading model is set to GL_SMOOTH the colour of a polygon is interpolated among the colours of its vertices. An example will demonstrate this particular difference later on.

Back in the **main** function, two more calls follow the call to the function **init**. The first one, named **glutDisplayFunc,** is the first and most important event callback function that will appear in this report. The callback functions are special functions that are registered in order to do some specific operations. Whenever *GLUT* determines that the contents of a window need to be redisplayed, the callback function registered by **glutDisplayFunc** is executed. Therefore all the code that has to do with drawing must be inside the display callback function.

The following code shows the function **display**. **display** is the function registered as the display callback by calling the function **glutDisplayFunc**(**display**).

Example 2.4 Basic display function

```
void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT) ;
   glutSwapBuffers() ;
}
```

Like all the functions that will be used as display callback functions, **display** is of type void. As this program is quite simple and no actual drawing happens in the window, the contents of this function are quite simple, too. They are actually the only “compulsory” function calls that should always appear in any display callback function.

The first one, **glClear** must be called prior to any drawing as it clears the background. It can be omitted if it is desired to draw several times without clearing the background! It accepts one argument that specifies the desired buffer to be cleared. In this program, as no actual drawing happens, this function just clears the background to the colour set previously in the **init** function by the function **glClearColor**.

The function **glutSwapBuffers** does exactly what its name implies. It swaps the back buffer with the front buffer, as when a window is double buffered, the default drawing buffer is the back buffer. Any actual drawing happens in the back buffer and when the drawing is ready the two buffers are swapped in order to achieve smoothness and remove any flickering. If a window had only a single buffer the call **glFlash** would be used instead.

Back in the **main** function the last routine called is **glutMainLoop**. After all setup is done, *GLUT* programs enter this event−processing loop, never to exit until the program is finished.

The results of the program, after compiling, linking and running can be viewed in plate 2.1. Further information on how to make an *OpenGL* project in Borland C/C++, compile and link, can be found in Appendix I.

Copyright (C) Fotis Chatzinikos 1999

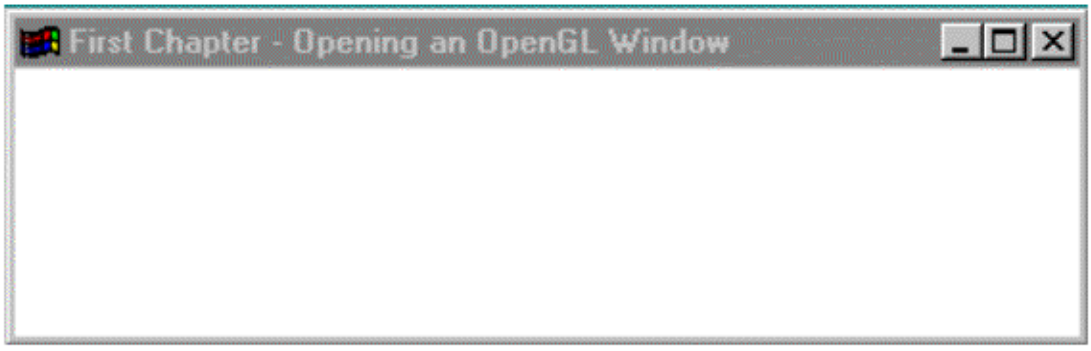


Plate 1.1 A simple *OpenGL* Window

2.2 Creating and showing a cube

Now that it has been demonstrated what must be done to create and show a simple *OpenGL* window, it is the time to go a bit further and create a simple cube.

In this section the previously acquired knowledge is going to be used in order to create a window. When the window is created, it is then quite easy to draw a simple wireframe cube just by calling some OpenGL functions. The steps needed to create such a simple cube will be the topic of this section.

As the following programs (in the next chapters) will be quite complicated, a first attempt will be made in this program to try and create a project that its code will be separated in more than one files (in order to keep the code simple and easy to understand). For the purposes of this example only three files will be needed. The first one will contain the main program and is named *main.c*, the second one is called *model.c* and will contain the functions that will be responsible for drawing any models, in this case a simple function that draws a wireframe cube of constant size. The last file is named *model.h* and it is the header file that will contain any function definitions needed by the main program. These functions will be implemented in the *model.c* file.

This program is mainly the same as the previous one with the only additions being a slight change of the **display** function in order to draw a wireframe cube and the splitting of the project in three different files.

Example 2.5 Display function that draws a wireframe cube

```
void display(void)
{
        glClear(GL_COLOR_BUFFER_BIT) ;
        Draw_Wireframe_Cube() ;
        glutSwapBuffers() ;
}
```

As it can be seen in Example 2.5 the only difference from the **display** function in the previous program (Example 2.4) is the inclusion of the function **Draw_Wireframe_Cube**. This function is responsible for creating a simple wireframe cube of size one. This function is available to the main program by including the header file *model.h*. The structure of this file can be seen in example 2.6.

Example 2.6 Basic header file

```
#ifndef MODEL
#define MODEL

#ifdef __FLAT__
   #include <windows.h>
#endif
```

Copyright (C) Fotis Chatzinikos 1999

```
#include <gl/glut.h>

void Draw_Wireframe_Cube(void) ;

#endif
```

This header file contains the definition of the function **Draw_Wireframe_Cube**. It can be seen that this function is of type void and that it does not accept any parameters. The implementation of this function can be found in the file *model.c* and example 2.7 shows the contents of this file. The statement *#ifndef* is used for conditional compilation and compilation time minimisation. When the compiler tries to compile the particular file, it checks if the file is already defined. If the file is not defined, then it continues compiling, otherwise it does not compile the file. For example if a particular header file is referenced from several implementation files, and the compiler has already compiled the particular header file (and named it using the *#define 'identifier'* statement) there would be no reason to recompile it.

Example 2.7 Basic implementation file

```
#include "model.h"
void Draw_Wireframe_Cube (void)
{
        glColor3f(0.0,0.0,0.0) ;
        glutWireCube(1.0) ;
}
```

It can be seen in this example that an implementation file must include its definition file (in this case *model.h*) and of course the implementation of the functions defined in the header file. When including files, <> are used to direct the compiler to look for the particular file in the systems directory and "" are used to direct the compiler to look in the current directory for the specified file.

In example 2.7, two new *GL* functions are introduced (an *OpenGL* and a *GLUT* function). The *OpenGL* function named **glColor3f** is responsible for setting the current colour. As the working colour mode is RGB (Red−Green−Blue), this function accepts three parameters; one for the red, one for the green and one for the blue value of the colour. The values of these parameters can range from 0.0 to 1.0 (black being zeros for all red, green and blue parameters and white being ones for all three parameters). In this example the colour is set to black (0.0, 0.0, 0.0).

The *GLUT* function named **glutWireCube** is responsible for drawing a wireframe cube that its size is specified by its one, floating point, parameter. In this example the size of the wireframe cube is set to one. So the function **Draw_Wireframe_Cube** just sets the colour to black and draws a wireframe cube.

At this point the program is ready. If it is compiled and run the results will be the ones shown in plate 2.2.

The cube looks like a simple rectangle because the default *OpenGL* projection is orthographic (more on projections in section 2.4), so only the front face of the cube is visible and in such a way that it hides the five remaining faces.

Copyright (C) Fotis Chatzinikos 1999

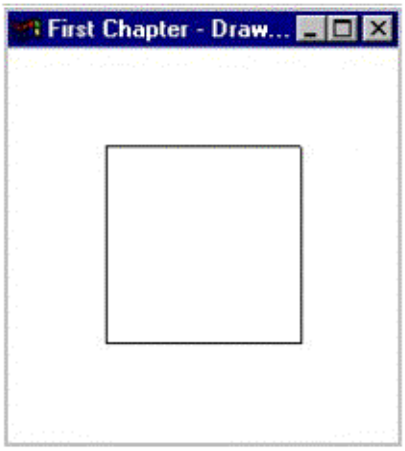


Plate 2.2 Drawing a simple wireframe cube

2.3 Difference between flat and smooth shading

In this example the difference between flat and smooth shading will be demonstrated. In this case a square will be drawn, but without using the GLUT function **glutWireCube** (or **glutSolidCube**). These *GLUT* functions will not be used, as they do not provide any way of setting different colours to different vertices, and in order to demonstrate smooth shading at least two of the vertices of a polygon should be of a different colour. It would be easier to create a rectangle by using the function **glutSolidCube** and then scaling it down in order to make it flat (a flat cube is a square), but as it was just mentioned this can not be done (because of the colours). A custom function will be created that will draw a square with four different colours assigned to each of the four vertices. Example 2.8 demonstrates this function called **Draw_A_Rectangle**.

Example 2.8 Function Draw_A_Rectangle

```
void Draw_A_Rectangle(void)
{
   glBegin(GL_QUADS) ;
   glColor3f(0.0,1.0,0.0) ;
   glVertex2f(0.25,0.25) ;
   glColor3f(1.0,1.0,0.0) ;
   glVertex2f(0.25,0.75) ;
   glColor3f(1.0,0.0,0.0) ;
   glVertex2f(0.75,0.75) ;
   glColor3f(0.0,0.0,1.0) ;
   glVertex2f(0.75,0.25) ;
   glEnd() ;
}
```

A square has four vertices and as seen in example 2.8 only four calls to the function glVertex2f are needed. The function **glVertex2f** specifies two−dimensional vertices. Other graphics systems need an extra vertex in order to ‘close’ a shape; *OpenGL* does not need this extra call as when **glBegin** is called with the parameter GL_QUADS, *OpenGL* automatically connects the first and the fourth vertices. When a shape (a geometric primitive) is constructed in *OpenGL*, it is always bracketed between the commands **glBegin** and **glEnd**.

Between **glBegin** and **glEnd** several different OpenGL commands can be issued. In this example only two different ones are used; **glColor3f** to set the current colour and immediately afterwards **glVertex2f** to specify a vertex of the previously set colour. By passing the particular values in the four **glVertex2f** commands, a rectangle that lies from 0.25, 0.25 to 0.75, 0.75 is created. With its four vertices having the colours green, yellow, red and blue (from left to right).

If the previous program (demonstrated in section 2.2) is slightly changed and instead of using the function

Copyright (C) Fotis Chatzinikos 1999

Draw_Only_Cube uses the function **Draw_A_Rectangle** (inside the display function), the result will be the one shown in Plate 2.3. The square will appear blue and at the upper right corner of the window.

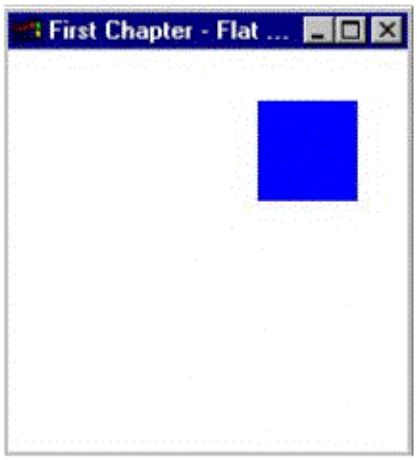


Plate 2.3 Flat shaded rectangle without a reshape function

The square appears blue because the shading mode was set to GL_FLAT (in example 2.3); that means that only one colour per polygon is used. Later on this example the effect of smooth shading will be shown. At the moment, the concentration will be on why the rectangle appears on the upper right corner of the window. This happens because the default projection mapping of *OpenGL* is orthographic and has boundaries from –1 to 1 in all three dimensions. An orthographic projection can be thought as a 3D rectangle. This results in the showing of the rectangle in the upper right corner of the window, as the centre of the window has the co−ordinates 0,0 and the upper right corner the co−ordinates 1,1 (on the X, Y axes). If the rectangle needs to be shown in the centre of the screen, a means of manipulating the projection area has to be found. Introducing the function **glutReshapeFunc** can do this.

This is another *GLUT* callback function, quite similar to the one described before named **glutDisplayFunc**. This function specifies the function that will be called whenever the window is resized or moved. It can also be used to initialise the projection type. Example 2.9 shows the reshape function for the particular program.

Example 2.9 Reshape function that specifies a 2D area (0,0 to 1,1)

```
void reshape(int w, int h)
{
   glViewport ( 0, 0, (GLsizei)w, (GLsizei)h) ;
   glMatrixMode(GL_PROJECTION) ;
   glLoadIdentity() ;
   gluOrtho2D(0.0,1.0,0.0,1.0) ;
   glMatrixMode(GL_MODELVIEW) ;
   glLoadIdentity() ;
}
```

In this function a few new *OpenGL* functions are used. The first one, **glViewport** is responsible for setting the current window's viewport. A viewport specifies the part of the window that all the drawing will take place, with parts out of the viewport normally clipped out. In this case the viewport will be the whole window as the values that are passed to the function **glViewport** specify the viewport to lie from point (0, 0) and for w pixels width and h pixels height. w and h are the width and height of the current window, so the viewport is the whole window.

The next call is to the function **glMatrixMode**. This function is responsible for setting the current matrix mode, as in *OpenGL* more that one mode exists. As it is seen the argument to the first call of **glMatrixMode** is GL_PROJECTION; this means that any matrix manipulations from this point onwards will

Copyright (C) Fotis Chatzinikos 1999

affect the projection matrix. A couple of lines later the same routine is called, but this time the argument is GL_MODELVIEW. This indicates that succeeding transformations now affect the modelview matrix instead of the projection matrix.

In example 2.9 after a call to the routine **glMatrixMode** (in both occasions), a call to the routine **glLoadIdentity** follows. This routine is responsible for clearing the current modifiable matrix from any previous transformations by setting it to the initial identity matrix.

The function that is responsible for setting the projection area of the window follows. As previously noticed, the default *OpenGL* projection is 3D orthographic with boundaries from –1 to 1 in all three dimensions. The routine **gluOrtho2D** is used to transform the projection to two−dimensional (by setting the z boundaries to –1 and 1). The clipping boundaries are specified with four arguments that the routine accepts. In this case the 2D orthographic projection area is set to be from (0, 0), the lower left corner of the window to (1, 1) being the upper right corner of the window.

If the **main** function is slightly modified in order to include a call to the routine **glutReshapeFunc** with the argument being **reshape** (just after the call to **glutDisplayFunc**), the results from the program will be the ones shown in Plate 2.4.a. The problem is that if the window is slightly reshaped (Plate 2.4.b) the rectangle will also be
reshaped (it will stop being a square).

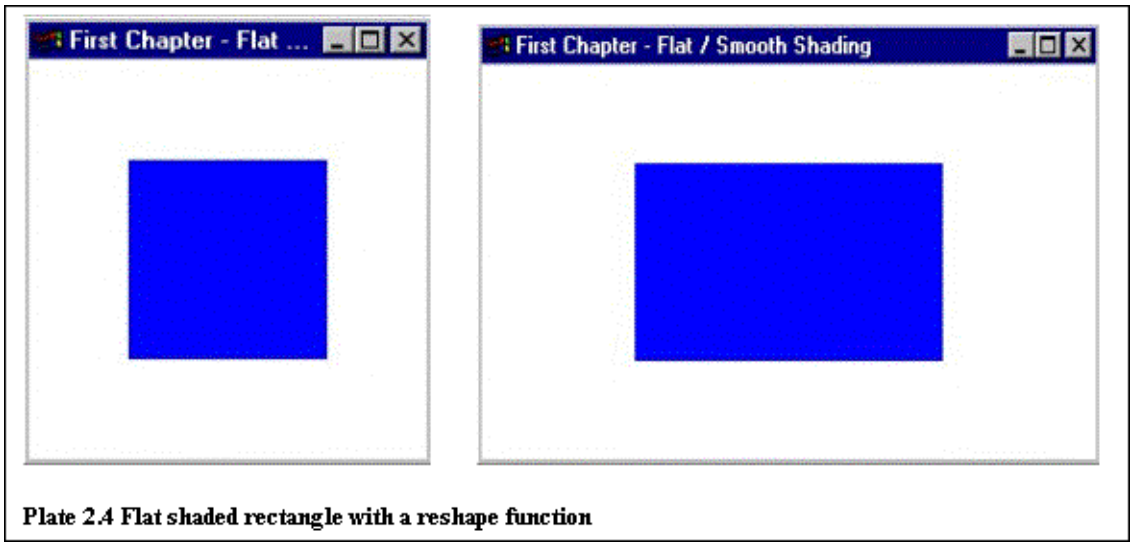


Plate 2.4 Flat shaded rectangle with a reshape function

This can be changed so that the rectangle will always appear as it was initially set. Example 2.10 demonstrates the slight change to the **reshape** function in order to accommodate that.

Example 2.10 Reshape function that ……..

```
void reshape(int w, int h)
{
   if (w >= h)
      glViewport(0, 0, (GLsizei)h, (GLsizei)h) ;
   else
      glViewport(0, 0, (GLsizei)w, (GLsizei)w) ;
   ……
}
```

Copyright (C) Fotis Chatzinikos 1999

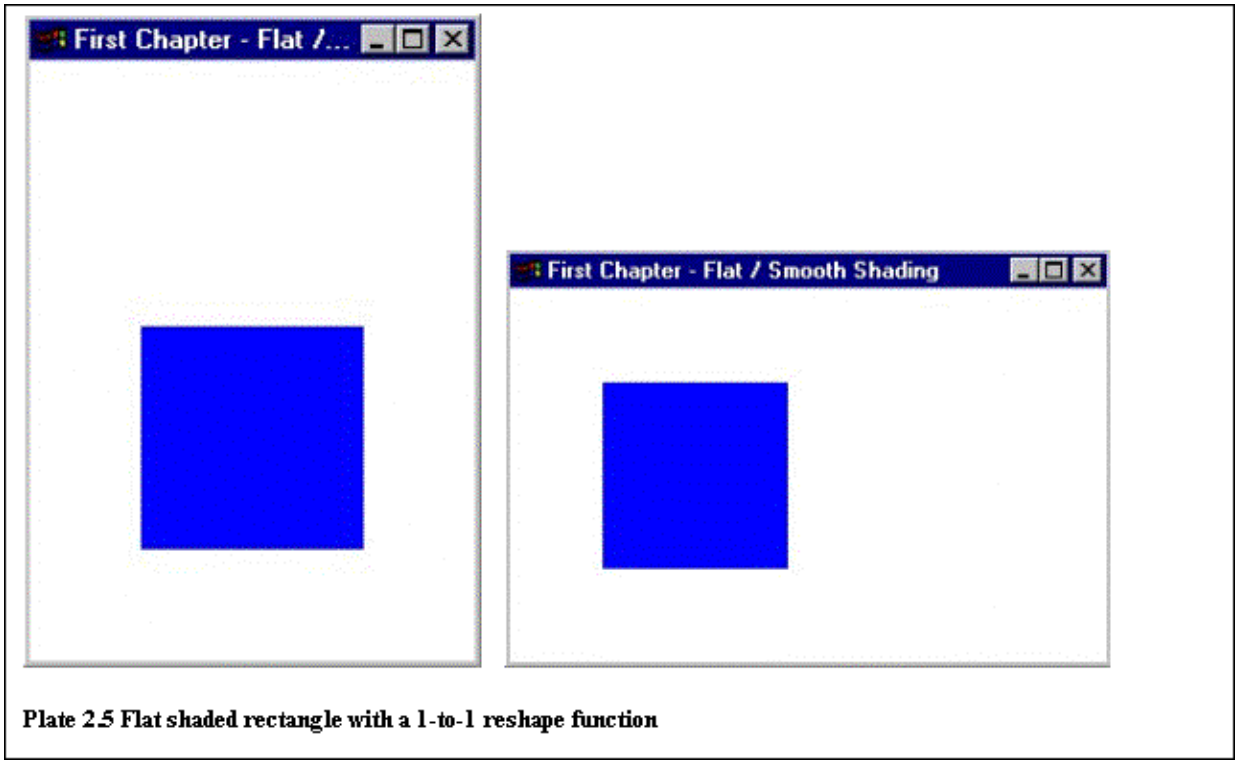


Plate 2.5 Flat shaded rectangle with a 1-to-1 reshape function

This code finds which of the two sides (height or width) is longer and then sets the viewport in such a way that it is always square (either h x h or w x w). When the new program is compiled and run the results are the ones shown in Plate 2.5.

Now that everything about the projection used is explained, it is time to go on and see what happens if the argument passed to **glShadeModel** changes from GL_FLAT to GL_SMOOTH. This time the rectangle will not appear blue but its colour will be calculated by interpolating the colours of its four vertices. Plate 2.6 shows exactly that.

This example also gives an opportunity to introduce keyboard interaction, by introducing the GLUT routine **glutKeyboardFunc**. This function is similar to **glutDisplayFunc** and **glutReshapeFunc**, as it is used to register a keyboard callback routine. Example 2.11 shows the structure of the keyboard function that can change between flat and smooth shading by using the keys ‘f’−‘F’ (for flat shading) and ‘s’−‘S’ (for smooth shading).

Example 2.11 The keyboard function

```
void keyboard (unsigned char key, int x, int y)
{
   switch (key)
   {

      case 's' :
      case 'S' :
         glShadeModel(GL_SMOOTH) ;
         break ;
      case 'f' :
      case 'F' :
         glShadeModel(GL_FLAT) ;
         break ;
      default :
         break ;
   }
   glutPostRedisplay() ;
```

Copyright (C) Fotis Chatzinikos 1999

```
}
```

A new *GLUT* routine is also introduced in this function, **glutPostRedisplay**. This routine marks the current window as needing to be redrawn. At the next opportunity, the callback function registered by **glutDisplayFunc** will be called to redraw the window. If the routine **glutPostRedisplay** was not included at this point, the user could press keys without receiving any feedback at all. This happens because *OpenGL* does not know that the contents of the window change whenever the user presses a key, as it is quite normal that the keyboard will be used for reasons other than changing the contents of the window.

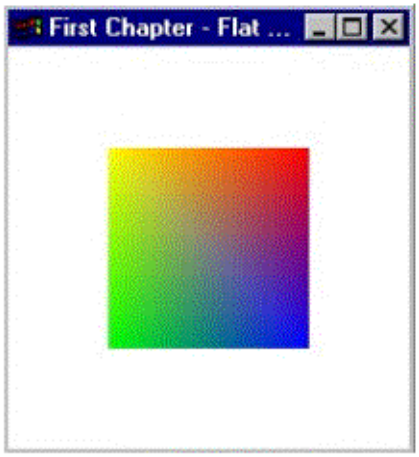


Plate 2.6 Smooth shaded rectangle

2.4 Modelling and projection transformations

This section will try to explain the main modelling transformations, translate, rotate and scale and the basic projection transformations, orthographic and perspective projection. In order to achieve that, four cubes will be drawn in the four quadrants of the window. The upper two cubes will be shown using orthographic projection; whereas the lower two by using perspective projection. Different order of modelling transformations will be used to demonstrate this particular difference.

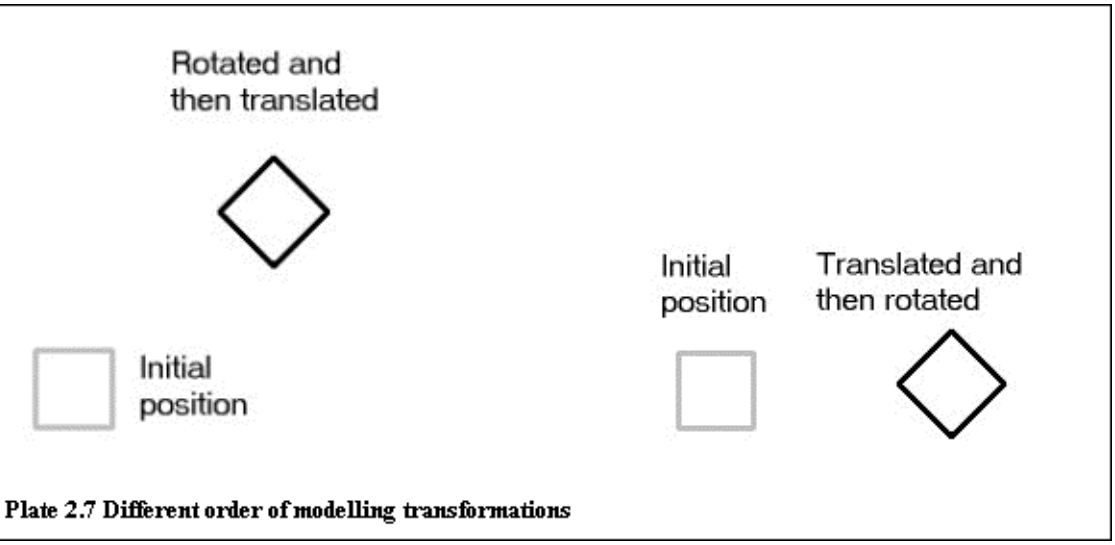


Plate 2.7 Different order of modelling transformations

Modelling transformations are used to position and orient the models. Three basic transformations are available in *OpenGL* and these are translation, rotation and scaling. The order of these transformations is not irrelevant to the final transformation. For example, if an object is firstly translated and then rotated, it will have a different position and orientation from the same object that has been firstly rotated and then translated. This particular difference will be demonstrated later in the example program, but for now this difference can be viewed in Plate 2.7. In the picture on the left, a cube is firstly rotated 45 degrees and then translated x

Copyright (C) Fotis Chatzinikos 1999

units. In the picture on the right, the same cube is firstly translated the same x units and then rotated by 45 degrees. After having introduced the concept of modelling transformations, the concept of projection transformations will follow.

Specifying the projection transformation is like choosing a lens for a camera. This transformation can be thought as choosing the field of view (FOV) or viewing volume and therefore what objects are inside and how they look. Back to the camera example, it is like choosing among different lenses. With a wide−angle lens, a bigger area is included in the photo than with a normal or telephoto lens, but with a telephoto lens more detail appears in the photograph, as objects look nearer. In computer graphics zooming in and out of an object is much easier than changing lenses in a camera, as the only thing to be done is to choose a smaller field of view.

In addition to the field of view considerations, the projection transformation determines how objects project (look) on the screen. Two types of projections are provided with *OpenGL*, perspective and orthographic projection.

The first type of projection, perspective projection, matches how objects appear in real life. Perspective makes objects that are further away appear smaller; for example it makes the two sides of a road appear to converge in the distance. For realistic looking pictures, this type of projection should be used.

Orthographic projection on the other hand, maps objects directly on the screen, without altering their relative size. This type of projection is useful for many CAD−based applications like circuit design or architectural planning, as the user needs to see actual measurements of objects, rather than how these objects look. Architects can use perspective projection in order to visualise how a particular building or room would look from a particular viewpoint, and then switch to orthographic projection in order to print out the blueprint plans.

Now that the theory of modelling and projection transformations is partially explained, the actual example that demonstrates these transformations can follow.

This project is split into four different implementation files and their corresponding header files (except the main program that does not have a special header file). The file main.c contains the main program, the file model.c contains the modelling routines, the file transformations.c contains the modelling transformation routines and the file keyboard.c contains the keyboard interaction routines.

Starting by the main program it can be noted that there is no need for a **reshape** function as all the operations that would normally occur in the body of such a function are carried out in the body of the **display** function. A two−dimensional array of type float is used in order to communicate the rotation, translation and scaling values in the four different files. Actually this array is needed in the transformations file, so it was declared in the header of this file as **#extern**.

In the body of the **init** function (example 2.11) this array is partially initialised. The remaining array elements are not initialised here, as there is no need to do so. The constants SCALE and ROTATE are declared in the file model.h as 2 and 1 (TRANS is also declared there as 0). The scaling elements are initialised to 1 (no scaling) and the rotate elements of the array are initialised to zero, except the first one that is initialised to 1 (initially rotate only the x−axis).

Example 2.12 Init function

```
void init(void)
{
   glClearColor(1.0, 1.0, 1.0, 0.0) ;
   glShadeModel(GL_FLAT) ;
   tran[SCALE][0] = 1.0 ;
   tran[SCALE][1] = 1.0 ;
```

Copyright (C) Fotis Chatzinikos 1999

```
   tran[SCALE][2] = 1.0 ;
   tran[ROTATE][0] = 1.0 ;
   tran[ROTATE][1] = 0.0 ;
   tran[ROTATE][2] = 0.0 ;
}
```

The **main** function of the program is the same as the one in the previous program with the only difference being that there is no **glutReshapeFunc**, as a **reshape** function is not needed. A function that is quite different (from the previous program) is the **display** function, as this is the place where the actual drawing and placing of the four cubes that constitute this example happens. Example 2.12 contains the code of this function.

Example 2.13 display function that positions and draws four cubes

```
void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT) ;
   glShadeModel(GL_FLAT) ;
   glMatrixMode(GL_PROJECTION) ;
   glLoadIdentity() ;
   glViewport(0,125,125,125) ;
   glOrtho(−2.0,2.0,−2.0,2.0,2.0,−2.0) ;

   glMatrixMode(GL_MODELVIEW) ;
   Draw_Cube_Transl_Rot() ;

   glMatrixMode(GL_PROJECTION) ;
   glLoadIdentity() ;
   glViewport(0,0,100,100) ;
   gluPerspective(60.0, 1.0,1.0,20.0) ;
   glTranslatef(0.0,0.0,−4.0) ;

   glMatrixMode(GL_MODELVIEW) ;
   Draw_Cube_Transl_Rot() ;

   glMatrixMode(GL_PROJECTION) ;
   glLoadIdentity() ;
   glViewport(100,100,100,100) ;
   glOrtho(−2.0,2.0,−2.0,2.0,2.0,−2.0) ;

   glMatrixMode(GL_MODELVIEW) ;
   Draw_Cube_Rot_Transl()

   glMatrixMode(GL_PROJECTION) ;
   glLoadIdentity() ;
   glViewport(100,0,100,100) ;
   gluPerspective(60.0, 1.0,1.0,20.0) ;
   glTranslatef(0.0,0.0,−4.0) ;

   glMatrixMode(GL_MODELVIEW)
   Draw_Cube_Rot_Transl() ;
   glutSwapBuffers() ;
}
```

This function contains four very similar parts. All of them start by setting the projection matrix as the current. A call to **glLoadIdentity** follows in order to initialise the matrix and then a call to **glViewport** is done in order to set the viewport. The arguments to the **glViewport** routine are such that will divide the window (of 200 by 200 pixels) in four equal quadrants. After this is done a call to either **glOrtho** or **gluPerspective** is done in order to set the projection of the current quadrant. Plate 2.8 shows the co−ordinates of the four quadrants and their projection.

Copyright (C) Fotis Chatzinikos 1999

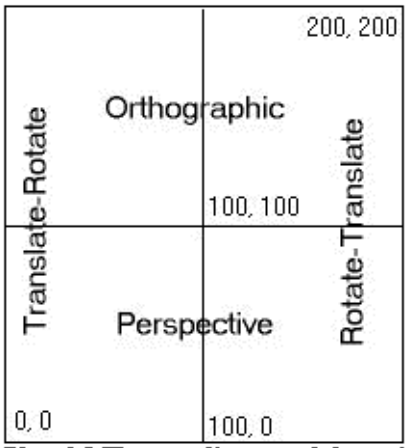


Plate 2.8 The co-ordinates and the projection used in the four quadrants

The call to **glOrtho** and its arguments are quite easy to understand, as it is similar to the routine **gluOrtho2D** that has already been explained (the only difference is that **glOrtho** sets also the z boundaries). The routine **gluPerspective** needs further explanations. This routine creates a symmetric perspective−view frustum (a three−dimensional area). The first argument to the routine is the angle of the field of view (in the x−z plane). The second argument is the aspect ratio of the frustum (normally its width divided by its height) and the remaining two arguments are the near and far values of the frustum. These values the distances between the viewpoint and the clipping planes along the negative z−axis, and they should always be positive. As the frustum will always lie on the negative z−axis the object shown most times will be needed to be translated some value x before shown to the screen. This is why a call to **glTranslatef**(0.0, 0.0, −4.0) is necessary after a call to **gluPerspective**. The correct amount of translation and frustum creation is not something that can be shown exactly but comes naturally after becoming ‘comfortable’ with the concept.

When this is done a call to **glMatrixMode**(GL_MODELVIEW) resets the current matrix to the modelview matrix in order to draw the four cubes. After the matrix is set to modelview a call to either **Draw_Cube_Transl_Rot** (for the two quadrants on the left−hand side) or **Draw_Cube_Rot_Transl** (for the two quadrants on the right−hand side) is done. This is done because these two functions that both draw a cube, contain calls to the routines **glTranslate** and **glRotate**. As explained before the order of these routines is important. To demonstrate this, the routine **Draw_Cube_Transl_Rot** draws the cube after applying the transformations in the order of translate and then rotate, whereas the second one draws the cube after applying these two transformations in the opposite order, in order to visualise the difference.

At this point the main program is ready. The two custom functions that were used, **Draw_Cube_Transl_Rot** and **Draw_Cube_Rot_Trans** are available in the program by including the header file keyboard.h as this file contains references also to the files model.h and transformations.h .

This section will be continued by examining these two custom functions whose implementation is in the file transformations.c. Example 2.13 contains the code of the first one, **Draw_Cube_Transl_Rot**. The code of the second one is the same with the only difference being that the order of the routines **glTranslate** and **glRotate** is the opposite.

Example 2.14 display function that positions and draws four cubes

```
void Draw_Cube_Transl_Rot (void)
{
   glPushMatrix() ;
      glTranslatef (tran[TRANS][0],tran[TRANS][1],tran[TRANS][2]) ;
      glRotatef (tran[ROTATE][3], tran[ROTATE][0], tran[ROTATE][1],tran[ROTATE][2] ) ;
      glScalef(tran[SCALE][0],tran[SCALE][1],tran[SCALE][2])
```

Copyright (C) Fotis Chatzinikos 1999

```
      Draw_Black_Cube() ;
   glPopMatrix() ;
}
```

It is noticeable that two statements named **glPushMatrix** and **glPopMatrix** wrap the body of the function. The routine **glPushMatrix** is responsible for saving the current matrix in the matrix stack. The matrix stack is a stack that is used to save and restore transformation matrices during the execution of an *OpenGL* program. Actually there are three different matrix stacks; one for modelling transformations, one for projection transformations and one for texture transformations. This means that a particular matrix can be saved in the stack by using **glPushMatrix**, transformations that modify the matrix can be done and when the previous to the modifications matrix is needed again, it can be loaded or 'popped' from the stack by using the command **glPopMatrix**. These two routines are particularly helpful when building hierarchical models and they are going to be explained in detail when time comes.

Between these two calls four more routines are called, a **glTranslate**, a **glRotate** and a **glScale**. **glTranslate** and **glScale** accept three arguments being the X, Y, and Z values the object should be translated or scaled. In the case of **glScale** an argument of 1 means no scaling, an argument of 0.5 means reduce the scale in half and an argument of 2 means double the scale. **glRotate** accepts four arguments with the first one being the amount of degrees the object should be scaled and the other three varying from 0 to 1. If 0 is passed the particular axis is not rotated, whereas if 1 is passed the particular axis is fully rotated. The arguments of these routines (elements of the *tran* array) are modified externally by other functions that will be explained shortly. After all modelling transformations are done, a call to **Draw_Black_Cube** is made in order to draw a transformed (due to the modelling transformations) black cube. The routine **Draw_Black_Cube** can be found in the file model.c and it is a very simple routine as it just sets the current colour to black and uses the routine **glutWireCube**(1.0) to draw a cube of size 1.0.

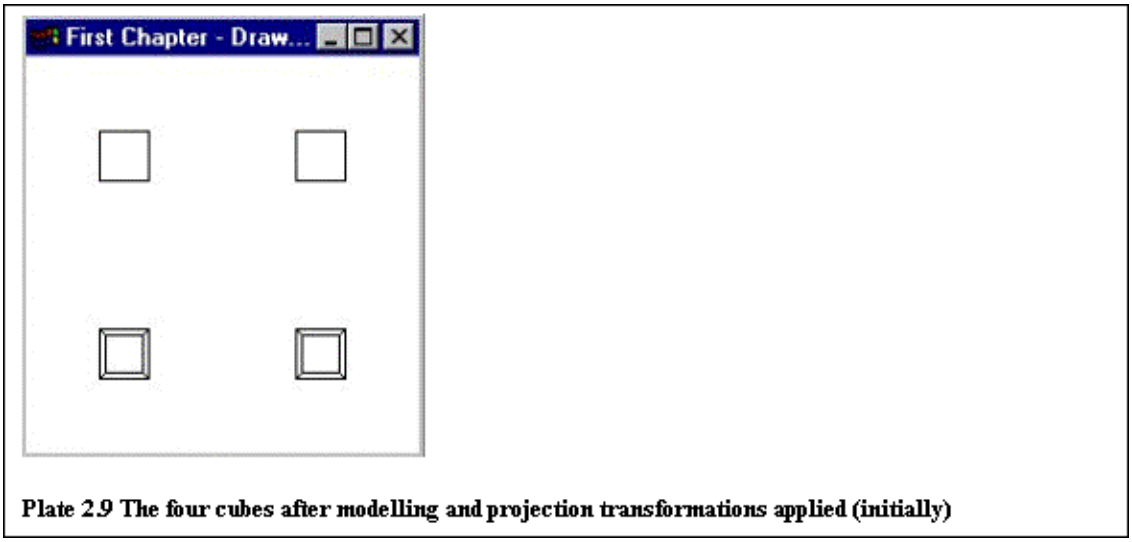


Plate 2.9 The four cubes after modelling and projection transformations applied (initially)

At this point if the program is compiled and run the results will be the ones shown in Plate 2.9. The difference between perspective and orthographic projection is clearly visible, but the difference in the order of the application of the modelling transformations is not yet visible, as none of them have been applied yet.

The file keyboard.c contains the **keyboard** function that is needed in order to interact with the program through the keyboard. This function is shown in example 2.15.

Example 2.15 display function that positions and draws four cubes

```
void keyboard (unsigned char key, int x, int y)
{
   switch(key)
```

Copyright (C) Fotis Chatzinikos 1999

```
   {
      case 'a' :
         Rotate_Cube() ;
         glutPostRedisplay() ;
         break ;
      case 's' :
         Rotate_Cube3D() ;
         glutPostRedisplay() ;
         break ;
      case 'd' :
         Move_Cube() ;
         glutPostRedisplay() ;
         break ;
      case 'f' :
         Scale_Cube() ;
         glutPostRedisplay() ;
         break ;
      case '1' :
         glutIdleFunc(Small_Anim) ;
         break ;
      default :
         glutIdleFunc(NULL) ;
         break ;
   }
}
```

After examining example 2.14 it is clear that by pressing the keys a, s, d, f and 1 five different things will happen. These five functions will be explained shortly. Something new to this function is the call to the function **glutIdleFunc**. This function can be called in order to do something when the program is idle, for example a small animation. The effect of this function becomes inactive when a NULL is passed to it. As it is seen in the example if the user presses any other than the specified keys, a call to **glutIdleFunc** is done with a NULL argument in order to stop any previously issued **glutIdleFunc** routine.

Back in Example 2.10, a single call to **glutPostRedisplay** was issued just after the end of the switch statement. This approach is not followed here because it is not needed to issue a **glutPostRedisplay** command every time a key is pressed. However, in the case of the key ‘1’ the call to **glutPostRedisplay** must be inside the function **Small_Anim**.

These five functions are part of the file transformations.c. These functions will be explained here but their code will not appear as they are quite simple to understand. The first one, **Rotate_Cube3D** sets the first three rotation elements of the *tran* array to 1, and the fourth one is incremented by 1 each time the function is called. This is done because of the structure of the **glRotate** function. As these array elements are used in a call to **glRotate** of the same form used in example 2.13 and the function is required to rotate all three axes, the last three arguments to the **glRotate** function should be 1 (tran[0] to tran[2]) and the first argument should contain the degrees of the required rotation (tran[3]).

The second function, **Rotate_Cube** gradually rotates first the x−axis for 85 degrees, then the y−axis for the same degrees, then the z−axis for the same amount of degrees and finally continuously rotates all three axes.

The third function, **Move_Cube** does some translation transformations that result in the movement of the cube in a square pattern (rightwardsà upwards à leftwards à downwards à rightwards à and so on).

The fourth function, **Scale_Cube** scales the cube up and down between twice its original size and a quarter of it.

The remaining function, **Small_Anim** uses the previously defined functions **Rotate_Cube3D** and **Move_Cube** in order to demonstrate the difference in the application of the modelling transformations. The results of this function will be different when using the functions **Draw_Cube_Trans_Rot** and **Draw_Cube_Rot_Trans** to visualise the cube as it simultaneously translates and rotates the axes.

Copyright (C) Fotis Chatzinikos 1999

If the program is compiled and run, and the user presses the key ‘1’ to invoke the function **Small_Anim**, the results will be the ones shown in Plate 2.10. The difference between both the order of applying the modelling transformations and the projection transformations is now clearly visible.

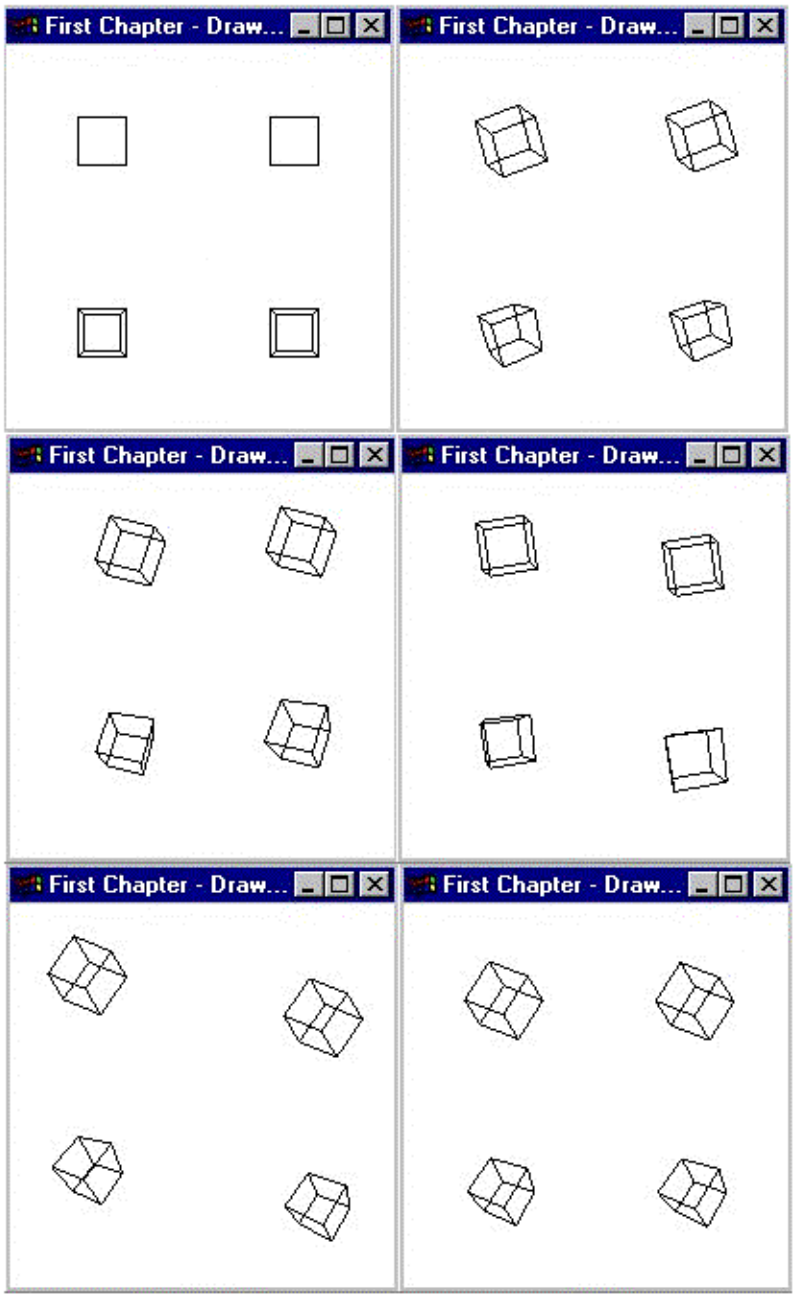


Plate 2.10 Snapshots from the modelling & projection transformations program

Copyright (C) Fotis Chatzinikos 1999

Chapter 3 – Creating a hierarchical, 3D, wire frame model

Now that the basic *OpenGL* and *GLUT* structure and routines has been explained, it is time to go on and try to put this new knowledge in work. The goal of this chapter is the creation of a hierarchical, wire frame model of a man.

Previously acquired knowledge, like modeling and projection transformations will be used in order to create and animate this model. This chapter is divided into two sections.

In the first section, in order to discuss and explain hierarchical models and their creation, without having to cope with an overcomplicated example, instead of trying to create the whole model of the man, only its lower part will be constructed including its base, legs and feet.

This incomplete model will then be animated. A 'walking' function will be created for this reason. When this example will be finished and enough knowledge and experience will be accumulated, the second section will follow naturally.

In the second section, the incomplete model of the man will be completed and by the end of the section a complete, three dimensional, wire−frame model of man (based on rectangles and spheres) will be ready. The 'walking' function from the previous section will be slightly modified in order to accommodate the whole body.

3.1 Building a basic hierarchical model

This section will start by describing not the code of the program but the structure of and the concepts behind hierarchical models.

Imaging that it was required to build a car for some simulation reasons. For the sake of this example this car is composed of the car's body, four wheels and six windows. There is a front window, a back window and two windows on each side of the car. The side windows are symmetrical and each wheel has five bolts (Plate 3.1).

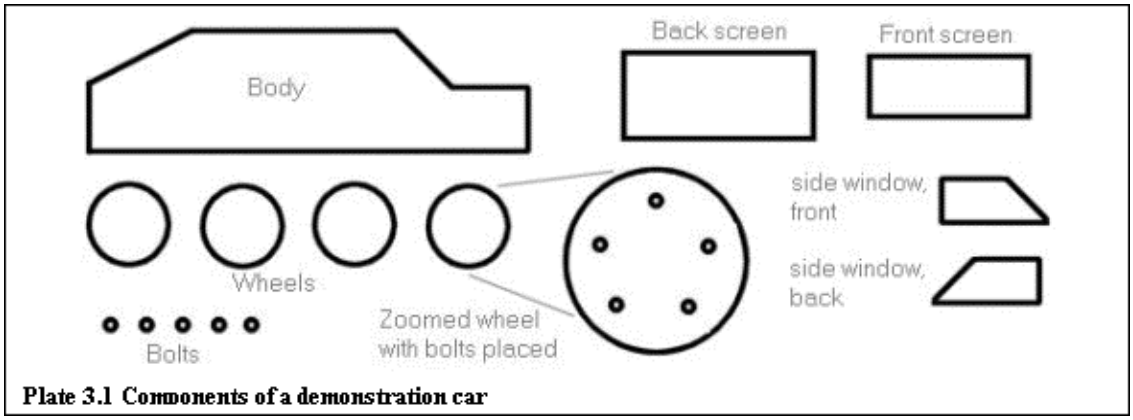


Plate 3.1 Components of a demonstration car

For the purposes of this example only the right (visible in the picture) side components will be used. That is, two wheels, ten bolts and two side windows.

In order to avoid repetition it would be also be desirable to build the car in a hierarchical way; meaning that when the five bolts are correctly placed on the wheel, they should move in accordance with the wheel without any exterior help. Also it would be desirable that when the body of the car moves, its parts would not stay behind but follow its movement.

Copyright (C) Fotis Chatzinikos 1999

In order to achieve this a hierarchy has to be built with the car's body being the topmost item in it and then following it, the windows and the wheels, with the bolts being subordinate to the wheels. This hierarchy has the result of when the car's body is moving, the windows and wheels will move in accordance with it. Further on when the wheels rotate the bolts will rotate also.

OpenGL provides the means to build hierarchical models through the functions **glPushMatrix** and **glPopMatrix**. If it assumed that functions are available that each one of them draws a part of the car i.e. bolt, wheel, window and body then Plate 3.2 demonstrates the needed hierarchy and example 3.1 the appropriate pseudo−code to achieve it.

Example 3.1 Pseudo−code that demonstrates the car's hierarchy

```
function draw_car
{
 glPushMatrix
 draw_body_of_car
 glPushMatrix
 go_to_side_window_front_position
 draw_side_window
 go_to_side_window_back_position
 inverse_axes
 draw_side_window

               inverse_axes
 go_to_front_wheel_position
 draw_wheel_and_bolts
 go_to_back_wheel_position
 draw_wheel_and_bolts
 glPopMatrix
      glPopMatrix
}
function draw_wheel_and_bolts
{
 glPushMatrix
 draw_wheel
 glPushMatrix
 for counter = 1 up to 5 do
 {
 go_to_bolt_position
 draw_bolt
 }
 glPopMatrix
 glPopMatrix
}
```

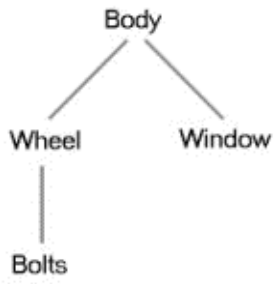


Plate 3.2 Components of a demonstration car

In example 3.1 the function **go_to*** is used to translate to the needed point every time. The function **inverse_axes** is used to inverse the x−axis in order to use the **draw_window** function to draw the side back

Copyright (C) Fotis Chatzinikos 1999

window (as it is the mirror of the side front window). If now a **glTranslate** routine is issued just before the function **draw_car**, the whole car will be moved including the windows and wheels and if furthermore a relation exists that when the car moves the wheels rotate, the bolts will also rotate in accordance with wheels.

Furthermore, the commands **glPushMatrix** and **glPopMatrix** can be used to save time when positioning parts of a scene. For example lets say that the body of the car has a length of 100 units and that the co−ordinates system is positioned at the center of the car, so the car co−ordinates lie from –50 to 50. Lets also assume that the wheels have to be positioned both at ten points before the boundaries of the car. This can be done in two ways.

Without using the matrix stack, a **glTranslate**(40, 0, 0) should be issued in order to move the center of the coordinates forty units on the x−axis, then the wheel would be drawn by calling **draw_wheel_and_bolts** and then the center of the co−ordinates should be moved eighty units back in order to position the second wheel, by calling **glTranslate**(−80, 0, 0).

If the matrix stack is used, and the appropriate commands **glPushMatrix** and **glPopMatrix**, the same can be done in the following, more robust way. A call to **glPushMatrix** can be done in order to save the current matrix and then a call to **glTranslate**(40, 0, 0) and a call to **draw_wheel_and_bolts** can be done in order to draw the first wheel in the correct position. Then a call to **glPopMatrix** can be done in order to retrieve the prior to the translation matrix and then a call to **glTranslate**(−40, 0, 0) and a call to **draw_wheel_and_bolts** can be done in order to position and draw the second wheel.

Now that some understanding of hierarchical models and the matrix stack has been acquired, the actual design of the basic model can start. Plate 3.3 shows the parts of this first basic model and their hierarchical relation.

As it is seen in Plate 3.3, in this occasion the top most item in the hierarchy is the base of the body, followed by the upper leg joint, the upper leg, the lower leg joint, the lower leg, the foot joint and finally the foot. Joints are depicted as spheres and the other parts of the body as rectangles.

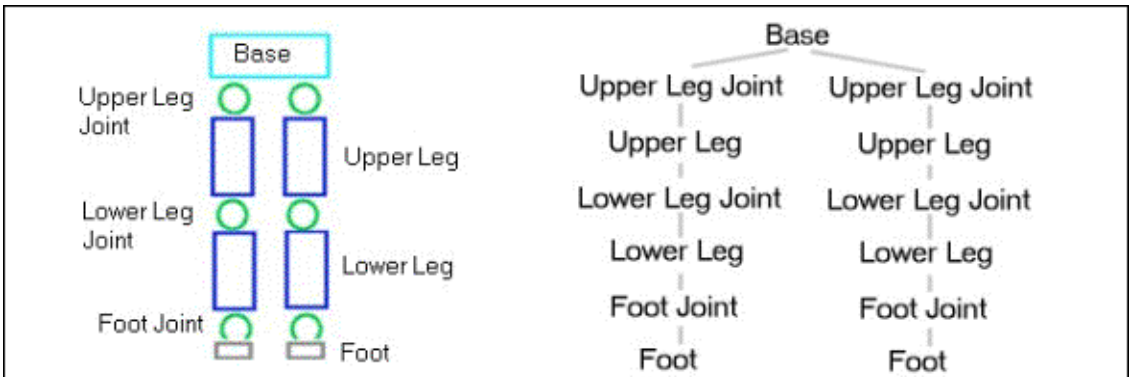


Plate 3.3 Basic model components and their hierarchy

Copyright (C) Fotis Chatzinikos 1999

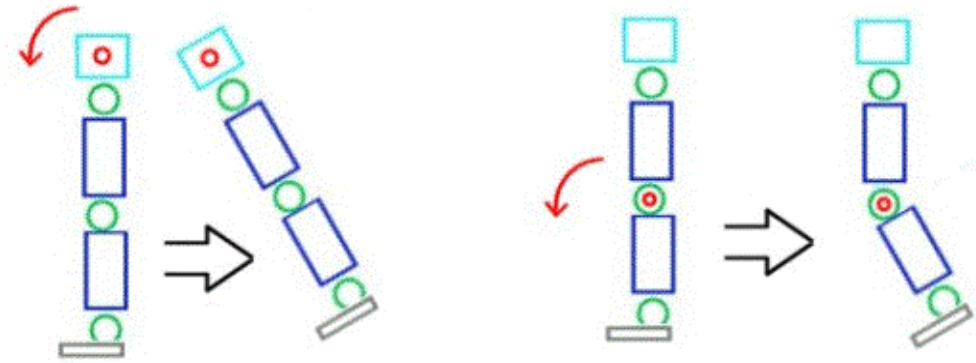


Plate 3.4 Applying rotation in different parts of the basic model

So now if a rotation is applied to the base, all other parts are going to be rotated whereas if a rotation is applied to the lower leg joint, only the joint and the parts lower from it will rotate (Plate 3.4).

Now is the appropriate time to go on and start constructing the code that will draw and latter on animate this model.

The project at this point is split into three files. As always the main program will residue in the file called main.c. Another file will be used called model.c that will contain all the functions that will be needed in order to position, draw and animate this model. The file model.h contains the function definitions of the file model.c.

In order to construct this basic model, some relative metrics have to be calculated that will approximate a human body. As the point of this example was not accuracy but a demonstration of hierarchical model creation, the relative heights and widths of the body parts were based on a not so accurate hand−drawn sketch of a man. Accurate modeling will be the subject of a latter chapter. Example 3.2 shows part of the file model.h, were the relative metrics of the body can be found.

Example 3.2 Size definitions of the models parts

```
#define FOOT_JOINT_SIZE HEAD_JOINT_SIZE
#define FOOT_HEIGHT FOOT_JOINT_SIZE * 2.0
#define FOOT_WIDTH LO_LEG_WIDTH
#define FOOT FOOT_WIDTH * 2.0
#define UP_ARM_HEIGHT TORSO_HEIGHT * 0.625
#define UP_ARM_WIDTH TORSO_WIDTH/4.0
#define UP_ARM_JOINT_SIZE HEAD_JOINT_SIZE * 2.0
#define LO_ARM_HEIGHT TORSO_HEIGHT * 0.5
#define LO_ARM_WIDTH UP_ARM_WIDTH
#define LO_ARM_JOINT_SIZE UP_ARM_JOINT_SIZE * 0.75
#define HAND_HEIGHT LO_ARM_HEIGHT / 2.0
#define HAND_WIDTH LO_ARM_WIDTH
#define HAND LO_ARM_WIDTH / 2.0
#define TORSO_WIDTH TORSO_HEIGHT * 0.75
#define TORSO_HEIGHT 0.8
#define TORSO TORSO_WIDTH / 3.0
#define HEAD_WIDTH HEAD_HEIGHT * 0.93
#define HEAD_HEIGHT  TORSO_HEIGHT * 0.375
#define HEAD_JOINT_SIZE HEAD_HEIGHT/6
#define BASE_WIDTH TORSO_WIDTH
#define BASE_HEIGHT TORSO_HEIGHT / 4.0
#define UP_LEG_HEIGHT LO_ARM_HEIGHT
#define UP_LEG_JOINT_SIZE UP_ARM_JOINT_SIZE
#define UP_LEG_WIDTH UP_LEG_JOINT_SIZE * 2.0
#define LO_LEG_HEIGHT UP_LEG_HEIGHT
#define LO_LEG_WIDTH UP_LEG_WIDTH
```

Copyright (C) Fotis Chatzinikos 1999

```
#define LO_LEG_JOINT_SIZE UP_LEG_JOINT_SIZE
#define LEG_HEIGHT   UP_LEG_HEIGHT + LO_LEG_HEIGHT + FOOT_HEIGHT + 2* (FOOT_JOINT_SIZE
                          + UP_LEG_JOINT_SIZE + LO_LEG_JOINT_SIZE)
```

As it seen in the example only the torso height is defined and all the other parts of the body are related to this height, for example the torso width is three quarters of the torso height etc. This was done having in mind the case that the model is needed to change dimensions, only the torso height has to be changed, as this change will affect all the other parts of the body. In this first section of the chapter not all of the previously defined parts will be needed, but nevertheless they were defined, as they will be needed in the next section, when a full body will be build.

Now that the size of the parts of the body is defined, it is the time to start building the body. The model at this point is constructed from three main parts, the ‘base’ (lower torso) and the two legs. Example 3.3 shows the code that creates the base.

Example 3.3 The function that draws the base of the basic model

```
void Draw_Base(int frame)
{
 glPushMatrix() ;
 glScalef(BASE_WIDTH, BASE_HEIGHT, TORSO) ;
 glColor3f(0.0,1.0,1.0) ;

 if (frame == WIRE)
 glutWireCube(1.0) ;
 else
 glutSolidCube(1.0) ;
 glPopMatrix() ;
}
```

The function **Draw_Base** accepts one argument. This argument will be used to draw either a wireframe base (by passing the value WIRE) or a solid base (by passing the value SOLID). At this point a solid base will be of no use (as the model will be wireframe) but the same function will be used later, when dealing with light, to construct a solid base.

The body of the function starts by calling the function **glPushMatrix** in order to save the current matrix before applying any modifications to it. A call to **glScale** follows with the values BASE_WIDTH, BASE_HEIGHT and TORSO. The result of this call is the scaling of the axes to these new values (from left to right, the axes x, y and z). Now when **glutWireCube**(1.0) is called (or **glutSolidCube**(1.0)) the result will not be a cube but a rectangle approximating the ‘base’ (as seen in Plates 3.3 and 3.4).

This function is quite easy to understand, as there is no hierarchy of objects involved or any rotations. The function **Draw_Leg** is slightly more complicated as it contains three parts, the upper leg, the lower leg and the foot. These three parts are constructed by three different functions. These three functions are similar to each other and example 3.4 shows the function that draws the upper leg, named **Draw_Upper_Leg**.

Example 3.4 The function that draws the upper leg of the basic model

```
void Draw_Upper_Leg(int frame)
{
 glPushMatrix() ;
 glScalef(UP_LEG_JOINT_SIZE, UP_LEG_JOINT_SIZE, UP_LEG_JOINT_SIZE) ;
 glColor3f(0.0,1.0,0.0) ;
 if (frame == WIRE)
 glutWireSphere(1.0,8,8) ;
 else
 glutSolidSphere(1.0,8,8) ;
 glPopMatrix() ;
 glTranslatef(0.0,− UP_LEG_HEIGHT * 0.75, 0.0) ;
```

Copyright (C) Fotis Chatzinikos 1999

```
 glPushMatrix() ;
 glScalef(UP_LEG_WIDTH,UP_LEG_HEIGHT,UP_LEG_WIDTH) ;
 glColor3f(0.0,0.0,1.0) ;
 if (frame == WIRE)
 glutWireCube(1.0) ;
 else
 glutSolidCube(1.0) ;
 glPopMatrix() ;
}
```

In the body of the function, the starting routine **glPushMatrix** is used to save the current matrix prior to the scaling. After the matrix is saved the function **glScale** is used to scale the axes in the appropriate dimensions for the drawing of the upper leg joint. When this is done, the current colour is set to green and the joint is drawn (either as wireframe or solid, depending on the value passed to the function) and then the matrix is restored by calling the function **glPopMatrix**. This has the effect of restoring the axes to their initial one−to−one relation. Following this a call to **glTranslate** is issued in order to move the centre of the axes in the new positioned required to draw the upper leg. At this point the just explained technique is repeated in order to save the matrix, scale the axes, choose the colour (blue this time) and finally draw the upper leg. At this point the function **Draw_Upper_Leg** (the function that draws the upper leg and the upper leg joint) is ready. The functions **Draw_Lower_Leg** and **Draw_Foot** are similar to this one, so they will not be explained explicitly.

Now is the time to take a look at the function **Draw_Leg**. This function combines the previously mentioned functions in order to build the whole leg, including the rotation routines, routines that will be needed for the animation of the model. Example 3.5 contains the code of this function.

Example 3.5 The function that creates the whole leg of the basic model

```
void Draw_Leg(int side, int frame)
{
 glPushMatrix() ;
 glRotatef(walking_angles[side][3],1.0,0.0,0.0) ;
 Draw_Upper_Leg(frame) ;
 glTranslatef(0.0,− UP_LEG_HEIGHT * 0.75,0.0) ;
 glRotatef(walking_angles[side][4],1.0,0.0,0.0) ;
 Draw_Lower_Leg(frame) ;
 glTranslatef(0.0,− LO_LEG_HEIGHT * 0.625, 0.0) ;
 glRotatef(walking_angles[side][5],1.0,0.0,0.0) ;
 Draw_Foot(frame) ;
 glPopMatrix() ;
}
```

As before, the function's body starts by saving the current matrix. Next is a call to **glRotate**. The values passed to this routine show that the object that is drawn after this function is called, will be rotated only on the x−axis (as the second parameter is 1.0 and the third and fourth are 0.0). The first parameter, is the amount of degrees the x−axis should be rotated. This value is contained in the array *walking_angles*. This is a two dimensional array of size two by six, that is declares in the file main.c (and is available to this file by declaring it as **#extern**) and contains all the required, for the walking animation, angles. Its structure is such that will keep six rotation angles (upper arm, lower arm, hand, upper leg, lower leg and foot) for both sides (left and right arms and legs).

Following that, the function **Draw_Upper_Leg** is called in order to draw the upper part of the leg. Next the centre of the axes is moved to the new required position by calling the routine **glTranslate** and the rest of the function continue to a similar to the just described manner (rotate axes, draw part and move the centre of the axes to the new position). When the leg is created (including upper leg, lower leg and foot) the function **glPopMatrix** is used to restore the initial (prior to this function) matrix.

Copyright (C) Fotis Chatzinikos 1999

Now that two functions are ready, one that draws a base and one that draws a leg, it is quite straight forward what is needed in order to have the completed (for this section) model. Example 3.6 shows the code needed in order to build finally the basic model.

Example 3.6 The function that creates the basic model

```
void Draw_Base_Legs(void)
{
 glPushMatrix() ;
 glTranslatef(0.0,base_move,0.0) ;
 Draw_Base(WIRE) ;
 glTranslatef(0.0,−(BASE_HEIGHT),0.0) ;
 glPushMatrix() ;
 glTranslatef(TORSO_WIDTH * 0.33,0.0,0.0) ;
 Draw_Leg(LEFT,WIRE) ;
 glPopMatrix() ;
 glTranslatef(−TORSO_WIDTH * 0.33,0.0,0.0) ;
 Draw_Leg(RIGHT,WIRE) ;
 glPopMatrix() ;
}
```

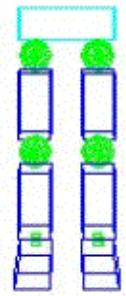


Plate 3.5 The basic model

As it is seen in example 3.6, just after saving the current matrix by calling the routine **glPushMatrix** a call to the routine **glTranslate** is done with one of its parameters being the value *base_move*. The particular call will be explained in a while. Following that, the base is drawn by calling the function **Draw_Base**. Next the centre of the axes is moved lower in order to draw the legs. The matrix is saved, the axes are moved to the left and the left leg is drawn; the matrix is restored, the axes are moved to the right and the right leg is drawn. Finally the routine **glPopMatrix** is called in order to restore the initial matrix. If this program is compiled and run the results will be the ones shown in Plate 3.5

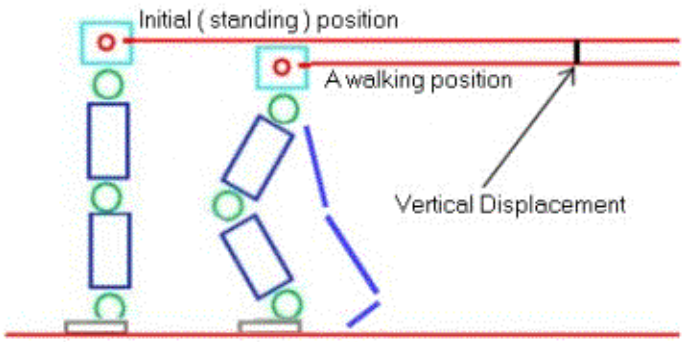


Plate 3.6 Vertical displacement of the body due to the legs angle

In example 3.6 the first call to the routine **glTranslate** was left without an explanation. As it can be seen a value is passed to this routine, named *base_move*. This value is the vertical displacement of the body, due to

Copyright (C) Fotis Chatzinikos 1999

the walking animation. When a human walks, its torso does not remain at the same point but moves slightly up and down due to the angle of the legs (Plate 3.6). Example 3.7 contains the function that calculates this vertical displacement.

Example 3.7 The function that calculates the vertical displacement of the body

```
double find_base_move(double langle_up, double langle_lo, double rangle_up, double rangle_lo)
{
 double result1, result2, first_result, second_result, radians_up, radians_lo ;
 radians_up = (PI*langle_up)/180.0 ;
 radians_lo = (PI*langle_lo−langle_up)/180.0 ;
 result1 = (UP_LEG_HEIGHT + 2*UP_LEG_JOINT_SIZE) * cos(radians_up) ;
 result2 = (LO_LEG_HEIGHT + 2 * (LO_LEG_JOINT_SIZE + FOOT_JOINT_SIZE) + FOOT_HEIGHT)
            * cos(radians_lo) ;
 first_result = LEG_HEIGHT − (result1 + result2) ;
 radians_up = (PI*rangle_up)/180.0 ;
 radians_lo = (PI*rangle_lo−rangle_up)/180.0 ;
 result1 = (UP_LEG_HEIGHT + 2*UP_LEG_JOINT_SIZE) * cos(radians_up) ;
 result2 = (LO_LEG_HEIGHT + 2 * (LO_LEG_JOINT_SIZE + FOOT_JOINT_SIZE) + FOOT_HEIGHT)
            * cos(radians_lo) ;
 second_result = LEG_HEIGHT − (result1 + result2) ;

 if (first_result <= second_result)
 return (− first_result) ;
 else
 return (− second_result) ;
}
```

As it can be seen in Plate 3.7 the vertical displacement *VD* can be calculated by subtracting the values *upper_leg_vertical* and *lower_leg_vertical* by the leg's length, LL:

$$VD = LL - (upper\_leg\_vertical + lower\_leg\_vertical) \quad (1)$$

At this point the vertical displacement due to the foot is not taken into account.

Back to the function **find_base_move**, the angles are firstly converted from degrees to radians (as the library routine **cos** that is used to find the cosine of the angles needs the angles to be in radians). Then the previously defined function (1) is used to find the vertical displacement. In the function, *VD* is represented as *final_result*, *upper_leg_vertical* as *result1* and *lower_leg_vertical* as *result2*. To find *result1* and *result2* the following functions are used (consult Plate 3.7):

$$result1 = X * \cos(r) \quad (2)$$

$$result2 = Y * \cos(f) \quad (3)$$

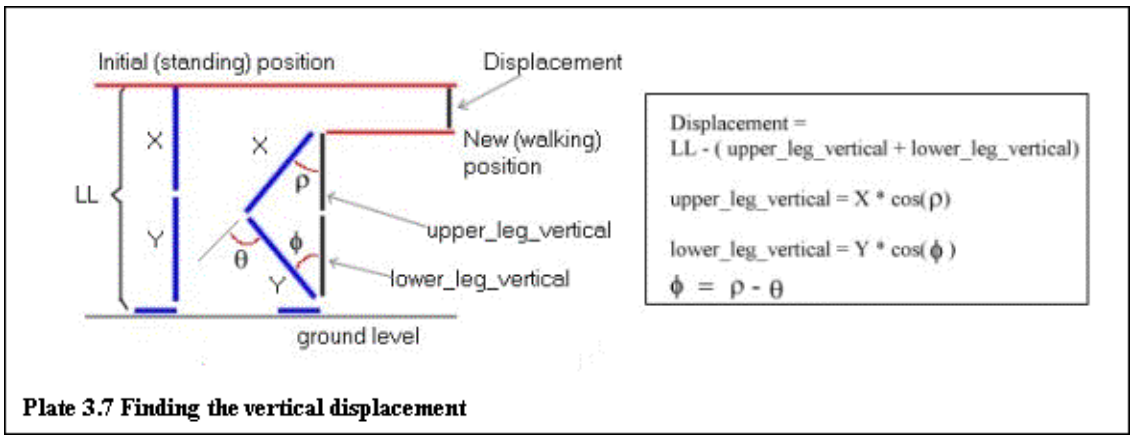


Plate 3.7 Finding the vertical displacement

Copyright (C) Fotis Chatzinikos 1999

The vertical displacement for both legs is found and then a check is done to see which one of the two is touching the ground (its vertical displacement will be less than the others will); this value is then returned by the function.

At this point there is available to the user a function that draws a basic model; there is also a function that is able to calculate the vertical displacement of this particular model. A remaining function to construct, is a function that will give life to this model, an animation function that will make the model walk.

In this, first section, of the chapter the angles of the walking animation will be ‘hardwired’, meaning that the program will not read them from a file but they will exist in the body of the animation function. Later, in the second section of this chapter, this will change, as the program will become data driven (it will read all its data from files).

This function will be based on the technique of key framing. This technique firstly identifies a number of key frames. These key frames are frames where something important for the animation happens. At these key frames the angle of every part of the body will be provided to the program, meaning that the programmer will explicitly calculate and pass these angles to the function. Then the function will use these key frames to calculate the angles of every part of the body for every frame of the animation.

This will be accomplished by taking the angle between two key frames and divide this angle among the other frames. For example, if the lower part of the leg has to be moved twenty degrees between two key frames and this has to be done in twenty frames, the function will calculate that the lower part of the leg has to be moved one degree every single frame (twenty degrees divided by twenty frames = one degree per frame), in order to accomplish the stated need.

The walking animation function was based on the book by Tony Wight “Moving Pictures”. In this book a walking animation cycle was provided based on eight key frames. The first four key frames were used in order to animate the first half of the walking movement and the rest four in order to animate the second half. The second half of the animation is the same as the first part but in reverse. In the first half of the animation the leg that was in front before the animation starts will end up being behind and the leg that was behind will end up in front. The second half of the animation does just the reverse of first half of the animation in order to complete the walking cycle and start from the beginning for a new cycle.

Copyright (C) Fotis Chatzinikos 1999

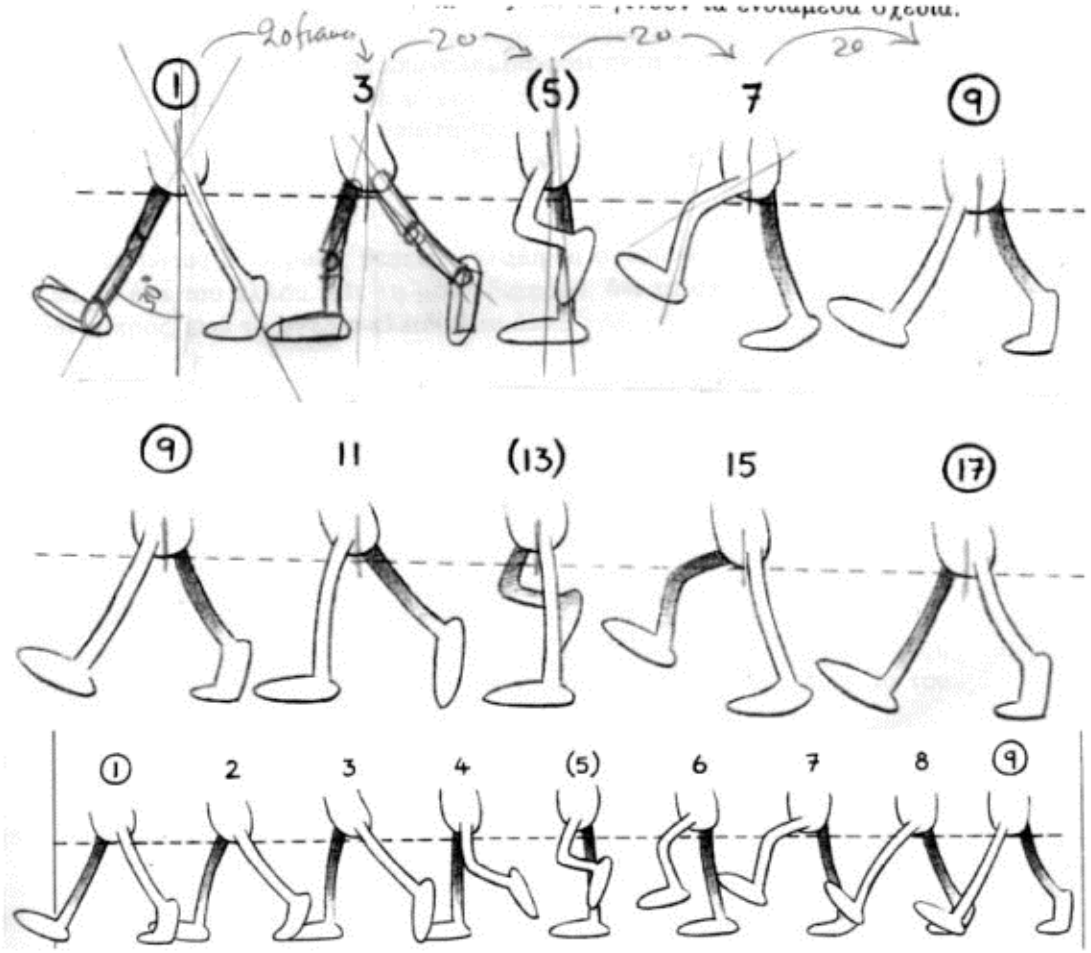


Plate 3.8 Walking Cycle as found in "Moving Pictures"

The animation function that was build for the previously discussed model, is based on the sketches found in this book, so its structure follows the structure that was described in the previous paragraph. The angles used were calculated from the sketches in the book. Plate 3.8 shows the walking cycle as appeared in the book. Example 3.8 contains part of the code of this function.

Example 3.8 Part of the walking animation function

```
void animate_base(void)
{
 static frames = FRAMES,
 zoom_fl = 0,
 flag = 1 ;
 float l_upleg_dif ,
 r_upleg_dif ,
 l_upleg_add ,
 r_upleg_add ,
 l_loleg_dif ,
 r_loleg_dif ,
 l_loleg_add ,
 r_loleg_add ;
 switch (flag)
 {
 case 1 :
 l_upleg_dif = 15 ;
 r_upleg_dif = 5 ;
```

Copyright (C) Fotis Chatzinikos 1999

```
 l_loleg_dif = 15 ;
 r_loleg_dif = 5 ;
 l_upleg_add = l_upleg_dif / FRAMES ;
 r_upleg_add = r_upleg_dif / FRAMES ;
 l_loleg_add = l_loleg_dif / FRAMES ;
 r_loleg_add = r_loleg_dif / FRAMES ;
 walking_angles[0][3] += r_upleg_add ;
 walking_angles[1][3] += l_upleg_add ;
 walking_angles[0][4] += r_loleg_add ;
 walking_angles[1][4] += l_loleg_add ;
 langle_count −= l_upleg_add ;
 langle_count2 −= l_loleg_add ;
 rangle_count −= r_upleg_add ;
 rangle_count2 −= r_loleg_add ;


 base_move = find_base_move ( langle_count, langle_count2, rangle_count, rangle_count2 ) ;
 frames−− ;
 if (frames == 0)
 {
 flag = 2 ;
 frames = FRAMES ;
 }
 break ;

 case 2 :
…… repeat until case 8 then go to case 1……
 if (zoom_flag)
 {
  switch (zoom_fl)
  {
  case 0 :
  zoom += 0.05 ;
  if (zoom > 2.5) zoom_fl = 1 ;
  break ;
  case 1 :
  zoom −= 0.05 ;
  if (zoom < −2.5) zoom_fl = 0 ;
  break ;
  default :
  break ;
  }
 }
 if (rotate_flag)
 {
 rotate = (rotate + 1) % 360 ;
 }
 glutPostRedisplay() ;
}
```

At the start of this function some variables are declared. The variables *frames*, *zoom_fl* and *flag* are declared as **static** because they are needed to be initialised only once (the first time the function is called). Just after the variables declaration a **switch** statement follows. This is the skeleton of the function, as all the operations needed to be done for the walking animation happen inside this statement.

This **switch** statement depends on the variable *flag*, which is initially set to 1. This means that the first part of this statement (the one under the label ‘case 1 :’) will be executed until the variable *flag* changes from 1 to a different value (in this case it will eventually become 2).

Inside this part of the **switch** statement, the variables l_upleg_dif, l_loleg_dif, r_up_leg_dif and r_loleg_dif are initialised to some values. These values are the difference of the angles of the left and right upper and lower leg between the first two key frames.

Copyright (C) Fotis Chatzinikos 1999

After this the variables l_upleg_add, l_loleg_add, r_upleg_add and r_loleg_add are calculated, by dividing the initial angle difference (between the two key frames) by the number of frames that are needed in order to make the animation. These will be the roatation values for a single frame animation. The number of needed frames between two key frames is constant and is defined in this case as twenty in the file model.h.

The next step is to copy the values of the previously calculated variables in the proper places in the array *walking_angles*. This, externally defined array that was described previously is used in the **draw_base** function in order to animate the model. The values of the variables are not just copied but they are added to the previous value of the array in order that the array will contain the angles for the next frame. This is done because the rotation is not incremental; for example if the function **glRotate**(20, 1.0, 0.0, 0.0) was used to rotate the upper leg by twenty degrees and at the next step the upper leg is needed to be rotated by five degrees more, the correct call will be **glRotate**(25, 1.0, 0.0, 0.0) and not **glRotate**(5, 1.0, 0.0, 0.0.). *OpenGL* follows this non−incremental technique in order to diminish cumulative errors that may appear if this particular modelling transformation was based on an incremental technique.

After this is done these values are subtracted from the variables langle_count, langle_count2, rangle_count and rangle_count2. These four variables are initialised externally, in the file main.c, and contain the initial values of the angles of the body parts. These variables will be used with the function **base_move** in order to calculate the body's vertical displacement. By doing so the values of the variables r (*angle_count) and q (*angle_count2) of both left and right legs (l / r) are retrieved (review Plate 3.7).

After the new angles are calculated by the technique explained in the previous paragraph, the values of these variables are passed to the function **base_move**, in order to find the vertical displacement of the body.

This is the end of the first cycle (transition from key frame one to key frame two). The variable *frames* is decrement and a check is done to see if the value of *frames* is equal to 0. If it is 0, it means that the second key frame is reached and that the value of the variable *flag* must be incremented (in order to move to the next **case** in the **switch**, the second cycle). The variable *frames* is also reinitialised to FRAMES (the defined, constant number of frames between two key frames).

This will continue until the end of **case** 8 will be reached and then *flag* will be set to 1 for the walking cycle to start from the beginning. At the end of the switch statement some more code is visible in example 3.8. This code calculates the value of the variables *zoom* and *rotate*. These variables are used externally, in the main program to zoom in and out (on the z−axis) and rotate the model (on the y−axis).

Keys and their operation

s	Single frame animation
S	Continues animation
d	Start/Stop Rotation of the model
D	Start/Stop zooming in and

Now that all the main functions of the program are ready, only one is left in order to finish the program. This is the **keyboard** function that will provide the needed interaction between the user and the program. This function has the same structure as the one described in example 2.11, so it will not be examined here. For reference, table 3.1 contains the keys that are used by the program and their operations.

At this point the program is nearly ready, as only a couple of operations remain to be done in the main program in order to have a fully working program. In the main program all the previously described as external variables are declared. When this is done, the variables langle_count, langle_count2, rangle_count and rangle_count2 are initialised to the values 30, 0, −30 and 0. These, as described before, are the initial

Copyright (C) Fotis Chatzinikos 1999

angles of both left and right, upper and lower leg. The variables zoom_flag and rotate_flag are initialised to GL_FALSE (at first the model will not be zoomed or rotated) and the variables rotate and zoom are set to 0.0, as the model is initially not zoomed nor rotated.

Something new appears also in the function **init**. Example 3.9 contains the code of this function.

Example 3.9 The init function that prints out general information about the *OpenGL* version

```
void init(void)
{
 const GLubyte* information ;
 glClearColor(1.0, 1.0, 1.0, 0.0) ;

 glShadeModel(GL_FLAT) ;

 information = glGetString(GL_VENDOR) ;
 printf("VENDOR : %s\n", information) ;

 information = glGetString(GL_RENDERER) ;
 printf("RENDERER : %s\n", information) ;

 information = glGetString(GL_EXTENSIONS) ;
 printf("EXTENSIONS : %s\n", information) ;

 information = glGetString(GL_VERSION) ;
 printf("VERSION : %s\n", information) ;

 walking_angles[0][3] = langle_count ;
 walking_angles[1][3] = rangle_count ;
 walking_angles[0][4] = langle_count2 ;
 walking_angles[1][4] = rangle_count2 ;

 base_move = find_base_move( langle_count, langle_count2, rangle_count, rangle_count) ;

}
```

In this function the variable information (of type Glubyte pointer) is used with the *OpenGL* function **glGetString** in order to retrieve and then print general information about the *OpenGL* version, vendor, extensions supported, etc.

In this function the array that is used to store the angles is also initialised. The initial vertical displacement of the model is found also by calling the function **base_move** and passing the legs initial angles.

A new callback function is also used in this program. The function **glutSpecialFunc** is similar to the **glutKeyboardFunc** but is used to register a callback function responsible for the keys that do not generate an ASCII code, like the directional keys, the Control and Alt key, and the Function keys (F1 to F12). The structure of this function is similar to the one of the keyboard function shown in example 2.11. After registering the function **special** by calling the function **glutSpecialFunc** in the **main** function the user will be able to use the up and down directional keys to zoom in and out of the model and the left and right directional keys to rotate the model on the y−axis. The calls to the routines **glTranslatef** (0.0, 0.0, zoom) and **glRotatef** (rotate, 0.0, 1.0, 0.0) just before drawing the model in the **display** function will allow for the zooming and rotation effects.

Copyright (C) Fotis Chatzinikos 1999

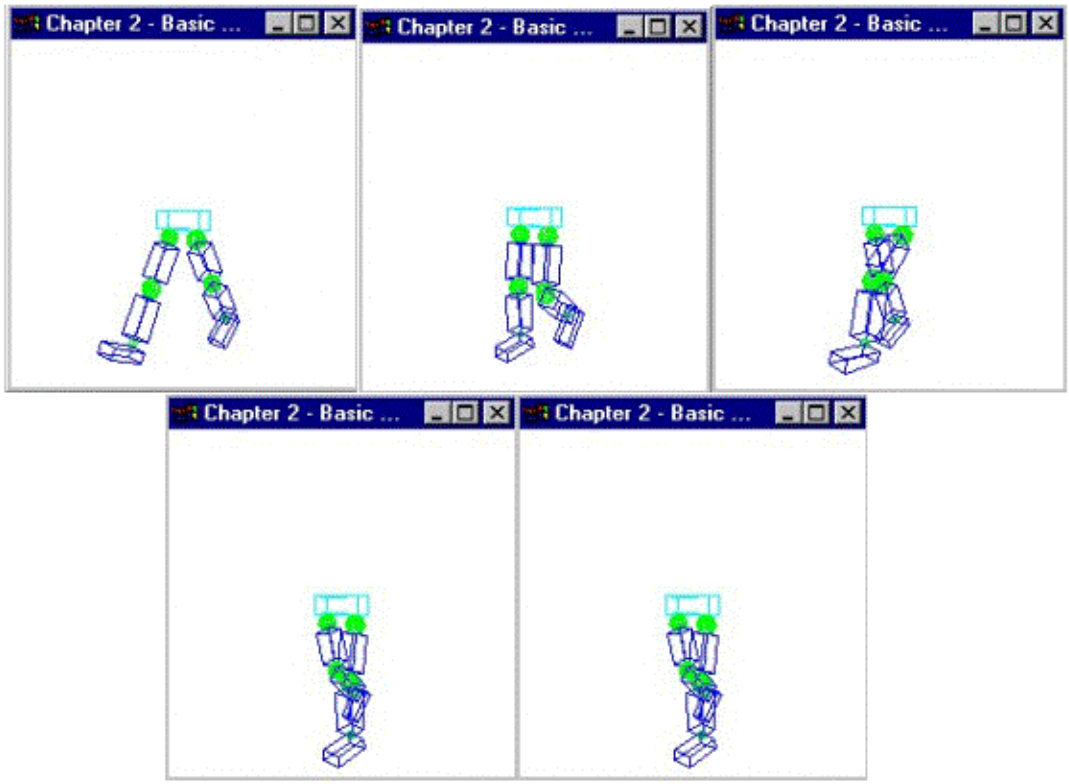


Plate 3.9a Snapshots from the walking basic model

At this point, the first section of the second chapter is completed. After compiling and run the program the user will be able to see this basic model walking. Plate 3.9 contains some screen−shots that were taken from this program.

Copyright (C) Fotis Chatzinikos 1999

Plate 3.9b Snapshots from the walking basic model

3.2 Improving the basic model

The goal of this section is to create a wireframe model of a man that will be able of walking. This program will be based on the previously constructed (in section 3.1) program. The structure of this new program will be similar to the previous one with the difference that this second program will have a more ‘professional’ touch. For example the animation angles will be read from a file instead of being hardwired in the walking procedure; the program will also be split into five different parts that each one of them will contain relevant functions. These five files are going to be main_model.c, model.c, inout.c, anim.c and keyboard.c. Each one of them will also have its header file. Another file will also be constructed that will contain general definitions, general.h.

This section’s keyboard interaction is the same as in the previous section, so no particular interest will be given to it. The animation function, **animate_body**, is also similar to the one used in the previous program, the only difference being that now, instead of calculating the walking angles of the legs, it calculates the walking angles of the arms also. The technique used to calculate the arms walking angles is the same as the previously explained one (the one that is used to calculate the legs walking angles), so there is no reason for explicit demonstration of this new function.

Copyright (C) Fotis Chatzinikos 1999

The file inout.c contains the newly created functions that are responsible for file input−output (reading the angles from a file and other file related functions). The custom function **Open_Files** was created for this particular program and accepts one argument. Depending on the argument it can open two different files. If the argument is ‘r’ it will try to open the file ‘data.txt’, for reading, from three pre−defined directories, ‘e:\bp\chapter3\model2\’, ‘g:\data\’, or ‘a:\’, in order to read the walking animation angles. If the file is not found in this directories the function notifies the user that the file was not found and the program exits. If the argument is ‘w’ the function will try to open the file ‘test.txt’ for writing in the same three directories. This function will be improved in a later example in order to check for the files, not in previously defined directories but in the directory the actual program is found. The file ‘test.txt’ is used later on from a function in order to print the walking angles for testing reasons.

The next function implemented in this file will use the structure *anim_angles* to store the animation angles read from the file. This structure is defined in the file general.h and can be found in example 3.10.

Example 3.10 The definition of the structure *anim_angles*

```
typedef struct
{
 float head ;
 float upbody ;
 float lobody ;
 float l_uparm ;
 float l_loarm ;
 float l_hand ;
 float l_upleg ;
 float l_loleg ;
 float l_foot ;
 float r_uparm ;
 float r_loarm ;
 float r_hand ;
 float r_upleg ;
 float r_loleg ;
 float r_foot ;

} anim_angles ;
```

As it can be seen in this example the structure *anim_angles* contains fifteen elements of type *float*. Each one of these elements will store the animation angle for a particular body part, for example the left upper arm(*l_uparm* in the structure), etc.

The function **Read_Data_From_File** calls the previously described function **Open_Files**(‘r’) in order to open the file data.txt for reading. This function accepts two arguments of type *anim_angles*. The first one named *init* is actually a pointer to the particular structure and is used to store the initial (prior to the animation) angles. The second argument, *array*[], is an array of four *anim_angles* elements. In this array the angles of the first four key frames will be stored. As the key frames are symmetrical (the last four to the first four) the values of this array will also be used to find the angles of the last four keyframes. The structure of this function is very simple as it just uses **fscanf** calls to read the angles from the file and place them in one of the two just mentioned variables (*init* or *array*). An integer variable named *scan_counter* is also used in this function to count how many values are actually read from the file. The number of the angles read is then output to the screen for testing reasons.

The last function implemented in this file is the function **Write_Test_Data**. This function accepts the same two arguments, the function **Read_Data_From_File** had, but instead of initialising them it uses them to output their values into a file, for testing reasons. Its structure is very simple, as it just calls the function **Open_Files**(‘w’) to open the file ‘test.txt’ for writing and then with several **fprintf** calls, it writes the angles of the animation in the file. By comparing the two files, ‘data.txt’ and ‘test.txt’ a user can find out if the program reads in the correct animation angles.

Copyright (C) Fotis Chatzinikos 1999

Now is the time to take a look at the contents of the file model.c. This file contains all the functions that are responsible for drawing the model on the screen. It contains all the previously implemented model functions like **Draw_Upper_Leg**, **Draw_Lower_Leg**, **Draw_Foot**, **Draw_Leg**, etc.

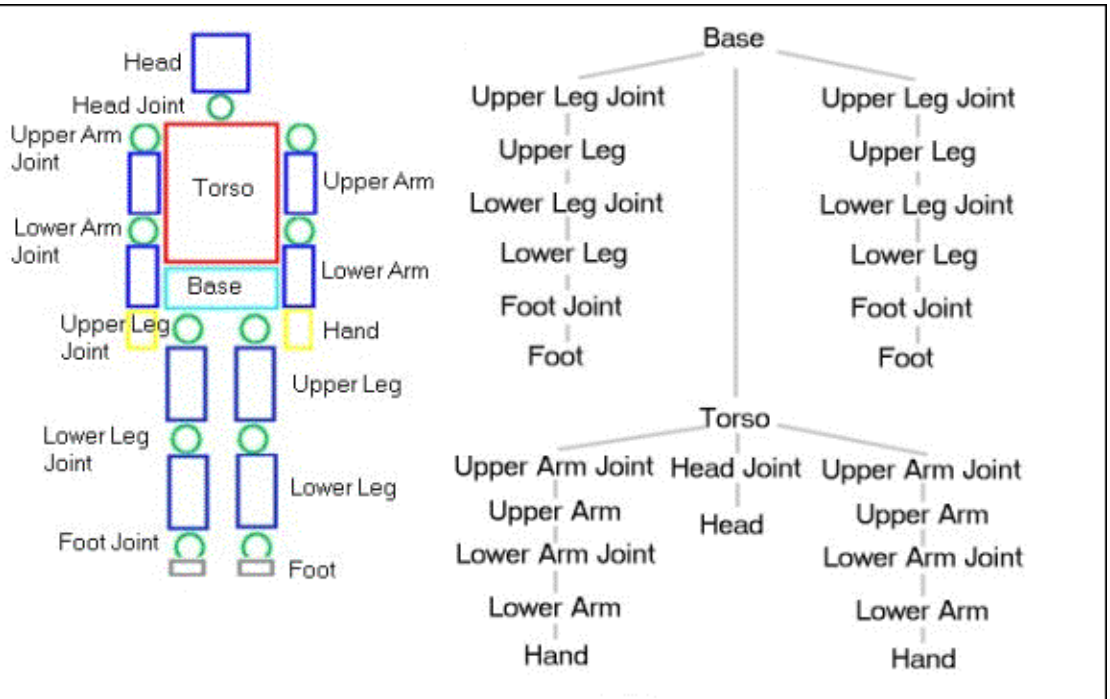


Plate 3.10 Complete body parts and their hierarchy

It also contains the newly created functions that draw the head, the upper arm, the lower arm, the hand and the torso. These functions were constructed in a manner similar to the one described in the first section of the chapter. A function that draws all the parts together, in order to draw the complete model of a man, was also created and implemented in this file. This pseudocode of this function, named **Draw_Model** can be seen in example 3.11. Plate 3.10 contains the parts that constitute the new improved model and their hierarchy.

Example 3.11 The pseudo−code of the function that creates and draws the complete model of a man.

```
function Draw_Model
{
 save_the_matrix (prior to this function)
 create_base
 save_the_matrix (to place the second in the hierarchy torso)
 translate_to_correct_place
 create_torso
 save_the_matrix(to place the third in the hierarchy head)
 translate_to_correct_place
 create_head
 restore_the_matrix
 restore_the_matrix
 save_the_matrix (to place the second in the hierarchy arms)
 translate_to_correct_place
 create_left_arm
 translate_to_correct_place
 create_right_arm
 restore_the_matrix
 save_the_matrix (to place the second in the hierarchy legs)
 translate_to_correct_place
 create_left_leg
 translate_to_correct_place
 create_right_leg
 restore_the_matrix
```

Copyright (C) Fotis Chatzinikos 1999

```
 restore_the_matrix
}
```

The structure of this function is similar to the structure of the function **Draw_Base_Legs**, which was described in the first section of this chapter.

Firstly the matrix prior to this function is saved, then the base is created and the matrix is saved again (to place the second in the hierarchy) torso. The centre of the co−ordinates is moved to the correct place and the torso is created. The matrix is saved again (to place the third in the hierarchy) head, the co−ordinates are moved to the new place and the head is created. The matrix is restored twice (to climb up the hierarchy twice) and the just explained technique is repeated in order to create the legs and arms. Example 3.12 contains the code of the function **Draw_Model**, based on the pseudo−code shown in example 3.11.

Example 3.12 The function that creates and draws the complete model of a man.

```
void Draw_Model(int frame)
{
 glPushMatrix() ;
 glTranslatef(0.0,base_move,0.0) ;
 Draw_Base(frame) ;
 glPushMatrix() ;
 glPushMatrix() ;
 glTranslatef(0.0, TORSO_HEIGHT / 2.0, 0.0) ;
 Draw_Torso(frame) ;
 glPopMatrix() ;
 glPushMatrix() ;
 glPushMatrix() ;
 glTranslatef(0.0, TORSO_HEIGHT + (HEAD_HEIGHT/2.0) +HEAD_JOINT_SIZE * 2.0, 0.0) ;
 Draw_Head(frame) ;
 glPopMatrix() ;
 glPopMatrix() ;
 glPushMatrix() ;
 glTranslatef(0.0,TORSO_HEIGHT * 0.875,0.0) ;
 glPushMatrix() ;
 glTranslatef(TORSO_WIDTH * 0.66, 0.0,0.0) ;
 Draw_Arm(LEFT,frame) ;
 glPopMatrix() ;
 glTranslatef(− (TORSO_WIDTH * 0.66), 0.0,0.0) ;
 Draw_Arm(RIGHT,frame) ;
 glPopMatrix() ;
 glPushMatrix() ;
 glTranslatef(0.0,−(BASE_HEIGHT*1.5),0.0) ;
 glPushMatrix() ;
 glTranslatef(TORSO_WIDTH * 0.33,0.0,0.0) ;
 Draw_Leg(LEFT,frame) ;
 glPopMatrix() ;
 glTranslatef(−TORSO_WIDTH * 0.33,0.0,0.0) ;
 Draw_Leg(RIGHT,frame) ;
 glPopMatrix() ;
 glPopMatrix() ;
}
```

Now that the functions that draw and animate the body are ready, only a couple of steps remain before having a complete and ready to run program.

The file main_model.c is similar to the first sections, main.c file. The only differences being the declaration of the variable *init_angles* and the array *angles*[4], that will be used from the in−out functions to store the animation angles.

Copyright (C) Fotis Chatzinikos 1999

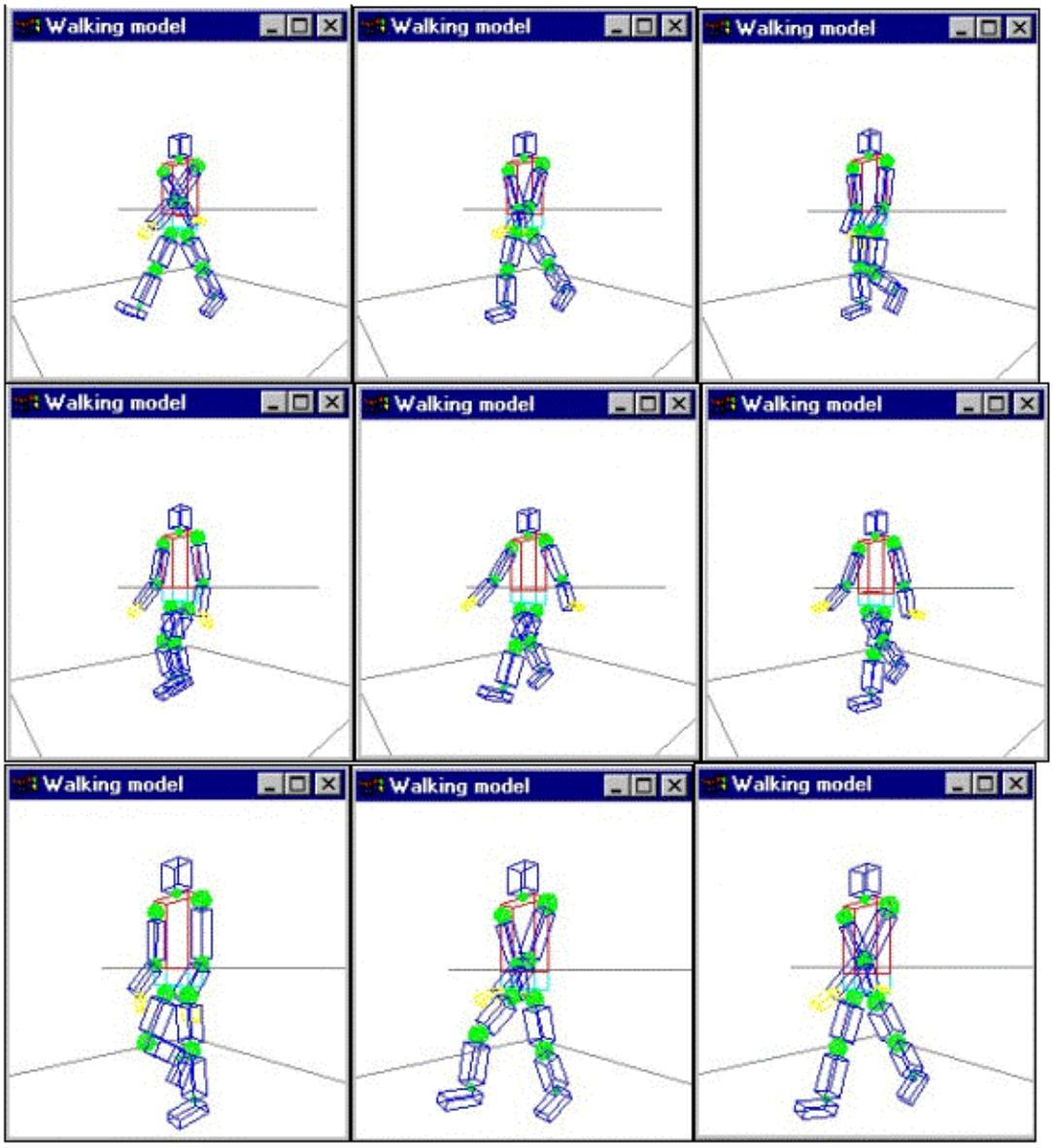


Plate 3.11 Snapshots from the walking improved model (wireframe)

A second difference is found in the function **display**, where instead of the call **Draw_Base_Legs** that was used in the previous section to draw the incomplete model, a call to **Draw_Model** is done to draw the complete model. Prior to drawing the model some code can be found. This code (shown in example 3.13) is responsible for creating a wireframe rectangle (at the place of where the floor should be) and a horizontal line near the ‘base’ of the model. These two serve as reference for the user, in order to help him see the vertical displacement of the body.

Copyright (C) Fotis Chatzinikos 1999

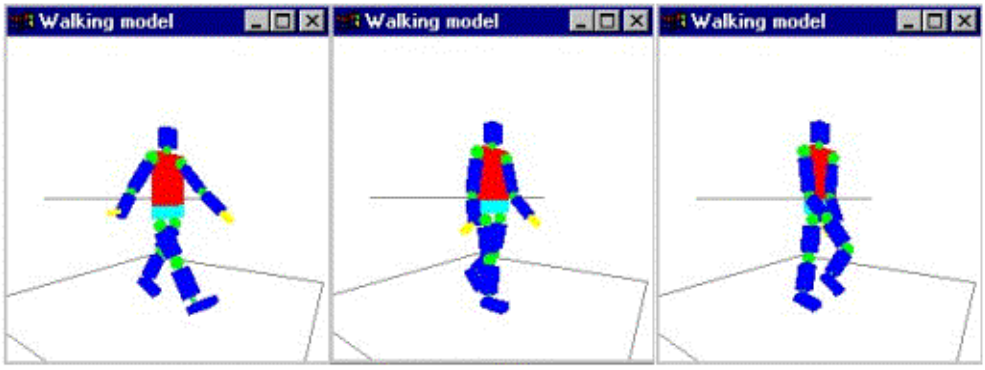


Plate 3.12 Snapshots from the walking improved model (solid)

At this point the program is ready. If it is compiled and run, the results can be found in Plate 3.11. If instead of using the value WIRE when calling the function **Draw_Model** the value SOLID is used, the results can be found in Plate 3.12. The solid model (shown in Plate 3.12) will be much improved in the next chapter with the addition of light.

Copyright (C) Fotis Chatzinikos 1999

Chapter 4– Lighting

As already demonstrated, OpenGL computes the colour of each pixel in a scene and that information is held into the frame buffer. Part of this computation depends on what lighting conditions exist in the scene and in which way objects absorb and/or reflect light. Actually in some cases the objects appear invisible until light is added.

Light is very important in real life as well as in graphics. For example, the sea looks bright green in the morning, blue during the day and black during the night. The colour of the water does not actually change, as it is always transparent, but the reflection conditions change.

By using OpenGL, the lighting conditions and the properties of the objects can be changed in order to produce many, sometimes stunning effects.

OpenGL approximates light and lighting as if light could be broken into red, green and blue components (R−G−B colour model). Thus, the colour of the light sources can be characterised from the amounts of the red, green and blue light they emit, and the material of an object characterised by the percentage of red, green and blue light it reflects in various directions.

In the OpenGL lighting model, light comes from several light sources in the scene that can be individually turned on and off. Some light comes from a particular direction and some light is scattered around the scene. For example, when you turn on a light bulb in a room, some light arrives at a particular object in the room directly from the bulb and some light arrives at the object after bouncing on one or more other surfaces, like walls, furniture, etc. This bounced light is called ambient and it is assumed that it is so scattered that it does not come from any particular direction, but from everywhere.

In the OpenGL lighting model, a light source has an effect only when some surface that absorbs and/or reflects light is present. Each surface is composed of a material that has various properties. A material might emit its own light (like headlights in cars), might absorb some percentage of the incoming light and might reflects some light in a particular direction.

The OpenGL lighting model considers light to be divided into four independent components: emissive, ambient, diffuse and specular. All four components are computed independently and then added together.

A fifth element might influence the appearance of an object and that is the shininess of the object. Depending on the shininess, a particular object reflects the incoming specular light in different ways.

This chapter is divided into five sections; each one dedicated to a particular lighting effect.

The first example draws four wireframe and four solid cubes. Some of them are drawn with lighting turned on, while some of them are drawn with lighting turned off in order to demonstrate the difference in appearance of objects when lighting is used.

In the second example, the OpenGL feature colour tracking is demonstrated. Normally when lighting is enabled, the objects must have a material assigned to them. Depending to the material used objects appear deferent when lit (like in the real world). For example a sphere that has a ‘wooden’ material (ambient, diffuse, specular and emissive values that approximate wood’s behaviour) will look different from a ‘silver’ sphere. A material (in OpenGL) is what colour is to programs that do not use lighting. By using the colour tracking feature a programmer might choose to assign colours to objects (instead of materials) and OpenGL will convert them to materials. In some cases this is quite useful, as it is much simpler to assign colour from assigning materials.

Assigning materials and manipulating their properties is the topic of the third example. Three red cubes are created and different values of shininess are assigned to each of them in order to demonstrate this particular

Copyright (C) Fotis Chatzinikos 1999

light property.

In the fourth section of the chapter a program called ‘Material−Lights’ will be created. A user will be able to change the material and light properties of several objects interactively in order to become familiar with the concept. The program will also be able to save a particular ‘colour’ (a combination of material and light conditions) for latter reference. The concept of windows and sub−windows will also be discussed in this section.

Finally, the goal of the fifth section of this chapter will be to improve the previously constructed model of a man. The model that was created in the last section of the previous chapter will be taken and its structure will be slightly modified in order to become a solid, lighted model.

4.1 Getting started with lighting

In this first section of this chapter, the necessary steps to create a light source will be explained. When lighting is used with objects that are not supposed to be drawn using lighting some strange effects appear. In order to demonstrate these effects and the correct use of lighting, this example draws four wireframe cubes and four solid cubes, each one of them with different lighting conditions. These lighting conditions will be a combination of the following: lighting enabled, lighting disabled, depth testing enabled and depth testing disabled. Depth testing is an OpenGL feature that does hidden surface removal by using the depth buffer.

When drawing solid, lighted objects it is very important to draw the objects that are nearer to the viewing position and eliminate any objects obscured by others nearer to the eye.

The elimination of parts of solid objects that are obscured by others is called hidden−surface removal. The easiest way of achieving this in OpenGL is to use the depth buffer. In order to use the depth buffer, a window must be created that will have such a buffer. Passing the argument GLUT_DEPTH in the function **glutInitDisplayMode** does this. When this is done the OpenGL function **glEnable** can be called with the value GL_DEPTH_TEST in order to add hidden−surface removal to the particular program.

This program is based on the example in the fourth section of chapter 2. It uses all the functions that were defined there in order to draw and position the eight cubes (four wireframe and four solid ones).

A difference is that all the cubes are drawn by using perceptive projection and that **glEnable** and **glDisable** statements appear inside the display function in order to activate and deactivate lighting and hidden−surface removal.

Before using lighting, at least one of OpenGL’s lights must be enabled. For this example one light is enough, and passing the value GL_LIGHT0 to the routine **glEnable** (inside the body of the **init** function) has the effect of activating one light. Different OpenGL implementations may provide different amounts of lights but all of them have at least eight lights. Example 4.1 contains the display function of this program.

Example 4.1 Display function that draws eight cubes with/without lighting and depth testing

```
void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT ) ;

   glViewport(0,win_size_V / 2, win_size_H / 4 ,win_size_V / 2) ;
   glDisable(GL_LIGHTING) ;
   glDisable(GL_DEPTH_TEST) ;

   Draw_Cube_Transl_Rot(WIRE) ;
```

Copyright (C) Fotis Chatzinikos 1999

```
   glViewport(win_size_H / 4,win_size_V / 2, win_size_H / 4 ,win_size_V / 2) ;
   glEnable(GL_DEPTH_TEST) ;
   Draw_Cube_Transl_Rot(WIRE) ;

   glViewport(2 * (win_size_H / 4),win_size_V / 2, win_size_H / 4 , win_size_V / 2) ;
   glEnable(GL_LIGHTING) ;
   glDisable(GL_DEPTH_TEST) ;
   Draw_Cube_Transl_Rot(WIRE) ;

   glViewport(3 * (win_size_H / 4),win_size_V / 2, win_size_H / 4 , win_size_V / 2) ;
   glEnable(GL_DEPTH_TEST) ;
   Draw_Cube_Transl_Rot(WIRE) ;

   glViewport(0,0, win_size_H / 4 ,win_size_V / 2) ;
   glDisable(GL_LIGHTING) ;
   glDisable(GL_DEPTH) ;
   Draw_Cube_Transl_Rot(SOLID) ;

   glViewport(win_size_H / 4,0, win_size_H / 4 ,win_size_V / 2) ;
   glEnable(GL_DEPTH_TEST) ;
   Draw_Cube_Transl_Rot(SOLID) ;

   glViewport(2*(win_size_H / 4),0, win_size_H / 4 ,win_size_V / 2) ;
   glDisable(GL_DEPTH_TEST) ;
   glEnable(GL_LIGHTING) ;
   Draw_Cube_Transl_Rot(SOLID) ;

   glViewport(3*(win_size_H / 4),0, win_size_H / 4 ,win_size_V / 2) ;
   glEnable(GL_DEPTH_TEST) ;
   Draw_Cube_Transl_Rot(SOLID) ;

   glutSwapBuffers() ;
}
```

As it seen in this example, the **display** function is divided into eight similar parts. Each one of them calls the routine **glViewport** in order to specify where the particular cube should be drawn. The first four cubes (upper part of the window) are drawn as wireframes, while the last four are drawn as solid (lower part of the window).

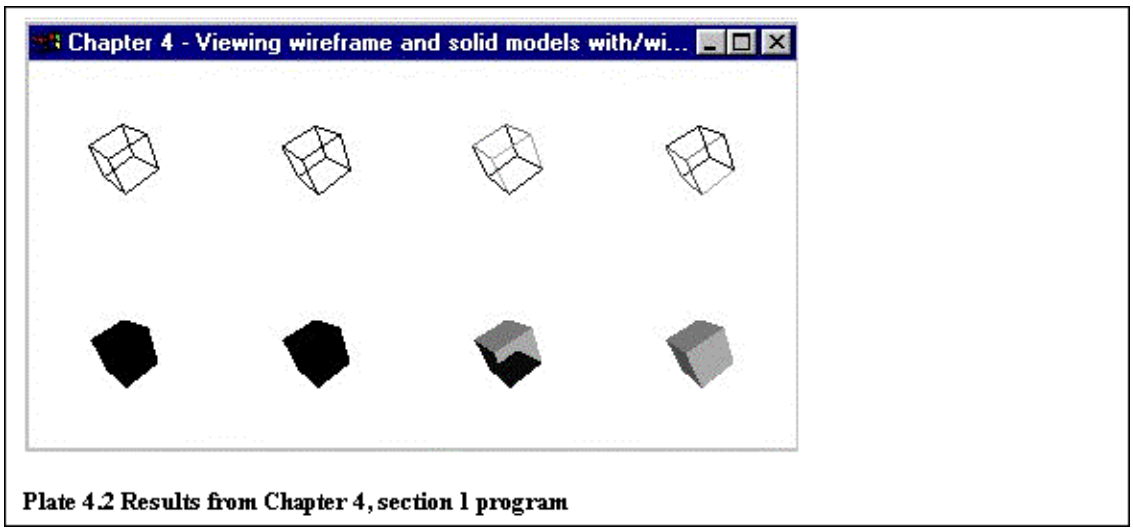


Plate 4.2 Results from Chapter 4, section 1 program

Plate 4.2 contains the results of the compiled and executed program. The remaining parts of the program are not discussed, as they are the same ones used in Chapter 2, section 4.

As seen in Plate 4.2 the best looking wire frame cube is the top, left−most cube and the best looking solid

Copyright (C) Fotis Chatzinikos 1999

cube is the bottom right−most one.

From this example some observations may be made. When drawing wire frame objects, depth testing does not have any effects (as it does hidden−surface removal and not hidden−line removal). Also when drawing wire frame objects all lights should be disabled, as in the opposite case the objects do not appear clear (i.e. the top, two cubes on the right of Plate 4.2).

On the other hand if the lower part of Plate 4.1 is observed, it can be seen that when solid models are drawn, lighting should be enabled, otherwise the objects do not appear three−dimensional. Depth testing should also be enabled when drawing solid, lighted models as in the opposite case (when depth−testing is disabled), the different parts of the object may be drawn in the wrong order with the results shown in the lower part of Plate 4.2, second cube from the right hand−side.

This happens because when depth testing is not enabled, no information is held about the depth of the objects on the screen relative to the viewpoint, so no calculation can be done in order to hide surfaces that are not visible.

4.2 Colour Tracking

As mentioned in the introduction of this chapter, colour tracking is an OpenGL feature that enables the programmer to assign colours instead of materials to objects that are going to be used in programs that use lighting. Colour tracking minimises also performance costs associated with material assigning.

This is a very useful feature of OpenGL, as it removes the overhead of having to assign manually the material properties of objects when something like that is not needed. If, for example a program just needs a simple red sphere and the properties of the material are of no importance (i.e. just a red sphere not a ‘wooden’ or ‘metal’ red sphere), the routine **glColor** can be used in conjunction with colour tracking in order to achieve the same effect more easily.

In order to demonstrate what colour tracking does, three solid cubes will be drawn on the screen each one having a different colour assigned to it (red, green and blue) with the routine **glColor**. Example 4.2 contains the code of the particular display function.

Example 4.2 Display function that draws three cubes (a red, a green and a blue one)

```
void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT ) ;

   glShadeModel(GL_SMOOTH) ;

   glViewport(0,0, win_size_H / 3 ,win_size_V) ;

   glColor3f(1.0,0.0,0.0) ;
   Draw_Solid_Cube_2() ;

   glViewport(win_size_H / 3,0, win_size_H / 3 ,win_size_V) ;
   glColor3f(0.0,1.0,0.0) ;
   Draw_Solid_Cube_2() ;

   glViewport(2*(win_size_H / 3),0, win_size_H / 3 ,win_size_V) ;
   glColor3f(0.0,0.0,1.0) ;
   Draw_Solid_Cube_2() ;

   glutSwapBuffers() ;
}
```

In this code, as it can be seen three viewports are defined, one for each.. Their colours are (from left to right)

Copyright (C) Fotis Chatzinikos 1999

red, green, and blue. The cubes are drawn by using the previously defined function **Draw_Cube_Transl_Rot** (Second Chapter). This function is slightly modified, in order to set the cube's colour outside the function.

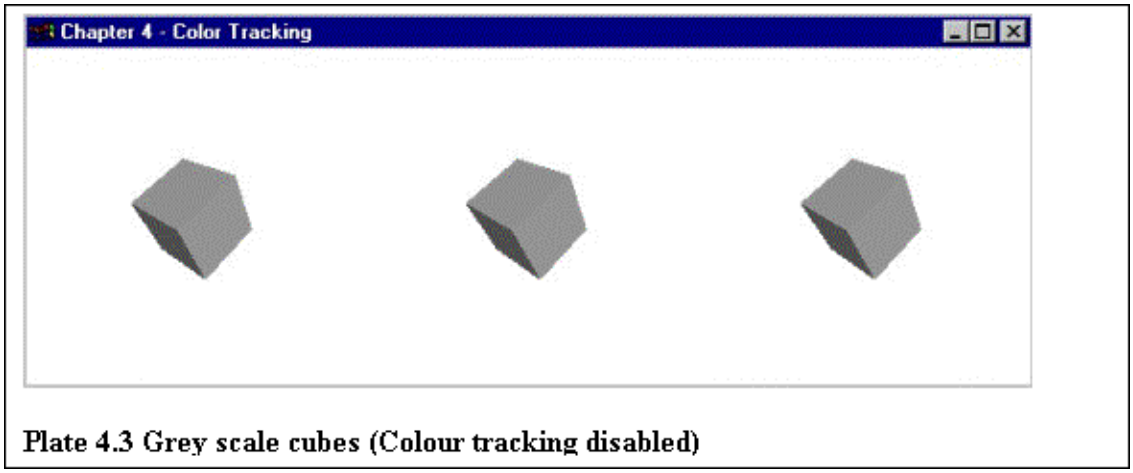


Plate 4.3 Grey scale cubes (Colour tracking disabled)

If this example is compiled and run, with lighting enabled (and the basic LIGHT0), the results will be the ones demonstrated in Plate 4.3.

The three cubes appear grey scaled and not colour because colour tracking was not enabled. As the material of the cubes was not specified, but instead calls to **glColor** were used, colour tracking must be enabled in order for the cubes to appear in colour.

This can be done either in the **display**, or in the **init** function by calling the routine **glColorMaterial**. This function accepts two arguments, the first one being the polygon face that colour tracking is to be enabled and the second is one of the four light components (diffuse, specular, ambient and emissive).

The polygon face can be the back face (GL_BACK), the front face (GL_FRONT) or both the back and front face (GL_FRONT_AND_BACK). By default front−facing polygons are the polygons whose vertices appear in a counter−clockwise order on the screen. Using the function **glFrontFace**, and supplying the desired front−face orientation (either GL_CCW for counter−clockwise orientation or GL_CW for clockwise orientation) can change what appears to be front−facing polygons.

Plate 4.4 contains the results of the program if colour tracking is enabled with the parameters GL_FRONT and GL_DIFFUSE.

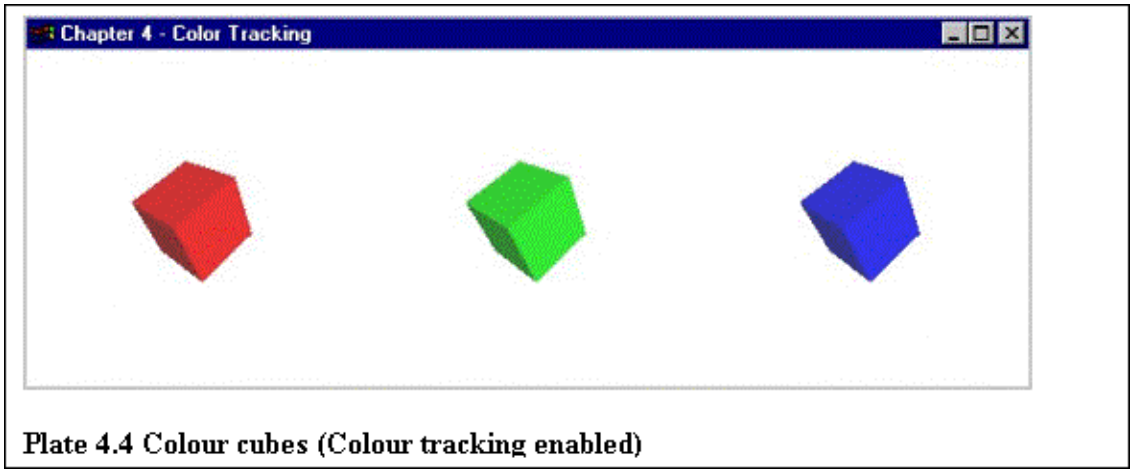


Plate 4.4 Colour cubes (Colour tracking enabled)

In order to use colour tracking the function **glEnable** must be called with the parameter GL_COLOR_MATERIAL, just after calling the function **glColorMaterial**.

Copyright (C) Fotis Chatzinikos 1999

4.3 Setting up an object's material properties and shininess

The subject of this section is the setting up of object's material. In this section instead of using the routine **glColor** in conjunction with colour tracking to create lighted objects, the more specific **glMaterial** will be used.

This routine will be used to specify the material's different components, diffuse, specular, emissive and ambient and how shiny objects are by setting the shininess. Because of the complex interaction between an object's material surface and incident light, specifying material properties so that an object has a desired, certain appearance is an art and is not something that can be learned from one moment to the other.

This routine, **glMaterial** accepts three arguments, the first being the face of the object that the material is going to be assigned, the second is the particular light component that needs to be set and the last one is a pointer to an array of values that will specify the appearance of the material (normally the array contains a red, a green, a blue and an alpha value). As mentioned before, the alpha value is used for blending and other 'special effects' and will not be used here. In the case of shininess the third parameter is not a pointer to an array but the actual value (0 to 128).

In this example the previously defined **Draw_Cube_Transl_Rot** will be slightly modified in order firstly to contain the appropriate material setting routines and secondly to draw a sphere instead of a cube (specular hilights are better shown on spheres, because of the larger amount of faces). Example 4.3 contains the code of this new function, called **Draw_Solid_Sphere**.

Example 4.3 Draw_Solid_Sphere funtion

```
void Draw_Solid_Sphere(GLfloat mat_diffuse[],GLfloat mat_specular[],
                           GLfloat mat_shininess[])

{
   glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse) ;
   glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular) ;
   glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess) ;


   glPushMatrix() ;
     glTranslatef(tran[TRANS][0],tran[TRANS][1],tran[TRANS][2]) ;
     glRotatef(tran[ROTATE][3], tran[ROTATE][0], tran[ROTATE][1], tran[ROTATE][2]) ;

     glScalef(tran[SCALE][0],tran[SCALE][1],tran[SCALE][2]) ;
     glutSolidSphere(1.0,16,16) ;
   glPopMatrix() ;
}
```

This function accepts three arrays of type Glfloat. These arrays contain the values of the diffuse, specular and shininess components of the material. In this program, the emissive and ambient properties of the materials are not changed.

The body of the function should appear familiar. The only difference from the function **Draw_Cube_Transl_Rot**, being the addition of the three **glMaterial** calls. As seen in the example, both three calls set the front−face of the polygon to the specified values of the particular material property.

Back in the main program these three arrays are initialised to the values shown in example 4.4.

Example 4.4 The arrays containing the material properties values

```
GLfloat mat_diff[] = {1.0, 0.0, 0.0, 1.0} ;
GLfloat mat_spec[] = {1.0, 0.0, 0.0, 1.0} ;
```

Copyright (C) Fotis Chatzinikos 1999

```
GLfloat mat_shin1[] = {0.0} ;
GLfloat mat_shin2[] = {5.0} ;
GLfloat mat_shin3[] = {50.0} ;
```

As seen in the example, the array *mat_diff*, that contains the values of the diffuse component of the material is set to red (1.0, 0.0, 0.0). The array *mat_spec* that contains the specular component values is also set to red. Three more array are specified called *mat_shin1*, *mat_shin2* and *mat_shin3*. These three arrays contain the shininess value of the three cubes that will be shortly drawn. Example 4.5 contains the code of the new display function. As it can be seen there, three viewports are defined and a red sphere of different shininess is rendered into each one of them. The results of this program can be seen in Plate 4.4.

Example 4.5 The display function that draws three red spheres with different shininess values

void display(void)

```
{
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT ) ;

   glShadeModel(GL_SMOOTH) ;
   glViewport(0,0, win_size_H / 3 ,win_size_V) ;
   Draw_Solid_Sphere(mat_diff,mat_spec,mat_shin1) ;

   glViewport(win_size_H / 3,0, win_size_H / 3 ,win_size_V) ;
   Draw_Solid_Sphere(mat_diff,mat_spec,mat_shin2) ;

   glViewport(2*(win_size_H / 3),0, win_size_H / 3 ,win_size_V) ;
   Draw_Solid_Sphere(mat_diff,mat_spec,mat_shin3) ;

   glutSwapBuffers() ;
}
```

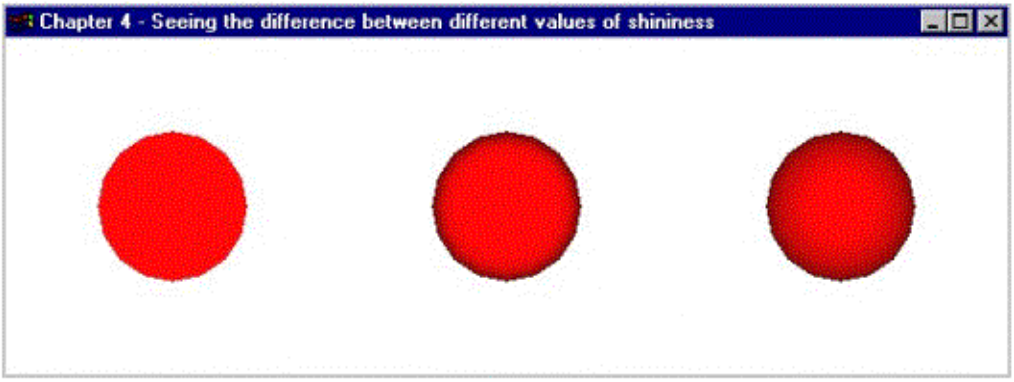


Plate 4.5 The effect of different shininess values

4.4 The Material – Lights program

The goal of this section is to create a program that a user will be able to see an object and how different materials and lights affect the appearance of the particular object. As this example is quite complicated, its discussion will be divided into several parts.

Firstly the appearance of the program will be considered. As the goal of this section is to show how different materials and lights affect an object, an object should appear on the screen. Also it should be clear by now that materials and lights are divided into deferent components. A material consists of four components (five if the shininess is included). These are the diffuse, the specular, the ambient and the emission. A light is also divided into components, and they are the diffuse, the specular and the ambient component.

Copyright (C) Fotis Chatzinikos 1999

All these components (except the shininess) are further composed from their red, green and blue elements. Some means of showing to the user the values of all these components and their elements should be found.

A solution was to create a function that would draw on the screen a graph, showing the red, green and blue values of a particular component. If now this function is used seven times, the four material components and the three light components could be visualised on the screen.

The problem is that orthographic projection should be used for drawing the graphs and perspective projection for drawing the objects. A solution to this problem would be to divide the window into several sub−windows, so that each sub−window could be assigned a different projection style.

It was then decided that eight sub−windows should be created. The main one would be used for drawing the objects and the other seven for drawing the seven material and light components.

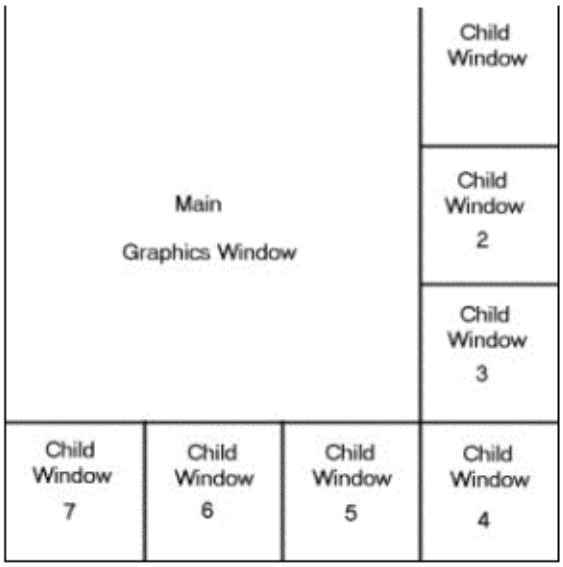


Plate 4.6 The position of the eight sub-windows

Plate 4.6 shows the positioning of the eight sub−windows.

The creation of the sub−windows can now start. *GLUT* provides a function named **glutCreateSubwindow** that can be used for this particular reason. This function accepts five arguments. The first one is the name of the parent window, the next two are the window initial x and y position and the last two are the window's width and height. Example 4.6 shows part of the main function that creates the main window and two of the sub−windows.

Example 4.6 Part of the main function that creates two sub−windows

```
parent_win = glutCreateWindow("Chapter 3 – Materials and Lights") ;

   init_parent() ;

   glutDisplayFunc(display_parent) ;
   glutReshapeFunc(reshape_parent) ;
   glutKeyboardFunc(keyboard) ;
   glutSpecialFunc(special) ;

   child_win = glutCreateSubWindow(parent_win, 3*(win_size_H/4), 0, win_size_H/4, win_size_V/4)

   init_child_mat() ;
```

Copyright (C) Fotis Chatzinikos 1999

```
   glutDisplayFunc(display_child) ;
   glutReshapeFunc(reshape_child) ;

   child_win2 = glutCreateSubWindow(parent_win, 3*(win_size_H/4), win_size_V/4, win_size_H/4,
                                    win_size_V/4) ;

   init_child_mat() ;

   glutDisplayFunc(display_child2) ;
   glutReshapeFunc(reshape_child) ;
```

As it can be seen in this example, the main window is created using the familiar function **glutCreateWindow**. Any callback functions that are needed for the main window are registered and then the first sub−window, named child_win is created by calling the function **glutCreateSubWindow**. Two callback functions are registered to this sub−window (a display and a reshape one) and then another sub−window is created by using the same technique.

As seen in the example both sub−windows use the same reshape function. This function (shown in example 4.7) just creates an orthographic projection.

Example 4.7 The sub−windows reshape function

```
void reshape_child(int w, int h)
{
   glViewport(0, 0,(GLsizei)w,(GLsizei)h) ;
   glMatrixMode(GL_PROJECTION) ;
   glLoadIdentity() ;
   glOrtho(10.0,70.0,−10.0,110.0,−1.0,1.0) ;
   glMatrixMode(GL_MODELVIEW) ;
   glLoadIdentity() ;
}
```

Now that the sub−windows are created, it is time to create their contents. As mentioned before, an object will be drawn in the main sub−window to show the effect of assigning different values to the material and light components. In order for the user to see better these effects, eight different objects will be available to him. The function that will draw these objects can be seen in example 4.8.

Example 4. 8 Function that draws one of eight possible objects

```
void Draw_Object(int object)
{
    glPushMatrix() ;
    glTranslatef(tran[TRANS][0],tran[TRANS][1],tran[TRANS][2]) ;
    glRotatef(tran[ROTATE][3], tran[ROTATE][0], tran[ROTATE][1], tran[ROTATE][2]) ;
    glScalef(tran[SCALE][0],tran[SCALE][1],tran[SCALE][2]) ;
    switch (object)
    {
       case 1 :
          glutSolidSphere(1.0,32,32) ;
          break ;
       case 2 :
          glutSolidCube(1.0) ;
          break ;
       case 3 :
          glutSolidCone(1.0,1.0,32,32) ;
          break ;
       case 4 :
          glutSolidTorus(0.3,0.7,32,32) ;
          break ;
       case 5 :
          glutSolidOctahedron() ;
          break ;
```

Copyright (C) Fotis Chatzinikos 1999

```
        case 6 :
           glutSolidTetrahedron() ;
           break ;
        case 7 :
           glutSolidIcosahedron() ;
           break ;
        case 8 :
           glutSolidTeapot(0.5) ;
           break ;
        default :
           break ;
     }
     glPopMatrix() ;
}
```

As seen in the example one of the following objects will be drawn to the screen depending on the value passed to the function: sphere, cube, cone, octahedron, tetrahedron, icosahedron, or teapot.

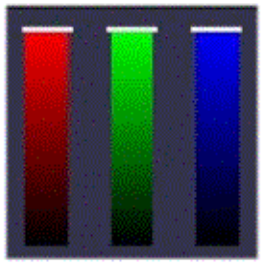


Plate 4.7 the result of the Draw_Graph function

The function that will draw the seven graphs (one by one) must be constructed now. Plate 4.7 shows what this function is needed to draw.

As seen in the Plate, this function should draw a graph containing a red, a green, and a blue bar. Each bar stands for one of the three elements of the material and light components (red, green and blue). Each bar will have an index that will show the current value of the element. The code of this function named Draw_Graph can be found in Example 4.9.

Example 4. 9 The Draw_Graph function

```
void Draw_Graph(GLfloat Red_Height, GLfloat Green_Height, GLfloat Blue_Height)
{
   GLfloat Red[]   = {1.0,0.0,0.0,1.0} ;
   GLfloat Green[] = {0.0,1.0,0.0,1.0} ;
   GLfloat Blue[]  = {0.0,0.0,1.0,1.0} ;

   glPushMatrix() ;
     Draw_Bar(15,0,Red,Red_Height) ;
     Draw_Bar(35,0,Green,Green_Height) ;
     Draw_Bar(55,0,Blue,Blue_Height) ;
   glPopMatrix() ;

   glutPostRedisplay() ;
}
```

As seen in the example this function accepts three arguments. These arguments are the red, green and blue values of the component's elements that will be passed to the function **Draw_Bar** in order to position the index of the bars in the correct position. The function **Draw_Bar** accepts four arguments. The first two are

Copyright (C) Fotis Chatzinikos 1999

used to position the bar inside the window the third one to colour the bar and the last one to position the bars index in the correct place.

The bars are drawn by using smooth shading in order to draw the lower part black and the upper part the specified colour. Using this technique the index's position can approximate the colour of the component's particular element. Example 4.10 shows the code of the **Draw_Bar** function.

Example 4. 10 The Draw_Bar function

```
void Draw_Bar(GLfloat x, GLfloat y,GLfloat color[],GLfloat height)
 {
    glPushMatrix() ;
    glBegin(GL_POLYGON) ;
    glColor3f(color[0] , color[1], color[2]) ;
    glVertex2f(x       ,y + 100.0) ;
    glVertex2f(x + 10.0,y + 100.0) ;
    glColor3f(0.0, 0.0, 0.0) ;
    glVertex2f(x + 10.0,y) ;
    glVertex2f(x       ,y) ;
    glEnd() ;
    glPopMatrix() ;
    glPushMatrix() ;
    glTranslatef(0.0,height*100,0.0) ;
    glBegin(GL_POLYGON) ;
    glColor3f(1.0,1.0,1.0) ;
    glVertex2f(x−1,y+2) ;
    glVertex2f(x+11,y+2) ;
    glVertex2f(x+11,y−1) ;
    glVertex2f(x−1,y−1) ;
    glEnd() ;
    glPopMatrix() ;
 }
```

As seen in the example this function is divided into two parts. The first part positions and creates a smoothly shaded rectangle while the second part uses the fourth argument of the function in order to position and draw the bar's index.

At this point nearly all the parts of the program used to demonstrate lighting effects are ready. A function, which still needs to be constructed, is the one that will be able to set the material and light properties. Actually two functions will be used for that purpose. The one named **Set_Material** will be responsible for setting up the object's material and the one called **Set_Light_ADS** will be used to set up the light components. Example 4.11 contains the **Set_Material** function and Example 4.12 contains the **Set_Light_ADS** function.

Example 4. 11 The Set_Material function

```
void Set_Material(GLenum pname, GLfloat ambient[], GLfloat diffuse[], GLfloat specular[],
                        GLfloat shininess[], GLfloat emission[])
{
    glMaterialfv(pname,GL_AMBIENT,ambient) ;
    glMaterialfv(pname,GL_DIFFUSE,diffuse) ;
    glMaterialfv(pname,GL_SPECULAR,specular) ;
    glMaterialfv(pname,GL_SHININESS,shininess) ;
    glMaterialfv(pname,GL_EMISSION,emission) ;
}
```

Example 4. 12 The Set_Light_ADS function

```
void Set_Light_ADS(GLenum light, GLfloat ambient[], GLfloat diffuse[],
                GLfloat specular[])
```

Copyright (C) Fotis Chatzinikos 1999

```
{
    glLightfv(light, GL_AMBIENT, ambient) ;
    glLightfv(light, GL_DIFFUSE, diffuse) ;
    glLightfv(light, GL_SPECULAR, specular) ;
}
```

The function **Set_Material** accepts six arguments. The first one, *pname* is used to specify the face to which the material is going to be applied. The other five arguments are arrays that contain the values that are going to be used in order to set the material up. This function uses the previously described routine **glMaterial** in order to set the various material components.

The function **Set_Light_ADS** (ADS stands for ambient, diffuse and specular) is similar to the function **Set_Material**. This time the function accepts four arguments. The first one is the light that is going to be set (i.e. LIGHT0) and the other three are arrays that contain the red, green and blue values of the diffuse, specular and ambient components of the light.

These arrays (containing the material and light components) are set inside the function keyboard. This function provides the needed keyboard interaction. The user can now manipulate the components and their elements by pressing several keys. Example 4.13 contains part of this function.

Example 4. 13 Part of the material−lights program keyboard function

```
void keyboard (unsigned char key, int x, int y)
{
   static float steping = 0.05;
   switch(key)
   {
      case 'Q' :
         Increase(MATERIAL,AMBIENT,RED,steping) ;
         glutPostRedisplay() ;
         break ;
      case 'q' :
         Decrease(MATERIAL,AMBIENT,RED,steping) ;
         glutPostRedisplay() ;
         break ;
```

As seen in the example the key ‘Q’ is used to increase the red element of the ambient component of the material by an amount equal to *steping*. ‘q’ is used to decrease the particular element by an amount equal to *steping.* The functions Increase and Decrease used here are two functions that increase or decrease the particular element of the particular component by an amount *steping*, making sure that the value of the element will not be greater than 1.0 or less than 0.0.

Plate 4.8 contains the program’s window when initially run. The user can manipulate the various components by using the keys shown in Table 4.2. A function was also created in this program that is able to save the current material and light configuration in a file, for later reference.

The user can now ‘play’ with the material and light properties in order to understand how these can be combined to produce the needed colours, materials and effects.

This program can also be used to create a particular colour and then save it to the disk. For example, if a ‘golden’ colour is needed for a particular object in another program, a programmer can experiment with this program until the needed ‘golden’ colour is approximated and then he can save it and uses it in the other program.

Copyright (C) Fotis Chatzinikos 1999

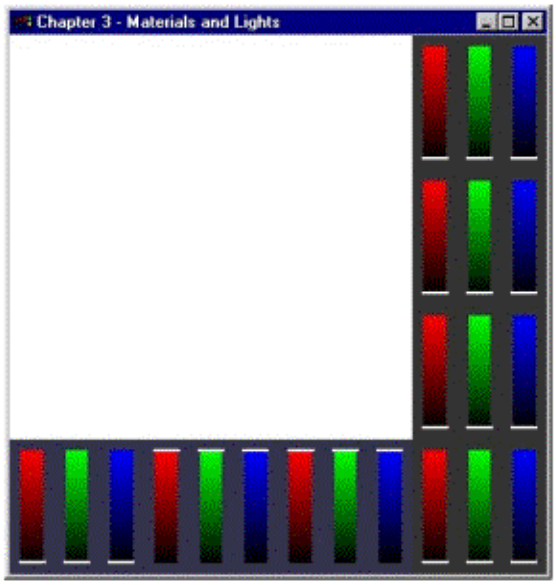


Plate 4.8 The initial window of the material-lights program

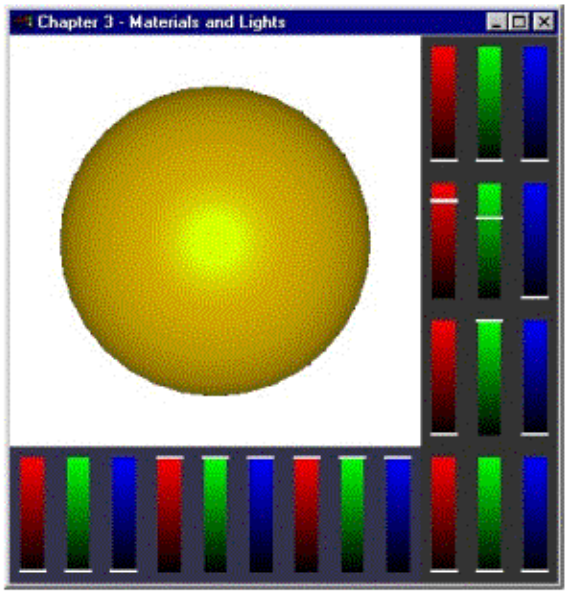


Plate 4.9 "Golden" colour in material-lights

Plate 4.9 contains the window of the program after a user has created a 'golden' colour. The graphs on the right and lower part of the screen show the current values of the red, green and blue elements of the material and light components. The four graphs on the right part of the screen show the material components (from top to bottom) ambient. diffuse, specular and emission and the three graphs on the lower part of the screen (left to right) the light components ambient, diffuse and specular.

Copyright (C) Fotis Chatzinikos 1999

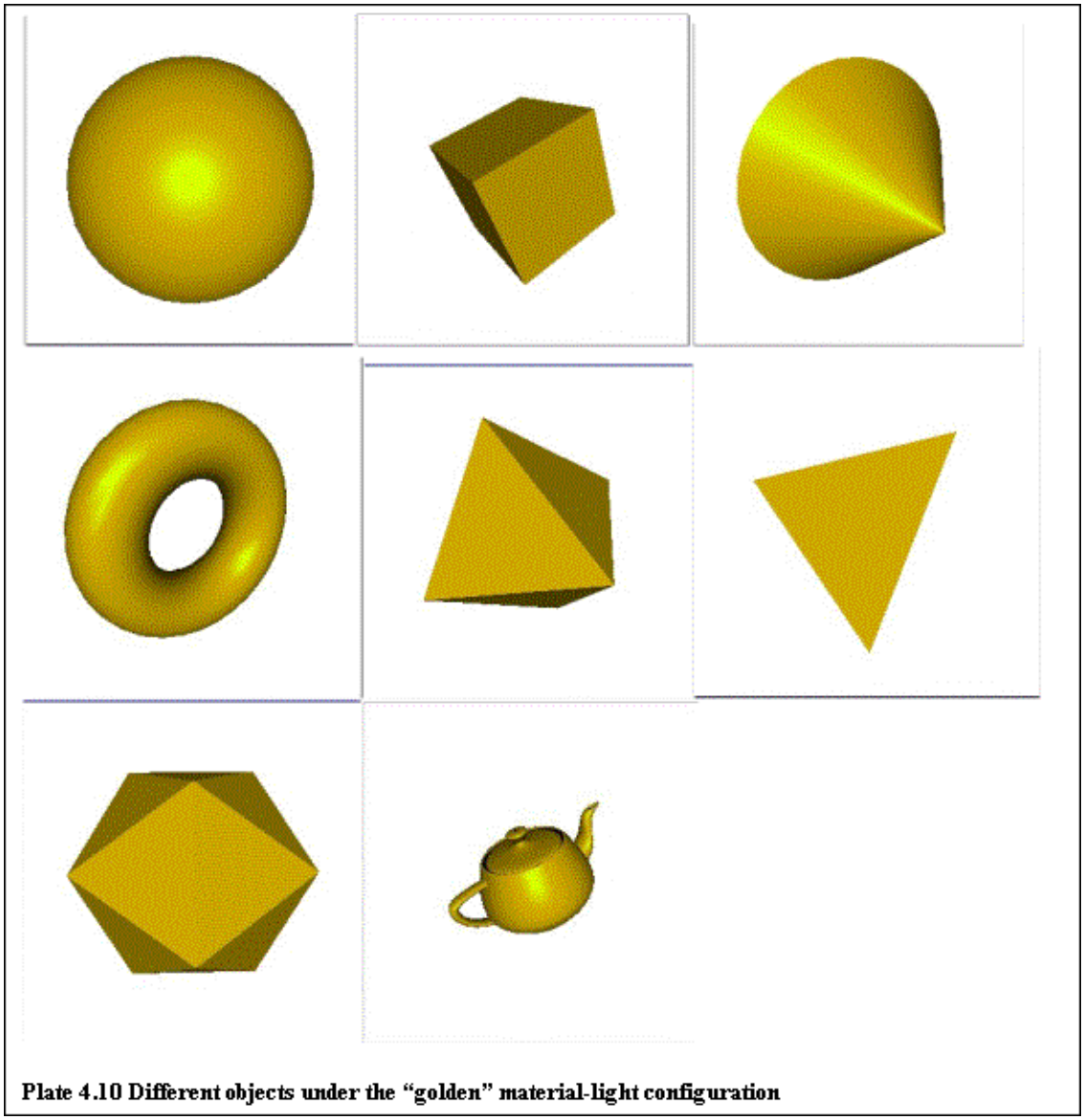


Plate 4.10 Different objects under the "golden" material-light configuration

Plate 4.10 shows how deferent objects appear under the same material−light configuration.

Copyright (C) Fotis Chatzinikos 1999

keys and their associated action

Key	Material (M) or Light(L)	Component	Element	Action
Q	M	Ambient	Red	Increase
q	M	Ambient	Red	Decrease
A	M	Ambient	Green	Increase
a	M	Ambient	Green	Decrease
Z	M	Ambient	Blue	Increase
z	M	Ambient	Blue	Decrease
W	M	Diffuse	Red	Increase
w	M	Diffuse	Red	Decrease
S	M	Diffuse	Green	Increase
s	M	Diffuse	Green	Decrease
X	M	Diffuse	Blue	Increase
x	M	Diffuse	Blue	Decrease
E	M	Specular	Red	Increase
e	M	Specular	Red	Decrease
D	M	Specular	Green	Increase
d	M	Specular	Green	Decrease
C	M	Specular	Blue	Increase
c	M	Specular	Blue	Decrease
R	M	Emission	Red	Increase
r	M	Emission	Red	Decrease
F	M	Emission	Green	Increase
f	M	Emission	Green	Decrease
V	M	Emission	Blue	Increase
v	M	Emission	Blue	Decrease
$	M	Shininess	-	Increase
4	M	Shininess	-	Decrease
P	L	Ambient	Red	Increase
p	L	Ambient	Red	Decrease
L	L	Ambient	Green	Increase
l	L	Ambient	Green	Decrease
<	L	Ambient	Blue	Increase
,	L	Ambient	Blue	Decrease
{	L	Diffuse	Red	Increase
[	L	Diffuse	Red	Decrease
:	L	Diffuse	Green	Increase
;	L	Diffuse	Green	Decrease
>	L	Diffuse	Blue	Increase
.	L	Diffuse	Blue	Decrease
}	L	Specular	Red	Increase
]	L	Specular	Red	Decrease
“	L	Specular	Green	Increase
‘	L	Specular	Green	Decrease
?	L	Specular	Blue	Increase
/	L	Specular	Blue	Decrease

Table 4.2a

Copyright (C) Fotis Chatzinikos 1999

keys and their associated action (continued)

1	Rotate object
2	Animate object
3	Cycle through objects
5	Save material-light configuration
6	Reset material
7	Reset Light
F1 to F8	Choose object
ESC	Exit Program

Table 4.2b

4.5 Adding lights to the basic model

This section is based on the program described in the second chapter, second section. This program is also data−driven, meaning that all its data are read from files.

For this reason two functions were used, named **Read_Data_From_File** and **Read_Material_From_File**. The first one (as described in Chapter 2) opens a file and reads the walking animation angles, the second one reads the body's materials. These functions will not be described here, as they contain only standard C calls in order to open a file and read some values.

The function **Read_Data_From_File** stores the data it reads into two variables of type *anim_angles* while the function **Read_Material_From_File** stores its data into variables of type *body_material*. The anim_angles structure was described back in Chapter 2, the custom type *body_material* is shown in Example 4.14.

Example 4. 14 The custom body_material

```
typedef struct
{
   float head[4][4] ;
   float head_j[4][4] ;
   float upbody[4][4] ;
   float lobody[4][4] ;
   float uparm_j[2][4][4] ;
   float uparm[2][4][4] ;
   float loarm_j[2][4][4] ;
   float loarm[2][4][4] ;
   float hand[2][4][4] ;
   float upleg_j[2][4][4] ;
   float upleg[2][4][4] ;
   float loleg_j[2][4][4] ;
   float loleg[2][4][4] ;
   float foot_j[2][4][4] ;
   float foot[2][4][4] ;
} body_materials ;
```

As seen in the example this structure is composed of floating point arrays. Body parts that appear twice in the body (like legs and arms) are arrays of dimension [2][4][4]. This array has these dimensions because a material has four components, diffuse, specular, ambient and emission ([2]**[4]**[4]), each component has four elements, red, green, blue and alpha ([2][4]**[4]**) and the body has two of the particular parts, left and right (**[2]**[[4][4]). Parts that appear only once (like the head for example) are simply arrays of type [4][4].

Copyright (C) Fotis Chatzinikos 1999

Now that materials and angles are read and available to the program and the animation functions exist from the previously constructed program only the functions that draw the lit model remain to construct.

This is actually quite easy, as the only action needed to be taken is slightly modify the already ready modelling functions (constructed in the second section of the second chapter).

For this reason the previously constructed (previous section) function **Set_Material** is going to be used. Example 4.15 contains the **Draw_Head** function constructed back in the second chapter. This function does not contain the appropriate for lighting use routines, so a new function **Draw_Head** is going to be constructed containing a call to the function **Set_Material** in the appropriate point. Example 4.16 contains the new **Draw_Head** function.

Example 4. 15 The old Draw_Head function

```
void Draw_Head(int frame)
{
   glPushMatrix() ;
   glPushMatrix() ;
   glScalef(HEAD_WIDTH,HEAD_HEIGHT, TORSO) ;
   glColor3f(0.0,0.0,1.0) ;
   if (frame == WIRE)
      glutWireCube(1.0) ;
   else
      glutSolidCube(1.0)
   glPopMatrix() ;
   glTranslatef(0.0,-HEAD_HEIGHT * 0.66,0.0) ;
   glPushMatrix() ;
   glScalef(HEAD_JOINT_SIZE,HEAD_JOINT_SIZE,HEAD_JOINT_SIZE) ;
   glColor3f(0.0,1.0,0.0) ;
   if (frame == WIRE)
      glutWireSphere(1.0,8,8);
   else
      glutSolidSphere(1.0,8,8);
   glPopMatrix() ;
   glPopMatrix() ;
}
```

Example 4. 16 The new Draw_Head function

```
void Draw_Head(int frame)
{
   glPushMatrix() ;
   glPushMatrix() ;
   glScalef(HEAD_WIDTH,HEAD_HEIGHT, TORSO) ;
   if (frame == WIRE)
   {
      glColor3f(0.0,0.0,1.0) ;
      glutWireCube(1.0) ;


   else
   {

     Set_Material(GL_FRONT,material.head[0], material.head[1], material.head[2],
                  mat_shine,material.head[3]);
      glutSolidCube(1.0) ;
   }
   glPopMatrix() ;
   glTranslatef(0.0,-HEAD_HEIGHT * 0.66,0.0) ;
   glPushMatrix() ;
   glScalef(HEAD_JOINT_SIZE,HEAD_JOINT_SIZE,HEAD_JOINT_SIZE) ;
```

Copyright (C) Fotis Chatzinikos 1999

```
   if (frame == WIRE)
   {
      glColor3f(0.0,1.0,0.0) ;
      glutWireSphere(1.0,8,8) ;
   }
   else
   {
      Set_Material(GL_FRONT,material.head_j[0], material.head_j[1], material.head_j[2],
                   mat_shine,material.head_j[3]) ;
      glutSolidSphere(1.0,8,8) ;
   }
   glPopMatrix() ;
   glPopMatrix() ;
}
```

As seen in the example only a small, easy to identify part of the code needs to be changed (hi−lighted in yellow). After applying these changes to all the modelling functions the program is ready to be compiled and run.

A new OpenGL feature is also used in this example in order to cut−down execution time. In the **init** function after the usual by now calls to **glEnable** with parameters GL_LIGHTING, GL_LIGHT0 and GL_DEPTH_TEST a new call can be found; that is **glEnable**(GL_CALL_FACE). When this value (GL_CALL_FACE) is passed to the routine **glEnable**, the OpenGL feature culling is enabled.

When culling is used all the back faced polygons are ‘removed’, meaning that no calculations are done concerning them and that they will not appear on the screen. That has the effect of cutting down to half the polygons a model is using, something that can dramatically increase the speed in cases of millions of polygons.

As the remaining of the program remains the same with the program discussed back in the second section of the second chapter, this program (and Chapter) can be considered finished.

After compiling and running this programs the results are the ones shown in Plate 4.11.

Copyright (C) Fotis Chatzinikos 1999

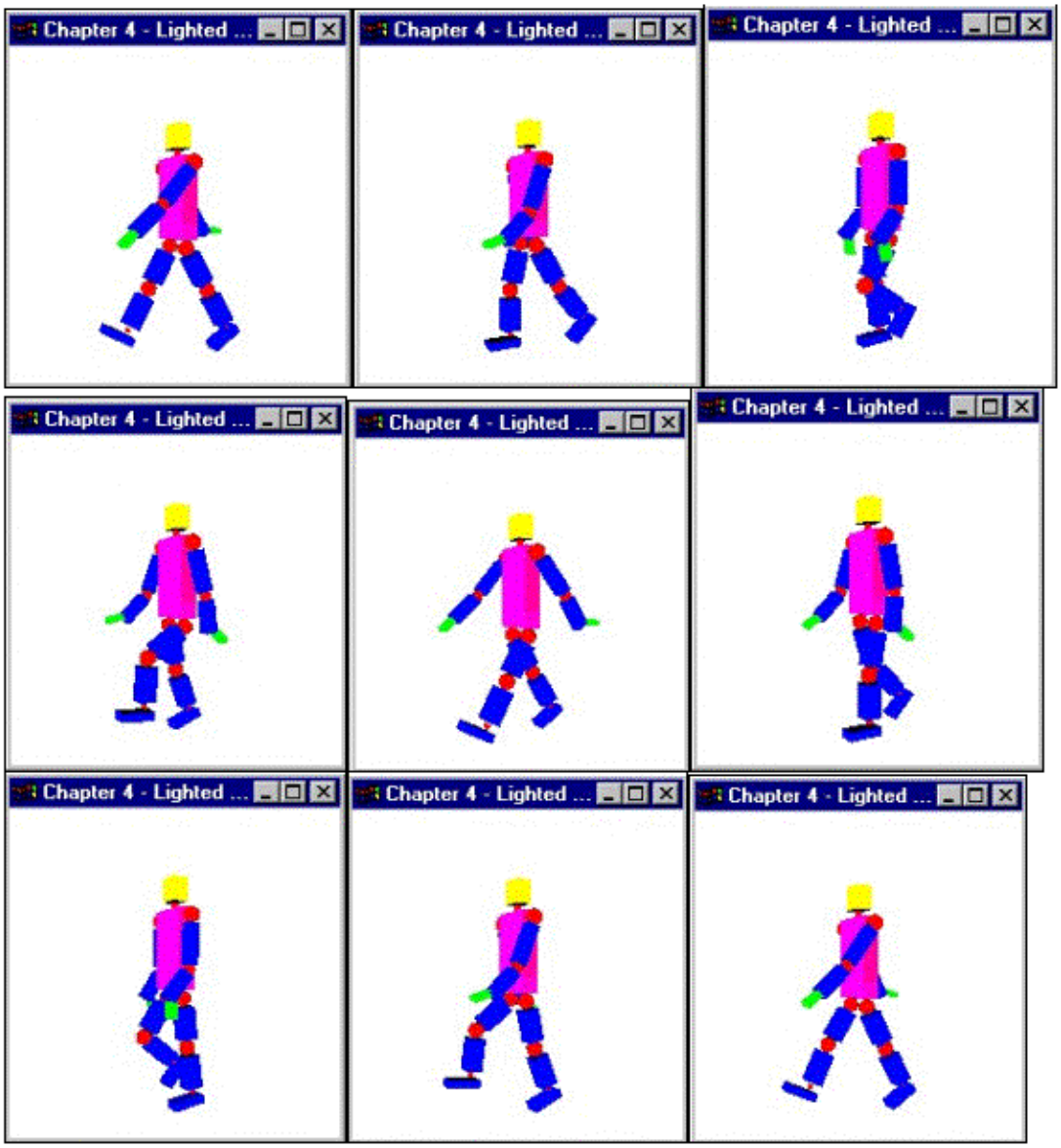


Plate 4.11 Screen-shots from the lit, animated, model

Copyright (C) Fotis Chatzinikos 1999

Chapter 5 – Improving the model: “A more elaborate geometrical example

The topic of this chapter is the improvement of the basic model (constructed from spheres and cubes) discussed in chapter 2. Such a model may be good enough for demonstrating basic OpenGL concepts but it is not good enough for a commercial application that needs a model that approximates the human body, like a game, a virtual reality application, etc.

The data set for a human body was needed in order to construct a more elaborate example. Such data sets are available through the Internet; some of them free of charge, some not. Such data sets are normally constructed by scanning a three−dimensional object (in this case a human body). As three−dimensional scanners are quite expensive and there is always the possibility of not being able to find a ready−made model, it was decided that for this project a good modeling exercise would be to create the model’s data set from scratch.

A technique had to be devised that would be able to create this three−dimensional model. In the first section of this chapter this technique is discussed, while the second part of the chapter contains a discussion of the program that reads the data set created in the first section and produces a three−dimensional model of a man.

After some research on the subject it was decided that a good technique was to try and analyze some conventional, two−dimensional photographs of a human body and try to retrieve the underlying three−dimensional model. A few anatomy books were consulted to find an appropriate model, and after some thought the model depicted in Plate 5.1 was chosen. Initially the model was not so clear, as in the process of photocopying and scanning, some noise was introduced to the image. This was corrected by applying the Paint Shop filter named find contour, resulting in the three images shown in Plate 5.1 (front, back and side view).

Copyright (C) Fotis Chatzinikos 1999

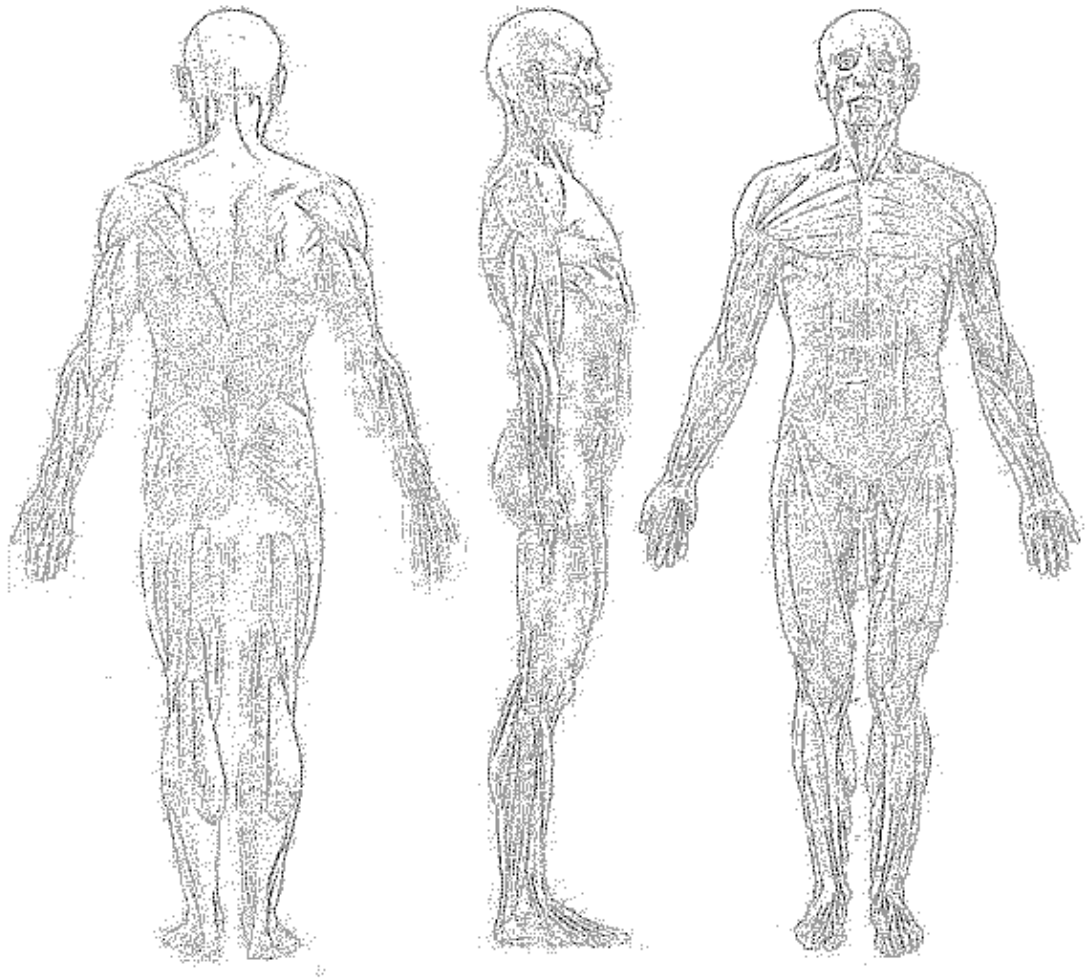


Plate 5.1 The anatomy model used to create the model’s data set

Each one of these three images contains two−dimensional information about the model. By using two images in conjunction, three−dimensional information can be created. The first step was to design some reference axes. Plate 5.2 shows the torso and the appropriate axes that were assigned to it.

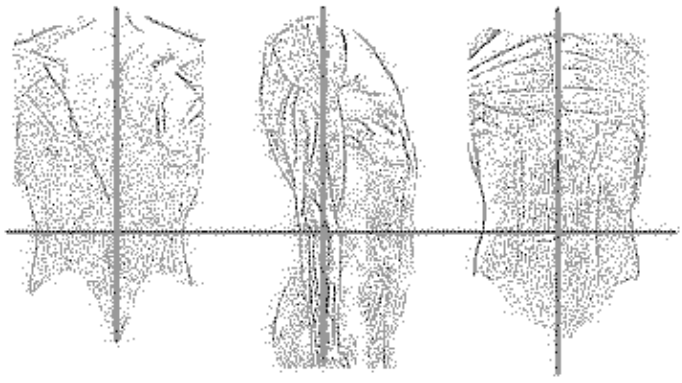


Plate 5.2 The torso and the assigned to it axes

Copyright (C) Fotis Chatzinikos 1999

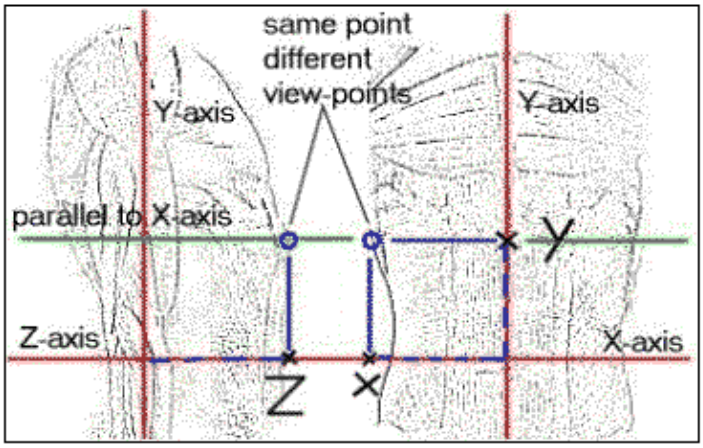


Plate 5.3 Creating a three-dimensional point from two related two-dimensional points

At this point, it is quite easy to retrieve the data sets of the front and back part of torso (in two dimensions, x and y). By making lines parallel to the x−axis, the third dimension (depth, z) can be found for every single point. Plate 5.3 shows how to find the third dimension (z) of a random (x,y) point.

Applying this technique to an appropriate number of points can result in a very good quality, three−dimensional data set of the torso. If the technique is used on all body parts, the three−dimensional data set of a man model, will be constructed. This data set is going to be used in the next section of this chapter to create the OpenGL based model. Also, as seen in Plates 5.2 and 5.3 some parts of the body, like the torso have a degree of symmetry, so only half the points are needed (as the other half can be created by mirroring).

The goal of this section was not to create the full data set of a human model, but to show that the particular technique is working. As time was short, it was decided that it would be better to go on with the following parts of the projects than spending time finishing the data set of the model. In order to have enough data for the next section (the creation of the model) the points of the neck, torso and legs were retrieved.

5.2 Creating the complex model

The goal of this section is to use the points retrieved in the previous section to create a better−looking model of a man. As the body points are saved in a file some functions were created in order to load and store these points. These functions read the body points and store them in a structure named *body_points*. This structure is a set of multi−dimensional arrays, one for each body part. This structure is defined in the file general.h and it can be seen in Example 5.1.

Example 5.1 The structure body_points

```
typedef struct
{
   float neck[2][2][10][3] ;
   float torso[2][2][23][3] ;
   float upper_leg[2][2][2][23][3] ;
   float lower_leg[2][2][2][18][3] ;
} body_points ;
```

As mention in section 5.1 only the neck, torso and legs are going to be created in this section, so the custom type *body_points* contains an array for each one of these four parts. Starting from right to left, the first dimension is used to store the three−dimensional elements of the points (x, y, z), the second dimension to store the actual points (1st, 2nd, 3rd and so on), the third dimension is used to distinguish between left and right side of a body part, the fourth to distinguish between front and back side of a body part, and when a fifth dimension is present is used to distinguish between left and right body parts (for example legs or arms).

Copyright (C) Fotis Chatzinikos 1999

Now that the structure were the points are going to be stored is explained, it is the time to discuss the functions that will read the points from the file.

These functions are implemented in the file named inout.c and the first one to be discussed is going to be the function **Return_Directory**. This function is going to be used in order to find the directory from which the program was executed so any needed files can be loaded from the same directory. This function accepts only one argument, a pointer to a character string. In every C/C++ program the first element of the *argv* array of the **main** function contains the directory from which the program was executed, including the name of the program (i.e. d:\programs\model\my_program.exe). The function **Return_Directory** takes this string and traverses it from right to left, until it finds a ‘\’ character, then it returns the rest of the string, as the remaining part is the directory from which the program was executed. This information will be used later from other function to read any needed data.

The function **Read_Body_Points_From_File** does what its name suggests. This function accepts two arguments, the first one is the directory where the file should be and the second is a pointer to a *body_points* structure where the points will be stored. The directory of the file is found at run time by using the previously discussed function **Return_Directory** (the file should be at the same directory the program was executed).

As mentioned in the first section of this chapter not all the points were digitised only the key ones. Points that could be calculated by mirroring other points were not digitised. In the body of this function, after reading the digitised points from the file, the functions **Mirror_Data_Neck_Torso**, **Mirror_Data_Upper_Leg** and **Mirror_Data_Lower_Leg** are used to create the rest of the body points. As the neck and the torso have y−axis symmetry, the function **Mirror_Data_Neck_Torso** calculates the left−hand side points of both the front and back side of the torso by mirroring the right−hand side (digitised) points. The leg does not have y−axis symmetry between its left and right sides but it has y−axis symmetry as the whole part (the left leg is the mirror image of the right leg). So the functions **Mirror_Data_Upper_Leg** and **Mirror_Data_Lower_Leg** instead of mirroring points inside the leg, they are used to calculate the points of the left leg by inverting the points of the right leg.

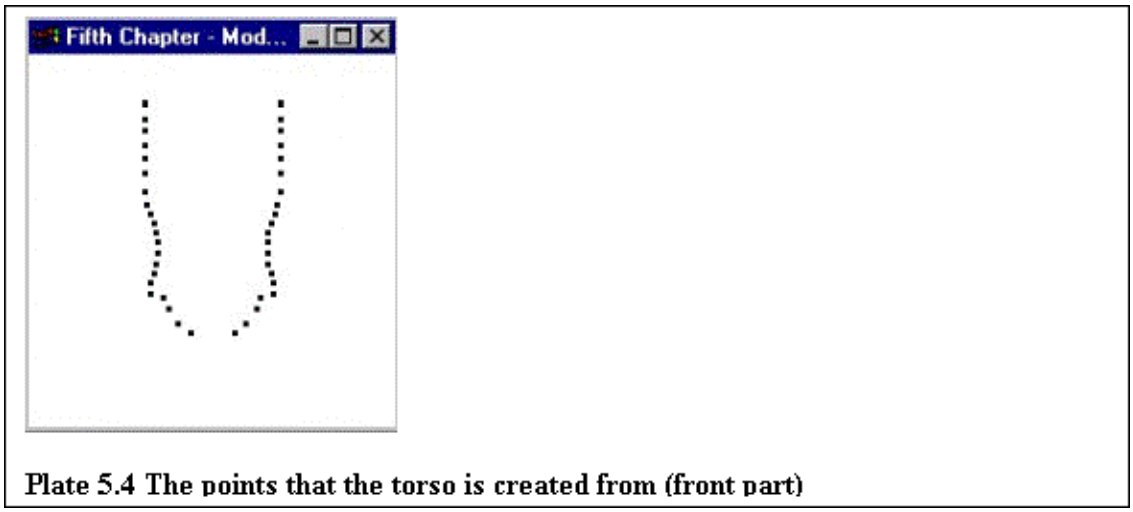


Plate 5.4 The points that the torso is created from (front part)

Now that the points are read and available to the program, some function have to be created that will use these points in order to create the model. Plate 5.4 contains the results of drawing the points that construct the front part of the torso. These points can be connected in a variety of ways in order to create a solid model. As 3D accelerators usually accelerate models constructed from triangles, this approach will be followed.

When OpenGL lighting is used, the normal of the surfaces must be calculated. A normal is a vector that is perpendicular to the surface at a particular point and is used to calculate how light is reflected from the surface. Until now there was no reason to calculate the normals of the objects used, as these objects were created by using the available GLUT functions that contain also the needed normals.

Copyright (C) Fotis Chatzinikos 1999

A function was created that given three points of a three dimensional area that lies on the same plane in space (these points do not lie on a straight line), can calculate the unit normal to the surface.

This function calculates the perpendicular to the plane (the normal) by using the function

[v1−v2]x[v2−v3]

where the symbol ‘x’ means the cross product. v1, v2 and v3 are the three vectors that can be created when the three supplied points are joined.

Now that the function that calculates the normal to a surface is created, it is time to create the actual surfaces. As all the functions that create the body parts are similar, it was chosen to describe only one of them, the one that creates the torso. Example 5.4 contains this function.

Example 5.4 The Draw_Torso function

```
void Draw_Torso(int frame, float torso[2][2][23][3])
{
   glPushMatrix() ;
   if (frame == WIRE)
      glColor3f(1.0,0.0,0.0) ;
   else
      Set_Material(GL_FRONT,material.torso[0], material.torso[1], material.torso[2], material.s
   create_torso_front(torso[FRONT][LEFT], torso[FRONT][RIGHT]) ;
   create_torso_back(torso[BACK][LEFT], torso[BACK][RIGHT]) ;
   create_torso_sides(torso[FRONT][LEFT], torso[FRONT][RIGHT], torso[BACK][LEFT], torso[BACK][R
   glPopMatrix() ;
}
```

As seen in the example the structure of the **Draw_Torso** function is similar to the previous **Draw_Torso** function the difference being that instead of calling the GLUT functions to create the torso the custom made functions **create_torso_front**, **create_torso_back** and **create_torso_sides** are called.

The function accepts two arguments. The first one, *frame* can take the values WIRE or SOLID and is used in order to assign a colour or set a material (depending on if the model is wireframe or solid). The second argument contains the points that will be used to create the torso. Example 5.5 contains the function **create_torso_front** the other two functions will not be discussed here, as they are similar to this one.

Example 5.5 The create_torso_front function

```
void create_torso_front(float left[23][3], float right[23][3])
{
   int counter  ;
   float normal[3] ;
   for (counter = 0 ; counter <22 ; counter++ )
   {
      Calculate_Normal( left[counter],left[counter+1],right[counter+1],normal) ;
      glBegin(GL_TRIANGLE_STRIP) ;
         glNormal3fv(normal);
         glVertex3fv(left[counter]) ;
         glVertex3fv(left[counter+1]) ;
         glVertex3fv(right[counter]) ;
         glVertex3fv(right[counter+1]) ;
      glEnd() ;
   }
}
```

Copyright (C) Fotis Chatzinikos 1999

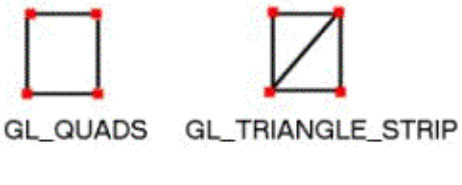


Plate 5.5 Difference between GL_QUADS and GL_TRIANGLE_STRIP

As seen in the example the body of this function is basically a **for** loop. Its time the for loop is executed, a normal is found by passing three appropriate values to the function **Calculate_Normal** and then the function **glBegin** is called with a GL_TRIANGLE_STRIP value. As explained previously the value passed to the function **glBegin** specifies what kind of object is going to be created. The value GL_TRIANGLE_STRIP is used to create triangle strips in the way shown in Plate 5.5.

The front part of the torso is constructed from 46 points. 23 of them are on the left−hand side and 23 on the right−hand side. The easiest way to create the front−torso surface using triangles is to create 22 triangle strips, each one of them constructed from 4 points. The way in which the points are selected is also important, as all triangles on the same surface should have the same orientation, in order not to have problems later when trying to use culling or a similar operation. Plate 5.6 shows the points and in which way they should be connected.

When four points are used to create a triangle strip, they should be specified in the order of: L1 à L2 à R1 à R2, as OpenGL will use vertices L1, L2 and R1 to create the first triangle and vertices R1, L2 and R2 to create the second triangle (in the exact order). In this way all triangles of a triangle strip are oriented in the same, anti−clockwise way.

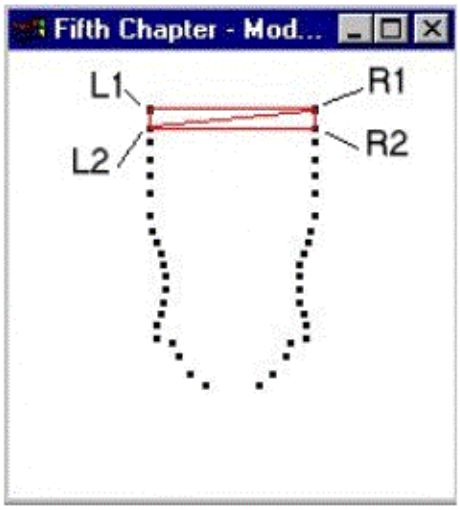


Plate 5.6 Creating triangle strips

Copyright (C) Fotis Chatzinikos 1999

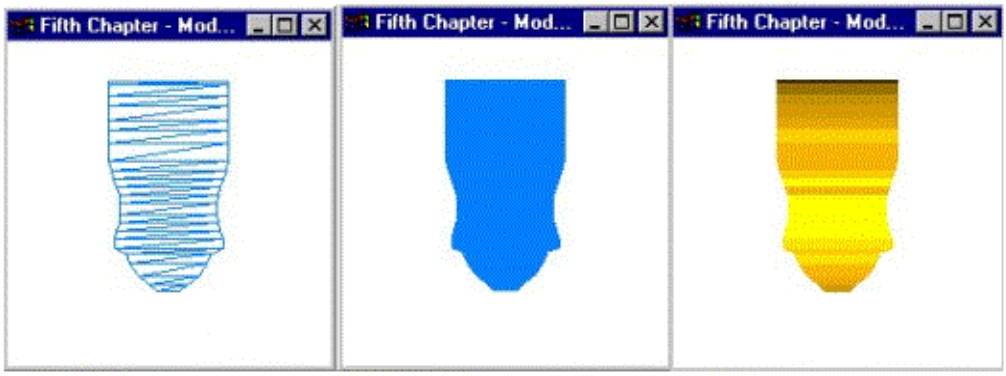


Plate 5.7 The results of the create_front_torso function (wireframe, solid, lit)

When the **for** loop finishes executing (22 times), the front part of the torso will be ready containing 22 triangle strips each one of them constructed from two triangles, giving a total of 44 triangles. The results of this function can be seen in Plate 5.7.

Using the same technique all the other modelling functions are created.

In order to animate the model a small change is needed in the **base_move** function, as the constants UP_LEG_HEIGHT, LO_LEG_HEIGHT and LEG_HEIGHT do not exist anymore. In order to solve this problem the two values UP_LEG_HEIGHT and LO_LEG_HEIGHT can be specified as externals and the value LEG_HEIGHT can be calculated (LEG_HEIGHT = UP_LEG_HEIGHT + LO_LEG_HEIGHT). The values of UP_LEG_HEIGHT and LO_LEG_HEIGHT are then calculated in the function **init**, in the main program, by finding the absolute value of the difference of the first and last points in the leg data set.

keys and their associated action

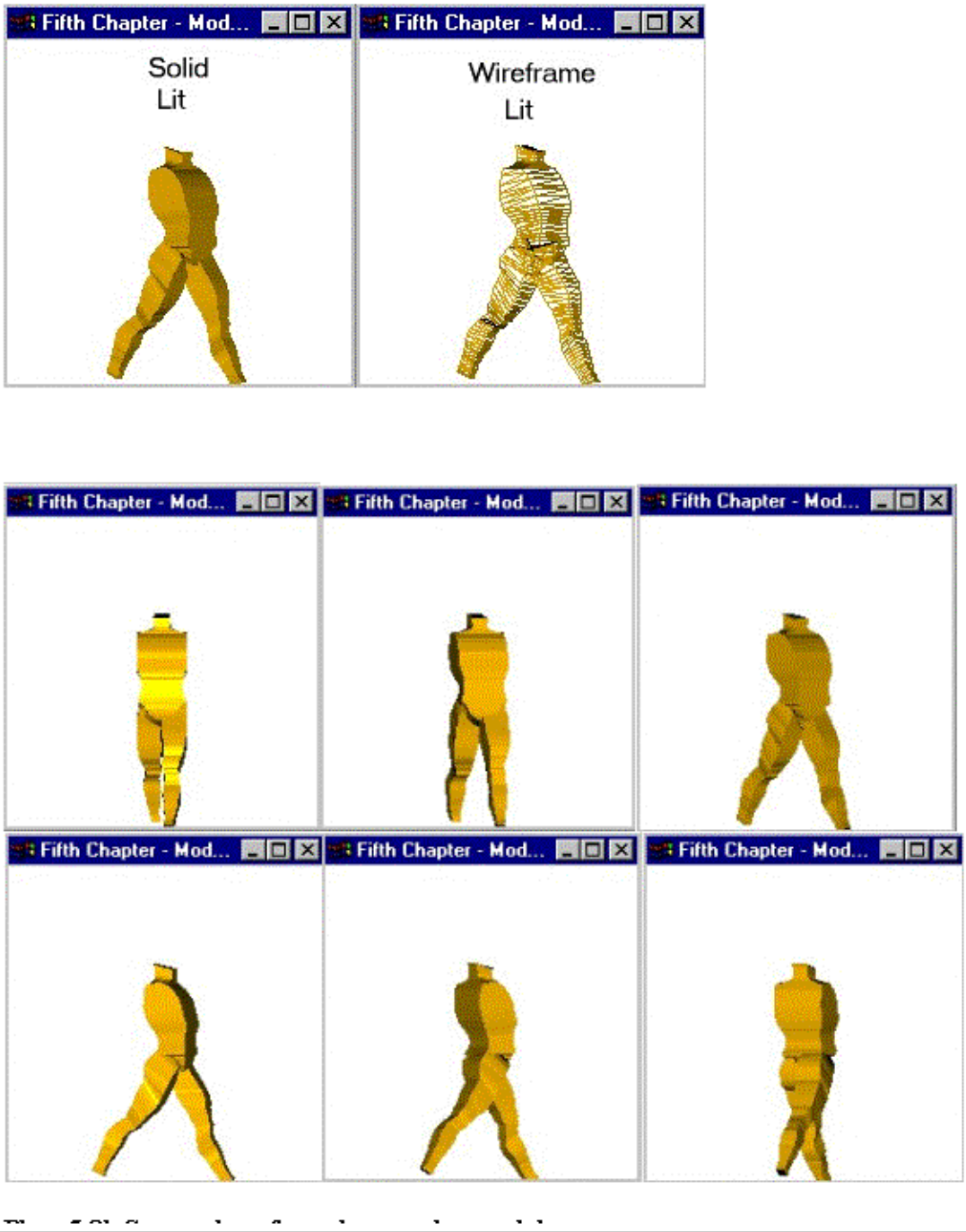
a	Rotate object left
s	Rotate object right
x	Choose solid or wireframe
c	Lights on/off
v	Culling on/off
1	Rotate in 3D
2-8	Choose body part
~	Animate model (walking)

Table 5.1

If this program is compiled and run the results will be the ones shown in Plate 5.8. Table 5.1 contains the keys used in the program and their associated operations.

Copyright (C) Fotis Chatzinikos 1999

Plate 5.8a Screenshots from the complex model program



Copyright (C) Fotis Chatzinikos 1999

Chapter 6 – Texture Mapping

Until now, every geometric primitive has been drawn as either a solid color or smoothly shaded by interpolating the colors of its vertices. Texture mapping allows images to be glued on polygons and then follow the polygon's transformations.

With texture mapping, any image (scanned, drawn, etc.) can be applied onto a polygon, giving the polygon a completely different touch. Texture mapping ensures that acceptable things will happen to the image when the underlying polygon undergoes any transformations. For example if perspective projection is used in the scene any images used as textures will appear smaller as they get further from the viewpoint.

Texture mapping has many applications, some of them being wallpaper patterns, ground images in flight simulators, textures that make polygons look like natural substances such as marble, wood and so on.

Textures are simply rectangular arrays of data. The individual values in the texture array are often called texels. What makes texture mapping tricky is that rectangular textures can be mapped to non−rectangular regions, and this must be done in a reasonable way.

As texture mapping is such a large area, this chapter will not try to discuss the whole subject but explain the basics of texture mapping, how texture mapping is used in OpenGL and some of the basic filters that can be applied to textures.

As the application/example that was constructed for this Chapter is quite a lot more complicated than any of the previously discussed ones, its development will be divided into several sections.

Section one will discuss the functions needed in order to open and display a windows bitmap image (bmp) file. This image file was chosen because it does not use any compression when saving the image, so it is relatively easy to open and load the image.

As this program uses several windows, section two will discuss what has to be done in order to open and manage more than one window. Some functions of the Fast Light Tool Kit library are also explained in this section, as they are used in order to create the programs interface (buttons, pull−down menus etc.).

Section three continues and describes how a texture is created and then applied onto a polygon. As a simple example the texture is applied onto a cube.

The last section of this chapter explains the operations needed to put everything together, as well as describing the needed changes in the human model function to incorporate texture mapping.

In order to use texture mapping, some images must be available to the program. The easiest way to load and use an image as a texture is to save the image as a bmp file. This file format is very common among Microsoft Windows platforms and all image manipulation applications are able to save in this format. When the image is saved in the particular file format some function have to be created to load the image and make any needed manipulations to its data format in order to be of use with OpenGL.

Three functions were created for this reason (based on functions found in the book "OpenGL Super bible") named **LoadBitmapMy**, **ConvertRGB** and **SaveDIBitmap**.

The first function, **LoadBitmapMy** accepts two arguments and returns a pointer of type void. The first argument is a pointer to a character string that contains the directory and filename of the image. The second argument is a pointer to a BITMAPINFO structure. Every bitmap image contains a header of that type (BITMAPINFO), so when function **LoadBitmapMy** is called, the header of the image will be stored in a variable of type BITMAPINFO and the actual bitmap bits (each pixel of the image) will be stored in a pointer of type void.

Copyright (C) Fotis Chatzinikos 1999

This function is quite complicated, but there is no need to explain in depth what happens inside it. The main point is that an attempt is made to open the bitmap image file specified in the parameter *filename*; if this operation is successful, a check follows to discover whether the file is a bitmap file; if this check succeeds, memory is allocated for the bitmap header, the bitmap header is read and if everything is correct, memory is allocated for the actual image and the image is read. If everything went fine, the image is contained in the variable *bits* (of type void pointer) which is returned from the function.

The function **SaveDIBitmap** is similar to this one, as instead of opening a file and reading a bitmap into a void pointer variable, it receives an image in the form of a void pointer variable and saves it to the disk.

A problem with bitmap files is that the colour values are not saved in the order used by *OpenGL* (R−G−B), but in the order Blue, Green, Red. A function was needed that would swap the red and blue values of the image and that is the purpose of the **ConvertRGB** function.

At this point, after calling the two functions **ReadBitmapMy**, and **ConvertRGB** a bitmap image is available to the program (in the form of a void pointer variable). Now some appropriate OpenGL calls are needed in order to display this image in a window.

First of all, and in order to display the image correctly without any distortions, the projection used must be orthographic and the lower−left corner must be at (0,0) while the upper−right corner at (width –1, height –1) where width and height are the width and height of the image. This projection makes sure that the image will be displayed ‘as−is’ in a one−to−one way. Example 6.1 contains the **reshape** function that does that.

Example 6.1 The reshape function used to display an image

```
void reshape_main(int w,int h)
{
   glViewport(0,0,w,h) ;
   glMatrixMode(GL_PROJECTION) ;
   glLoadIdentity() ;
   glOrtho(0, width−1,0, height−1, −1,1) ;
   glMatrixMode(GL_MODELVIEW) ;
   glLoadIdentity() ;
}
```

Now that the projection is set, is the time to do the actual drawing. The display function that is used to draw the image on the screen is shown in example 6.2.

Example 6.2 The display function used to display a bitmap image

```
void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT) ;

   if (BitmapBits != NULL)
   {
      glPixelStorei(GL_UNPACK_ALIGNMENT, 4);
      glRasterPos2i(0,0);
      glPushMatrix() ;
      glDrawPixels(BitmapInfo−>bmiHeader.biWidth ,
                   BitmapInfo−>bmiHeader.biHeight,
                   GL_RGB, GL_UNSIGNED_BYTE, BitmapBits);
      glPopMatrix() ;
   }
   glutSwapBuffers() ;
}
```

As seen in the example after the colour buffer is cleared by calling the function **glClear**, a check is done to see whether the variable *BitmapBits* (used to store the image) contains any data.

Copyright (C) Fotis Chatzinikos 1999

If the variable is empty (NULL), nothing is done, otherwise the function **glPixelStorei** is called with the arguments (GL_UNPACK_ALIGNMENT, 4). This call specifies how data are going to be ‘unpacked’ from memory in order to draw it, in this case, by the function **glDrawPixels**. As the image is represented as a linked list of values (Red, Green, Blue, Alpha, etc.) OpenGL needs to know how to ‘unpack’ this data, meaning that the programmer should specify that for example every pixel is represented in the linked list by four values (R−G−B−A).

When this is done, a call to **glRasterPos** follows in order to set the current raster position.

As the orthographic projection used has its lower−left corner at point (0,0), the current raster position is set to the lower−left corner of the screen. This is needed because the function **glDrawPixels** starts drawing the lower−left part of the image at the current raster position and incrementally to the top−right.

glPixelStore parameters

Parameter Name	Type	Initial Value	Valid Range
GL_UNPACK_SWAP_BYTES, GLPACK_SWAP_BUFFERS	GLboolean	FALSE	TRUE\FALSE
GL_UNPACK_LSB_FIRST, GL_PACK_LSB_FIRST	GLboolean	FALSE	TRUE\FALSE
GL_UNPACK_ROW_LENGTH, GL_PACK_ROW_LENGTH	GLint	0	any nonnegative integer
GL_UNPACK_SKIP_ROWS, GL_PACK_SKIP_ROWS	GLint	0	any nonnegative integer
GL_UNPACK_SHIP_PIXELS, GL_PACK_SKIP_PIXELS	GLint	0	any nonnegative integer
GL_UNPACK_ALIGNMENT, GL_PACK_ALIGNMENT		4	1,2,4,8

After the call to **glRasterPos** the current matrix is saved, the function glDrawPixels is called in order to draw the image and the matrix is restored.

The function **glDrawPixels** accepts five parameters. The first one is the width of the image to be drawn and the second is the height. The third parameter indicates the kind of pixel data elements to be used (Table 6.2) and the fourth one the type of each element (Table 6.3). The fifth parameter is a pointer to an array that contains the pixel data to be drawn.

As seen in the example, the width and height of the image are passed to the function by using the bitmaps header. The format is set to R−G−B mode and the type to unsigned bytes. The array that contains the pixel data is the previously mentioned array *BitmapBits*.

The bitmap’s header information (the width and the height) is also used in order to resize the window that the bitmap is drawn, in order to be the same size.

Copyright (C) Fotis Chatzinikos 1999

GL_COLOR_INDEX	A single colour index, unsigned 8-bit integer
GL_RGB	A red colour component, followed by a green, followed by a blue
GL_BGR	Same as GL_RGB, but the blue component is first
GL_RED	A single red colour component
GL_UNSIGNED_SHORT	A single green colour component, unsigned 16-bit integer
GL_SHORT	A single blue colour component, signed 16-bit integer
GL_UNSIGNED_INT	A single alpha colour component, unsigned 32-bit integer
GL_INT	A single luminance component, signed 32-bit integer
GL_LUMINANCE_ALPHA	A luminance component followed by an alpha colour component
GL_STENCIL_INDEX	A single stencil index
GL_DEPTH_COMPONENT	A single depth component

Plate 6.1 contains some screenshots from images loaded using this program.

Plate 6.1 Image shown by the read and display bitmap program

6.2 Opening several windows with OpenGL

This example will use several windows. One will be used for displaying the bitmap image from which the texture will be created (this window was the subject of the previous section). Another window will be used to show the texture (when this is created) and two more windows will be used to demonstrate texture mapping. One will be used for displaying a texture mapped cube and a second one to display the texture mapped model of a man.

The technique used to create the four windows is the same one employed in section 4, chapter 4. Example 6.3 shows part of the main function that creates the four windows and assigns to them any needed callback functions.

Example 6.3 Part of the main function that creates four windows

```
main_win = glutCreateWindow("Sixth Chapter – Texture Mapping") ;
glutHideWindow() ;
create_panel(argc,argv) ;
init() ;

glutDisplayFunc(display) ;
glutReshapeFunc(reshape_main) ;
glutKeyboardFunc(keyboard) ;
glutMouseFunc(mouse) ;
```

Copyright (C) Fotis Chatzinikos 1999

```
glutPassiveMotionFunc(passive_motion) ;
glutInitWindowSize(box_width,box_height) ;
glutInitWindowPosition(0,0) ;

texture_win = glutCreateWindow("Texture") ;
glutDisplayFunc(display2) ;
glutHideWindow() ;
init_cube_win() ;
glutInitWindowSize(200,200) ;
glutInitWindowPosition(0,100) ;

cube_win = glutCreateWindow("Distorted Cube Window") ;
glutDisplayFunc(display_cube) ;
glutReshapeFunc(reshape_cube) ;
glutSpecialFunc(special) ;
glutKeyboardFunc(keyboard) ;
glutHideWindow() ;
init_torso_win() ;

glutInitWindowSize(200,200) ;
glutInitWindowPosition(0,300) ;

torso_win = glutCreateWindow("Torso Window") ;
glutDisplayFunc(display_torso) ;
glutReshapeFunc(reshape_torso) ;
glutSpecialFunc(special) ;
glutKeyboardFunc(keyboard) ;
glutHideWindow() ;
```

As seen in the example, after the creation of every window (by calling the function **glutCreateWindow**) a call to the function **glutHideWindow** is issued. This GLUT function is responsible for hiding the current window, meaning that the window is created, but it is not visible to the user. This was done, because it was decided that no other windows other than the main interaction window should be visible to the user when the program is firstly run.

The main interaction window is created by calling the custom function **create_panel**. This function contains the needed Fast Light Tool Kit (FLTK) routine calls to create a FLTK window, containing some buttons and pull−down menus.

The problem with FLTK is that it is written in C++, so a C++ syntax must be used. After some thought it was decided that it would be overcomplicated to try and discuss the FLTK calls that are needed to create the user interface window, as the reader is used to the standard C conversions, so this window will be accepted ‘as is’.

As seen in the previous example each window is assigned its own **display** and **reshape** functions. This was done because each window displays different graphics and thus needs different reshape conditions.

Copyright (C) Fotis Chatzinikos 1999

Plate 6.2 The user interface window created with FLTK

A new GLUT routine also appears in this piece of code, named **glutPassiveMotion**. This routine is responsible for registering a passive motion callback function. What is meant by the term passive motion is that the mouse moves inside a window without any of its buttons pressed (active motion would therefore be the mouse does move when one or more of its buttons is pressed). This function is used in this program to animate a small selection box, when a texture is selected from a larger image.

6.3 Creating a texture

The topic of this section is the creation of a variable size texture. OpenGL textures can be of several different dimensions, depending to the implementation. Most OpenGL implementations support textures of dimensions up to 256 by 256 pixels (one−dimensional textures are possible but they are not discussed here). The size of two−dimensional textures must be a power of two (2x2, 4x4, 8x8 and so on).

In this program, the texture will be created by selecting a region of a (usually) larger image. A selection box will be rendered inside the image window (discussed in the first section of this chapter) and the user will be able to ‘lock’ the selection rectangle at some convenient to him point to create the texture. By locking it is meant that the selection rectangle will not follow the mouse movement from this point onwards.

The selection box is animated while the user moves the mouse inside the window using the callback function registered with **glutPassiveMotion**. When a convenient place is found the user can press the right mouse button to ‘lock’ the selection rectangle and then create the texture.

The texture is created by calling the function **show_texture_cb**. This function is shown in example 6.4.

Copyright (C) Fotis Chatzinikos 1999

Example 6.4 The function show_texture_cb

```
void showTexture_cb(Fl_Widget *, void *)
{
    NewBitmapBits = array ;
    glReadPixels(mouse_coX,height − mouse_coY, box_width, box_height, GL_RGB,GL_UNSIGNED_BYTE,
    glutSetWindow(texture_win) ;
    glutReshapeWindow(box_width,box_height) ;
    glutShowWindow() ;
    glutPostRedisplay() ;
    glutSetWindow(main_win) ;
}
```

Actually the texture is not created at this point but later on; what happens in this function is that the function **glReadPixels** is used to read the pixels which are under the selection area into the variable *array*. The routine **glReadPixels** has exactly the opposite effect of the previously explained routine **glDrawPixels**.

When the pixels inside the selection box are stored in the variable *array*, the routine **glutSetWindow** is used to set the texture window as the current, then the texture window is resized to the dimensions of the texture. Finally the window is shown.

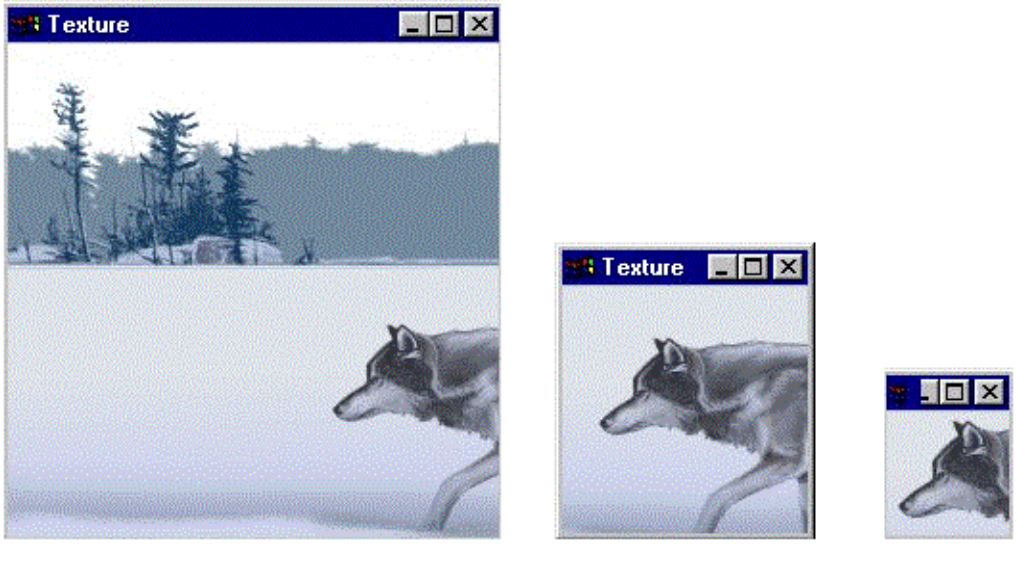


Plate 6.3 Textures of various sizes (128x128, 64x64 and 32x32)

The size of the texture created depends on the size of the selection box. The size of the selection box (and the texture's) can be set by calling the function **texturesize_cb**. The body of this function is actually a switch **statement** and each time the function is called a flag is incremented, thus cycling among the predefined texture sizes (one of the **switch** cases). Plate 6.3 contains various sizes textures, created by this method.

The actual texture, as mentioned before, is not created in the function **create_texture_cb**, but in the body of the **display** function that creates the texture mapped cube.

In the body of this function, after the usual function calls (**glClear** etc.), the routine **glEnable** is called with the value GL_TEXTURE_2D passed to it. This call enables OpenGL's texture mapping ability.

Copyright (C) Fotis Chatzinikos 1999

possible combination of values for the routine glTexParameter

GL_TEXTURE_WRAP_S	GL_CLAMP, GL_REPEAT
GL_TEXTURE_WRAP_T	GL_CLAMP, GL_REPEAT
GL_TEXTURE_MAG_FILTER	GL_NEAREST, GL_LINEAR
GL_TEXTURE_MIN_FILTER	GL_NEAREST, GL_LINEAR, GL_NEAREST_MIPMAP_NEAREST, GL_NEAREST_MIPMAP_LINEAR, GL_LINEAR_MIPMAP_NEAREST, GL_LINEAR_MIPMAP_LINEAR,
GL_TEXTURE_BORDER_COLOR	any four values in the range [0.0, 1.0]
GL_TEXTURE_PRIORITY	[0.0, 1.0] for the current texture object

The next routine appearing for the first time in this function is **glTexParameter**. This routine is responsible for setting various parameters that control how a texture is treated and it accepts three arguments; the first argument can be either GL_TEXTURE_2D, or GL_TEXTURE_1D to indicate a two− or one−dimensional texture. The possible combinations of values for the second and third parameter are shown in Table 6.4. Visual examples of the effect of this function will be found near the end of this section.

Back in the body of the **display_cube** function, just after the calls to **glTexParameter,** a call to **glTexEnv** is issued. The purpose of this function is to specify how the texture colours are going to be calculated. The colour of a texture can be the colour of its own texels, or a combination of its own colour and the surface on which it is applied. Visual examples of the effects of this routine will appear at the end of this section.

Following, the routine **glTexImage2D** is called. This is the most important routine in the **display_cube** function, as it is the one that defines the actual two−dimensional texture. This function accepts quite a few arguments, therefore an example is given here to explain what each one is used for.

glTexImage2D*(GL_TEXTURE_2D, 0, GL_RGB, box_width, box_height, 0, GL_RGB, GL_UNSIGNED_BYTE, array) ;*

The first parameter can be either GL_TEXTURE_2D or GL_PROXY_TEXTURE_2D. In this report the constant GL_PROXY_TEXTURE_2D is not discussed, so GL_TEXTURE_2D is used. The second parameter is used when supplying multiple resolutions of the texture. Multiple texture resolutions are used for mip−maping, something that is not covered in this report, so this parameter is set to 0 (only one resolution).

The next parameter indicates which of the R, G, B, and Alpha components or luminance or intensity values are selected for use in describing the texels of the image. This can be one for thirty eight symbolic constants. The one used here, GL_RGB, specifies that the Red, Green and Blue components are used to describe the texel.

The next two parameters specify the width and the height of the texture, and as described before they are in this case the width and the height of the selection box. The next parameter is the border of the texture and can be either 0 (no border) or 1. Both the width and the height of the texture must have the form $2^m + 2b$, where m is a nonnegative integer and b is the width of the border. In this example no borders are used (0 is passed to the function).

The following two parameters describe the format and type of the texture image data, and they have the same meaning as in the case of **glDrawPixels** (Tables 6.2 and 6.3), with the exception of GL_STENCIL_INDEX and GL_DEPTH_COMPONENT (for the format parameter).

Finally the last parameter contains the texture−image data. These data describe the texture image itself as

Copyright (C) Fotis Chatzinikos 1999

well as the border.

At this point the texture environment, the appearance and the texture itself are constructed, set and ready to use. It will now be discussed what texture co−ordinates are and how they should be specified.

In OpenGL, textures are treated as normal objects, so it is typical to be able to set their co−ordinates. When texture mapping is used, both object and texture co−ordinates must be provided for each vertex. After transformation, the object co−ordinates determine where on screen that particular vertex is rendered. The texture co−ordinates determine which texel in the texture map is assigned to that vertex. Texture co−ordinates are interpolated between vertices in the same way colour values were.

Depending on the texture co−ordinates applied, the texture can be mapped one−to−one, inverted, stretched, shrunk, etc. Certain visual examples will appear later on to help visualise the concept.

For the moment, every vertex of the cube is assigned either a 0 or 1 texture co−ordinate (inside the **display_cube** function). This results in the (square) texture, mapped one−to−one to each (square) face of the cube.

At this point the **display** function **display_cube** that will be used to draw a texture mapped cube is ready. Plate 6.4 contains some screenshots of the texture mapped cube (the default values of environment and texture parameters are used).

Plate 6.5 demonstrates the effects of the function **glTexEnv**, as it contains screenshots with different environment settings and Plate 6.6 shows the cube under the effect of different texture mapping filters (**glTexParameter**).

The last plate in this section, Plate 6.7 shows what happens when different texture co−ordinates are used (in conjunction with the **glTexParameter** parameters GL_TEXTURE_WRAP_S and GL_TEXTURE_WRAP_T). For this reason the function **display_cube** was slightly modified, and instead of setting the texture co−ordinates inside the function, global variables are used as the texture co−ordinates, which can be changed with the keyboard's directional keys (arrows).

Plate 6.4 The texture mapped cube

In this project global variables are used in some occasions, but not as a result of 'bad programming practice' but because functions like this one (**display**) are of type **void**, meaning that they can not receive any parameters.

Copyright (C) Fotis Chatzinikos 1999

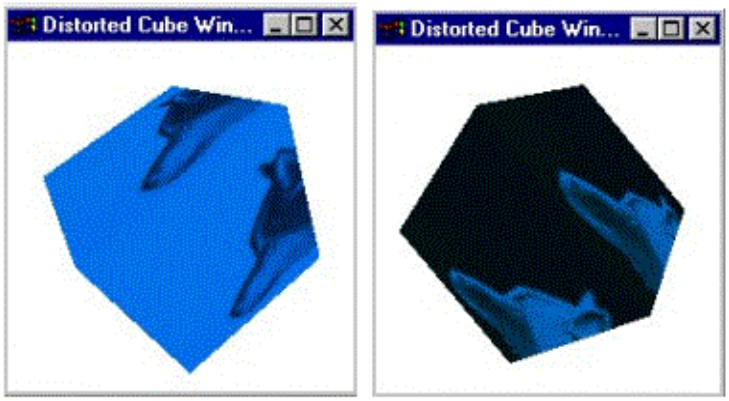


Plate 6.5 Different texture environment values (GL_MODULATE and GL_BLEND)

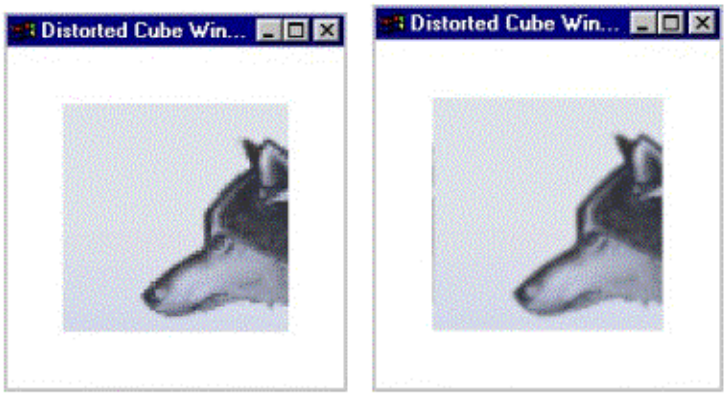


Plate 6.5 Different texture mapping filters (GL_NEAREST and GL_LINEAR)

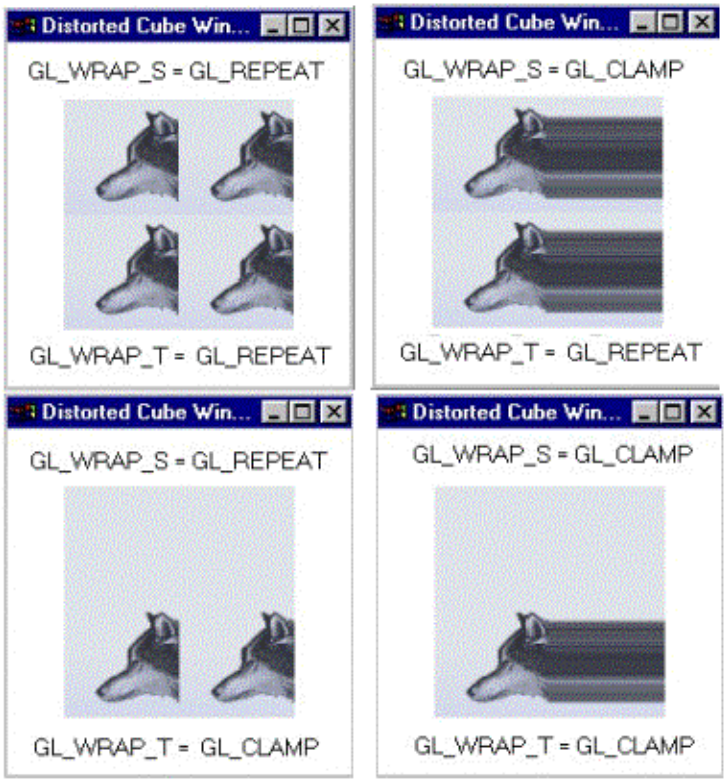


Plate 6.7 The effect of changing the texture co-ordinates in conjunction with GL_WRAP_S and GL_WRAP_T

Copyright (C) Fotis Chatzinikos 1999

6.4 A texture mapped man

The goal of this section is to apply the texture mapping techniques discussed in the previous section onto the model created in Chapter 5. For that reason the modelling functions constructed in Chapter 5 (**Draw_Leg**, **Draw_Arm**, etc.) will have to be modified to include texture co−ordinates statements.

The same texture will be applied to the whole body, so any texture construction and setting will be done in the main program. The modelling functions will only have to include the appropriate texture co−ordinates statements.

As mentioned in the previous section, every object must have its texture co−ordinates assigned appropriately, otherwise the texture will be so distorted that will be unrecognisable.

The same texture will be applied to all the parts of the body, once on each side (i.e. the texture will appear four times on the torso at its front, back, left and right side). As the body parts are not square and they are constructed from many vertices, a way must be found to calculate the texture co−ordinates for every vertex.

Every ‘side’ (front, back, left or right) is constructed from two arrays, the ‘left’ and the ‘right’ (refer to Chapter 5). The ‘horizontal’ texture co−ordinates are quite easy to assign, as no calculation is involved. If every vertex in the ’left’ array is assigned a x (horizontal) texture co−ordinate of 0 and every vertex in the ‘right’ array a x co−ordinate of 1, a nice, ‘tight clothes effect’, can be achieved (as the texture will be shrunk, the impression of clothes tight to the body will be given).

The problem appears when trying to assign the ‘vertical’ (y values) co−ordinates of the vertices. It is clear that the lower point of every part will be assigned the value 0 and the higher point the value 1, but how can the in−between values be calculated?

Every vertex in the array has three values, a x, an y and a z value. The appropriate y texture co−ordinate can be calculated by the following technique:

· the higher vertex of the body part is assigned a texture value of 1 (y value)

· the distance between this point and the next point is found

· this distance is divided by the total body part length

· the next (lower) vertex of the body part is assigned a value of 1 minus the calculated value ((point – next_point) / length)

· the technique, is repeated until all vertices have texture co−ordinates assigned to them, the last vertex of the body part will have a value of 0.

A demonstration of this technique, can be found in example 6.5, where the function that draws the front part of the torso is shown.

Example 6.5 The function create_torso_front (texture mapped)

```
void create_torso_front(float left[23][3], float right[23][3])
{
   int counter  ;
   float normal[3] ;
   float texLenght = abs(left[0][1])+abs(left[22][1]),
         texCooUp = 1,
         texCooDown = 0 ;
   for (counter = 0 ; counter <22 ; counter++ )
```

Copyright (C) Fotis Chatzinikos 1999

```
   {
      Calculate_Normal( left[counter], left[counter+1],              right[counter+1],normal)
      texCooDown += abs((abs(left[counter+1][1]) − abs(left[counter][1])))/texLenght ;

      glBegin(GL_TRIANGLE_STRIP) ;
         glNormal3fv(normal);
         glTexCoord2f(0.0,texCooUp) ;                    glVertex3fv(left[counter]) ;
         glTexCoord2f(0.0,1−texCooDown); glVertex3fv(left[counter+1]);
         glTexCoord2f(1.0,texCooUp);     glVertex3fv(right[counter]) ;
         glTexCoord2f(1.0,1−texCooDown); glVertex3fv(right[counter+1]);
      glEnd() ;
      texCooUp = 1−texCooDown ;
   }
}
```

When all modelling functions are changed the program is ready. Plate 6.8 contains the finished program. The user can manipulate the various buttons and pull−down menus in order to load a picture, create a texture, save a texture and so on. Table 6.5 contains a description of the components of the user interface and their associated actions. Plate 6.10 contains some screen shots from the final version, while Plate 6.9 shows the texture that was used to create the results shown in Plate 6.10.

The user interface of the texture mapping program

Toggle Full Screen	Switches between fullscreen and normal view (Image viewer)
Toggle Bitmap Viewer	Switches the bitmap viewer on and off (visible – invisible)
Load Bitmap	Shows a file browser and loads a bitmap image
Save Bitmap	Saves a texture as a bitmap file
Information	Shows some copyright information
Create Texture	Shows the selected image area in a separate window
Hide Texture	Hides the texture window
Cube	Displays the rotating, texture mapped cube (the spacebar toggles the animation on and off)
Torso	Displays the texture mapped torso (Previously described keyboard interaction applies also in this example)
WRAP_S	Selects a horizontal wrapping type (GL_CLAMP or GL_REPEAT)
WRAP_T	Selects a horizontal wrapping type (GL_CLAMP or GL_REPEAT)
MAG_FILTER	Selects a magnification filter
MIN_FILTER	Selects a diminution filter
TEX_ENV	Selects a texturing function
Change Size	Changes the selection box size (as well as the texture's size)
Exit	Exits the program

Table 6.5

Copyright (C) Fotis Chatzinikos 1999

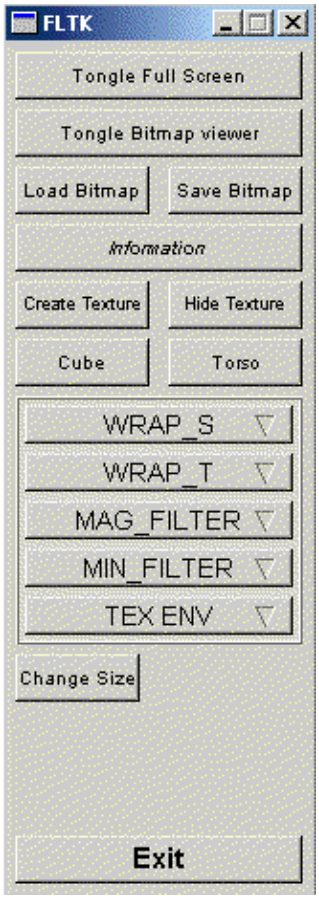


Plate 6.8

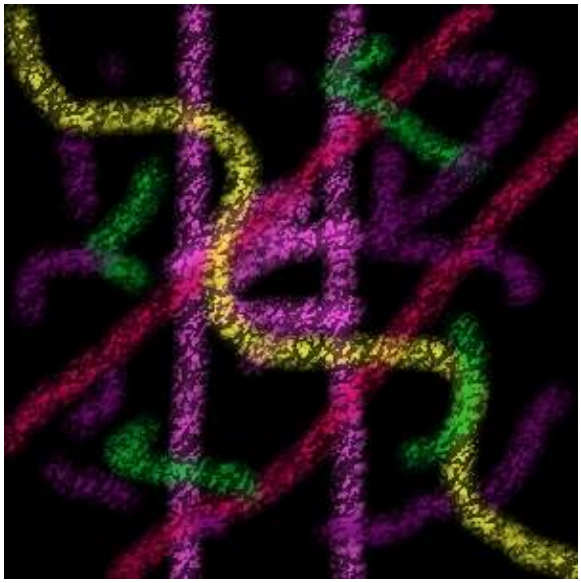


Plate 6.9

Copyright (C) Fotis Chatzinikos 1999

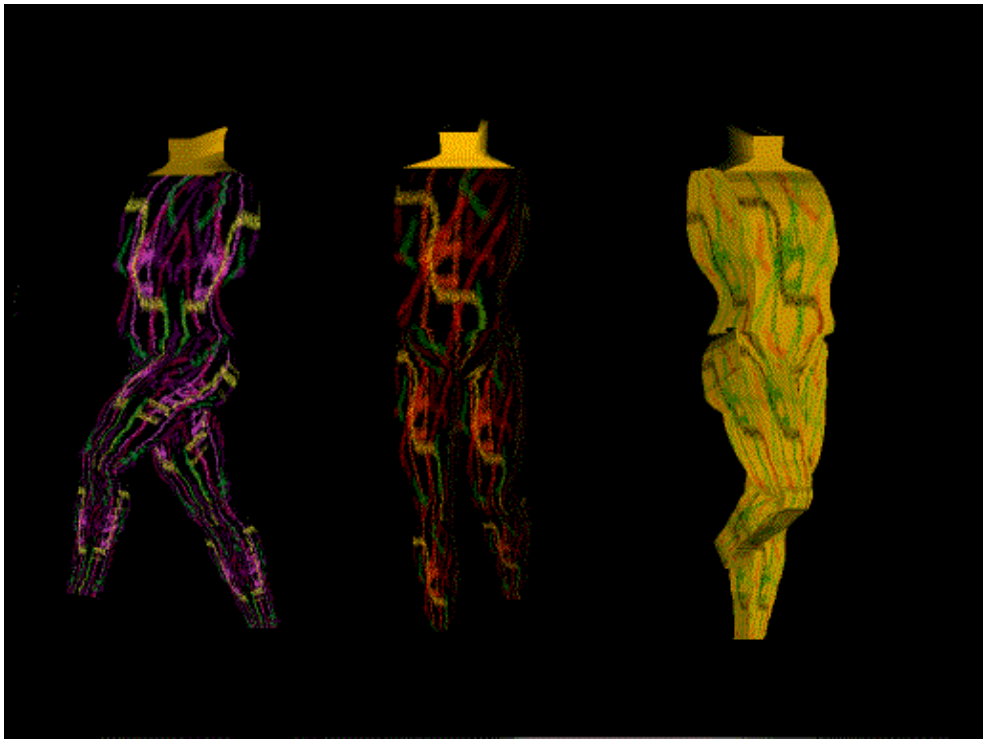


Plate 6.10

Copyright (C) Fotis Chatzinikos 1999

Chapter 7 − Conclusions – Future possibilities

This project constituted an introduction to OpenGL and three−dimensional graphics. It was an effort for the writer to learn and at the same time attempt to design a tutorial on this subject, so that future readers will be able to follow his work and possibly expand it.

This paper began with the discussion of the basics of both three−dimensional graphics and the OpenGL structure. The introduction contained the theoretical framework needed for the project, including the reasons for the specific structure followed. The topic of the second chapter was simple window construction, as OpenGL needs a graphical (windowing) operating system, and the introduction of modelling and projection transformations. In the third chapter a first attempt was made to create a simple model of a man and the appropriate animation cycle, which resulted in a model constructed from basic geometrical shapes (spheres and cubes).

The fourth chapter introduced OpenGL's lighting model and continued with the discussion of materials and their properties. A program was constructed where a user can experiment with the light and material properties in order to familiarise with the concept. A more elaborate geometrical example was presented in the fifth chapter, as its discussion topic was the improvement of the basic, until now, model. The sixth chapter introduced texture mapping, a technique that enables the use of images as parts of objects, making OpenGL programs more attractive. The topic of texture mapping has not been thoroughly exhausted, as the subject is quite complicated and the applications of texture mapping are inexhaustible.

During the specific time limits that were set for this project, all its primary objectives were accomplished. Given more time, further elaboration and 'special effects' could have been achieved. The list of those is practically unlimited but some ideas include shadows, fog, blending, collision detection, 'selection and feedback', and possibly the use of the DirectX component Direct Sound for sound effects. Actually, research was made on the previous two topics but it was not published in this paper as completion was not achieved.

This project was a good opportunity for the writer to be introduced in long scale, real life problems in opposition to the academic, small scale practical coursework. It would be an accomplishment if the present paper manages to assist people who wish to make a start with three−dimensional graphics and OpenGL.

Copyright (C) Fotis Chatzinikos 1999

Appendix I – Using Borland C++ 5.02

The main environment used in the development of this project was Borland C++ 5.02. In order to open windows using the OpenGL tool kit (GLUT) the C project must be specified as WIN32. The problem is that a WIN32 project is not as simple as a DOS C program, as the main function is not the one used anymore. The best way to open windows using GLUT is to build a WIN32 console project. This type of project still uses the main function as its basic function but has all the advantages of a WIN32 application. The following steps are needed in order to build such a project in Borland C++ 5.02.

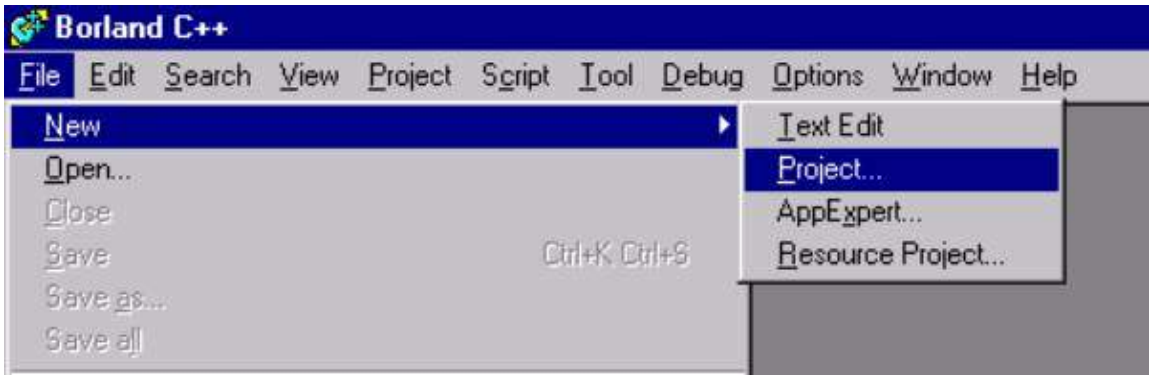


Figure I. 1

Run Borland C++ 5.02. Wait until the environment is loaded and select : New −> Project (figure I.1).

When this is done a window with several options will appear (figure I.2).In the sub−window named *Project Path and Name*, type in the path of the project (or press browse and select a path). In *Target Name* type in the project name. At sub−window *Target Type* select *Application[.exe]*. At the sub−window *Platform* select *Win32* and at *Target Model* select *Console*. Do not change any other options and press OK.

The new project is ready (figure I.3) !

Copyright (C) Fotis Chatzinikos 1999

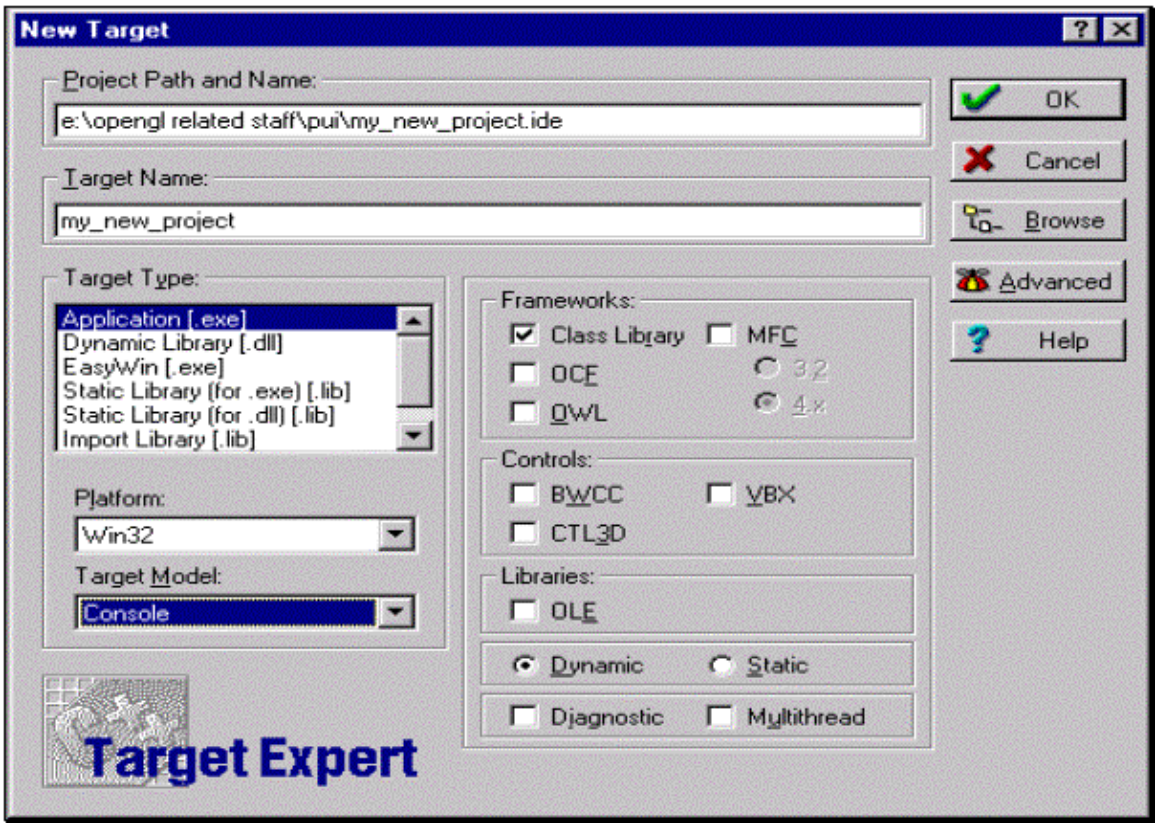


Figure I. 2

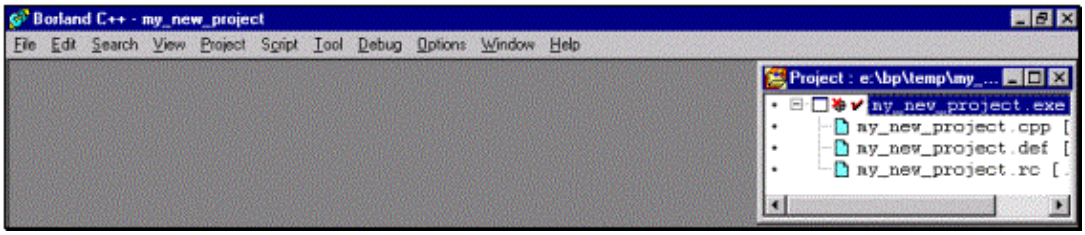


Figure I. 3

Delete any files the environment has created (such as my_new_project.cpp) by pressing the right mouse button on the file name and choosing *Delete node*.

Now insert the needed files for the project by pressing the right mouse button on the project name and selecting *Add Node* (figure I.4).

Copyright (C) Fotis Chatzinikos 1999

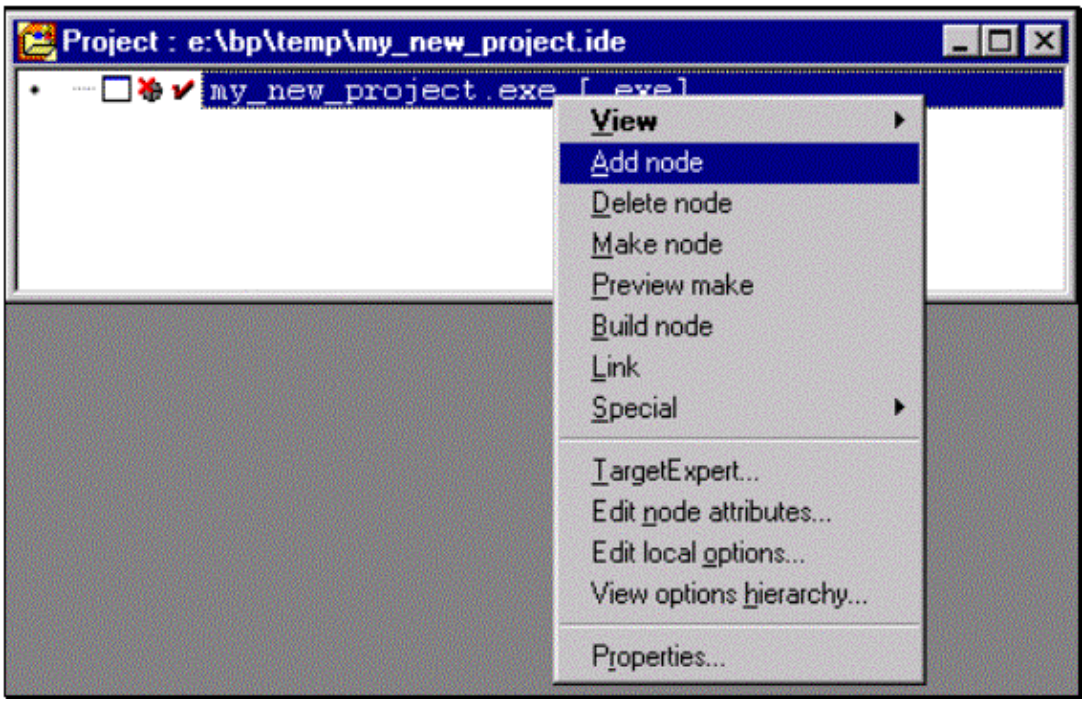


Figure I. 4

A browser window will appear, choose all the needed files (such as main.c etceteras). Insert also the files opengl.lib, glu.lib and glut.lib. These three libraries are the ones needed in order to use OpenGL. The environment is now going to look something like figure I.5.

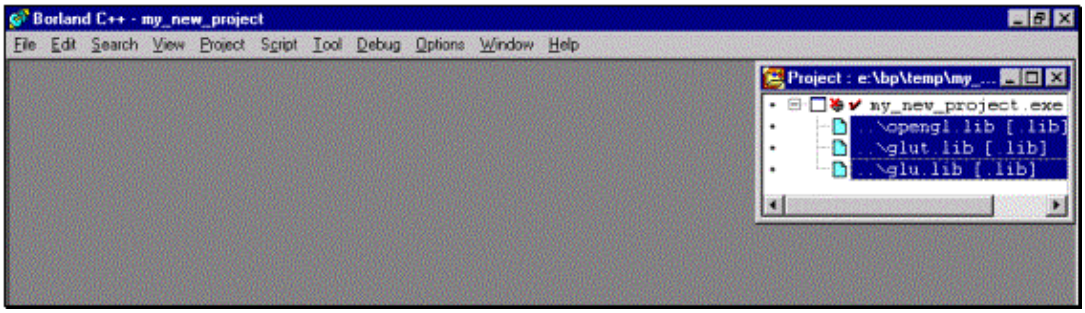


Figure I. 5

The project is now ready. The program can be executed by pressing *Debug−>Run*.

If you do not have the borland libraries, you can create them using the following procedure:

Find the opengl and glu dynamic link libraries (.dlls). These come normally with your graphics card drivers. When you have located them copy them in a temporary directory, and use borland's command line program 'implib'.

This program takes as input a dynamic link file (dll) and produces the corresponding library file (lib).

Copyright (C) Fotis Chatzinikos 1999

Appendix II – Using The FLTK Library

FLTK (Fast Light Tool Kit) is a GUI (graphical user interface) for UNIX (X−Windows) and Windows (95/98/NT) and is fully compatible with OpenGL. Because of its compatibility with both windows systems it was used for the creation of buttons and other widgets for some of the programs created in this project.

The following example of how to use FLTK to build a simple FLTK window with a box (widget) inside it saying hello world is taken from the FLTK help file.

More examples on using FLTK (taken from the created OpenGL programs) and the specification of the FLTK tool kit will appear in the website (www.dev−gallery.com).

```
#include <FL/Fl.H>
#include <FL/Fl_Window.H>
#include <FL/Fl_Box.H>
int main(int argc, char **argv)
{
Fl_Window *window = new Fl_Window(300,180);
Fl_Box *box = new Fl_Box(FL_UP_BOX,20,40,260,100,"Hello, World!");
box->labelsize(36);
box->labelfont(FL_BOLD+FL_ITALIC);
box->labeltype(FL_SHADOW_LABEL);
window->end();
window->show(argc, argv);
return Fl::run();
}
```

All programs must include the file <FL/Fl.H>. In addition the program must include a header file for each FLTK class it uses, here <FL/Fl_Window.H> and <FL/Fl_Box.H>.

The program then creates a window and then creates the widgets inside the window. Here a single Fl_Box is created. The arguments to the constructor are a value for the box() property (most constructors do not have this), values for x(), y(), w(), h() to define the position and size of the box, and a value for label() to define the text printed in the box.

All the widgets have several attributes and there is a method for setting and getting the current value of each of them. box−>labelsize(36) sets the labelsize() to 36. You could get the value with box−>labelsize().

Often you have to set many properties, so you will be relieved to know that almost all of these methods are trivial inline functions.

labelfont() is set to a symbolic value which is compiled into a constant integer, 3 in this case. All properties that cannot be described by a single small number use a 1−byte index into a table. This makes the widget smaller, allows the actual definition of the property to be deferred until first use, and you can redefine existing entries to make global style changes.

labeltype(FL_SHADOW_LABEL) also stores a 1−byte symbolic value, in this case indicating a procedure to draw drop shadows under the letters should be called to draw the label.

The constructor for widgets adds them as children of the "current group" (usually a window). window−>end() stops adding them to this window. For more control over the construction of objects, you can end() the window immediately, and then add the objects with window−>add(box). You can also do window−>begin() to switch what window new objects are added to.

window−>show() finally puts the window on the screen. It is not until this point that the X server is opened. FLTK provides some optional and rather simple command−line parsing if you call show(argv, argc). If you

Copyright (C) Fotis Chatzinikos 1999

don't want this, just call show() with no arguments, and the unused argument code is not linked into your program, making it smaller!

Fl::run() makes FLTK enter a loop to update the screen and respond to events. By default when the user closes the last window FLTK exits by calling exit(0). run() does not actually return, it is declared to return an int so you can end your main() function with "return Fl::run()".

Copyright (C) Fotis Chatzinikos 1999

Appendix III – Using Paint Shop Pro 5.0

The task of chapter four was to improve the simple (in chapter one) model of a human into something better than just rectangles and spheres. In order to do something like that data were needed in some form of a human body.

In the Internet there many resources of freely available data sets of human models. Another less straight forward way was chosen in this project. If the ready made data (from the Internet) were used the developer of this project would not know how to retrieve such data.

Nowadays three dimensional scanners are available but their price is so high that they are not yet massively available. In such a case that somebody does not posses a 3D scanner but has to model an object, and the objects data set is not freely available in the Internet, then the question is what happens?

In this project the pessimistic (but realistic) approach was chosen that the developer does not have any access to a 3D scanner and that the data set of the object is not available. In this case other means of retrieving the data of a three dimensional object have to be found.

One such technique is the one described in the following lines. For the purpose of this task, the painting program Paint Shop Pro 5.0 was used.

Firstly, three photographs of a human male body were scanned from a book on anatomy [5−10], a front, a back and a side view. These three photos were then put into Paint Shop Pro (figure III.2) . Four more layers were created on top of the basic one (the background) these four layers held the following data :

- 0 (background) the picture
- 1 black background (invisible in the beggining)
- 2 vertical rulers
- 3 horisontal rulers
- 4 points

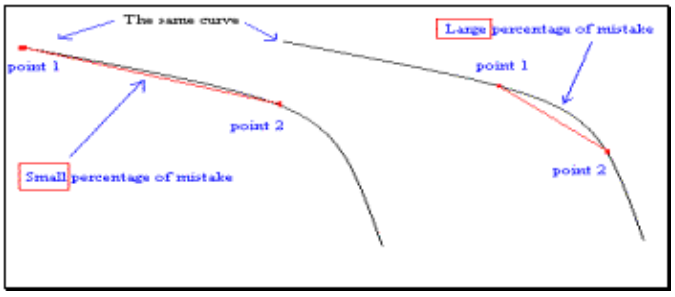


Figure III. 1

By using the mouse points were drawn (on the points layer) wherever a curve changed direction (in order that two consecutive points could form a line without a big percentage of loss of information) figure III.1.

Copyright (C) Fotis Chatzinikos 1999

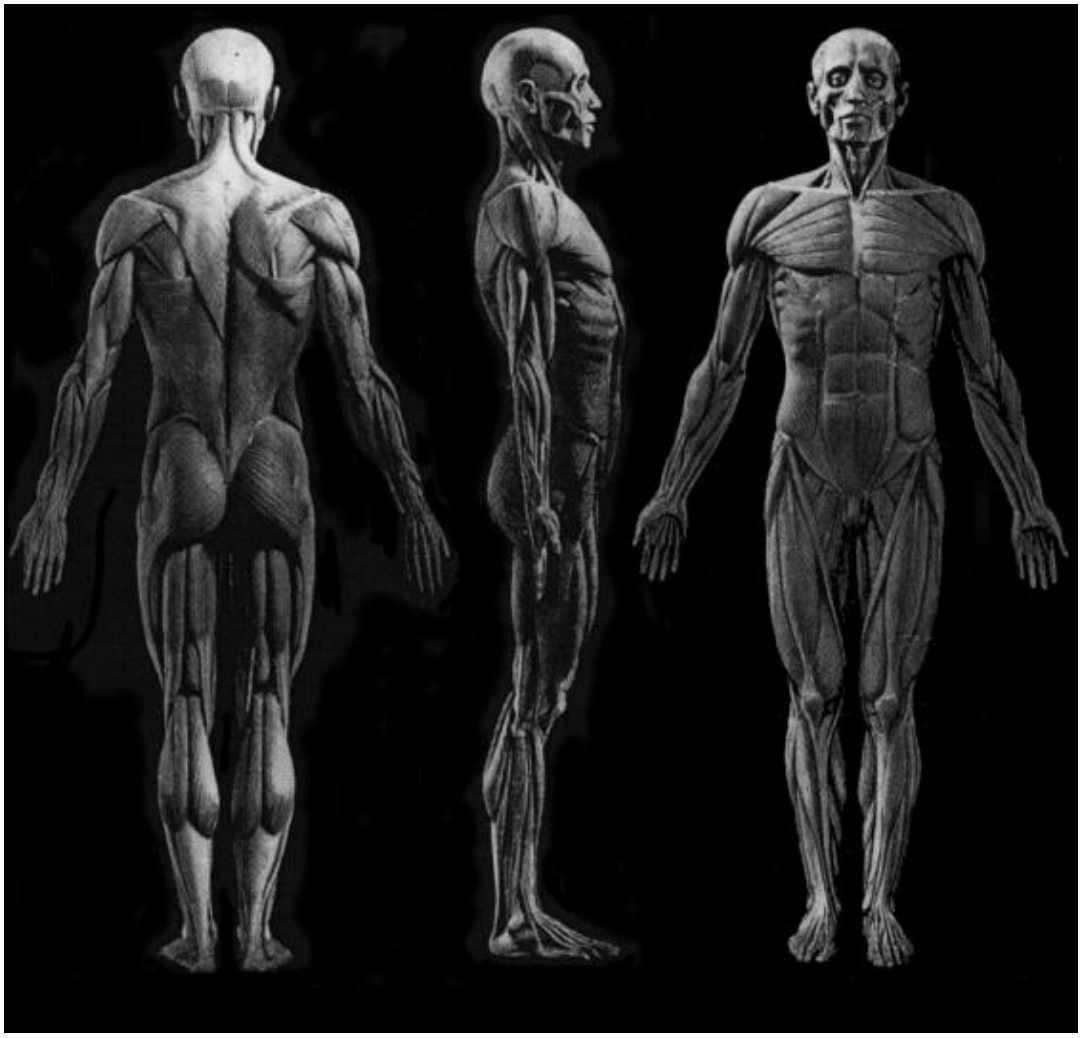


Figure III. 2

Figure III. 3

By following the technique demonstrated at figure III.1 the points of the body were retrieved from the three two dimensional images (figure III.4). By disabling all layers except the points and the black background layer the points can be observed quite better (without any other confusing information) (figure III.3).

Copyright (C) Fotis Chatzinikos 1999

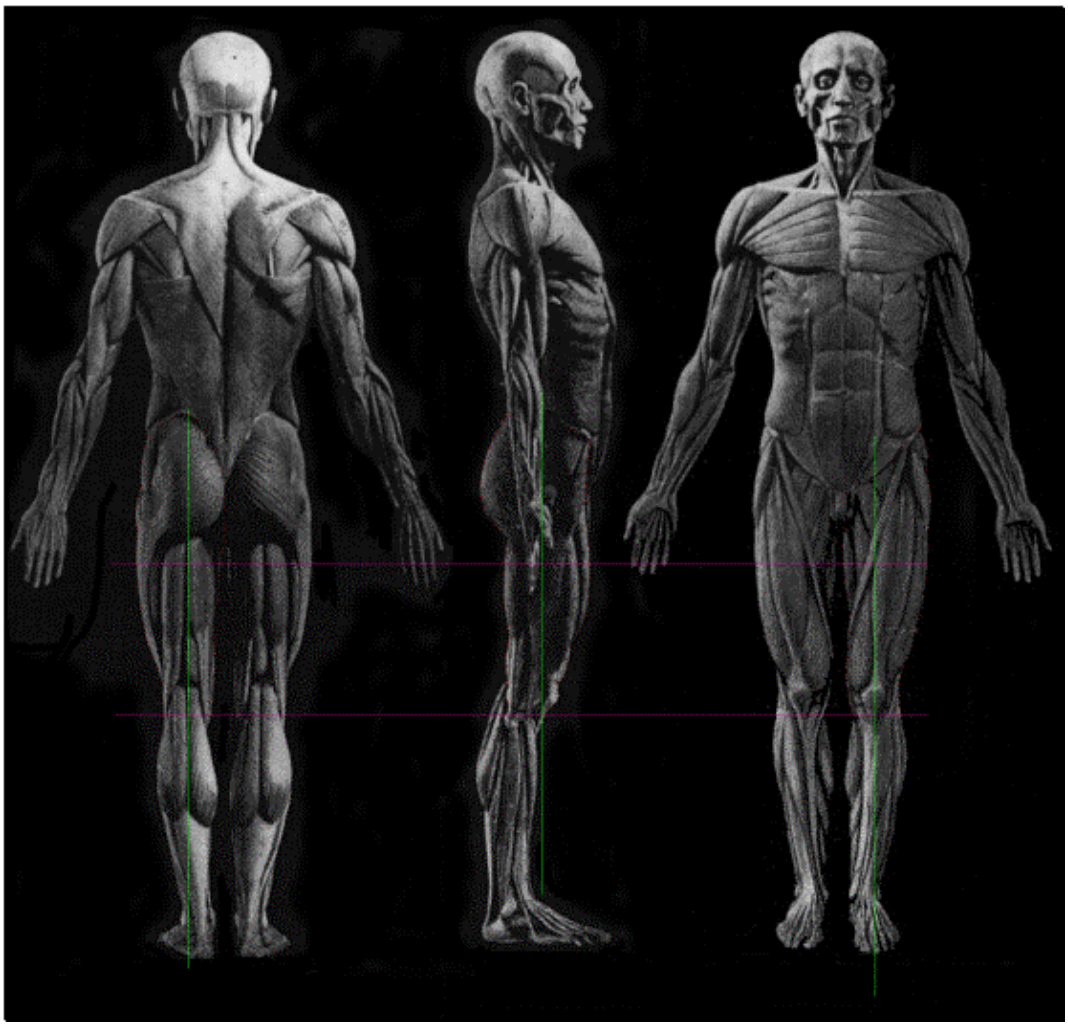


Figure III. 4

By working on several pictures like figure 3 the relations between height width and depth can be retrieved (further information in the final report).

The task is completed! A three dimensional image has been created by the manipulation of several two dimensional ones.

Note : The photocopies of human bodies used in this project were found in Farnham, Surrey (Library of the College of Arts).

Copyright (C) Fotis Chatzinikos 1999

Appendix IV – Bibliography

[1] **Moving Pictures** (In Greek)
Tony White

[2] **The Male And Female Figure In Motion**
Eadweard Muybridge
Dover Publications, Inc
New York

[3] **OpenGL Programming Guide** (Second Edition)
OpenGL Architecture Review Board
Mason Woo, Jackie Neider, Tom Davis
ISBN 0−201−46138−2

[4] **OpenGL Superbible**
Richard S. Wright Jr, Michael Sweet
ISBN 1−57169−073−5

[5] **The Human Machine** (The anatomical Strucure And Mechanism Of The Human Body)
George B.Buidgman
Dover Publications, Inc
New York

[6] **Anatomical Man, Bones And Muscles For The Student**
Silvio Zaniboni
London
Alec Tiranti 1963

[7] **Illustrator's Figgure Reference Manual**
Bloomsbury
ISBN 0−74750−008−8

[8] **Anatomical Diagrams For The Use Of Art Students**
James M.Dunlop A.R.C.A.
G.Bell & Sons Ltd, York House Portugal Street
W.C.2 MDCCCCLII

[10] **Anatomy For Artists**
Eugene Wolff
Fourth Edition (1962)

Copyright (C) Fotis Chatzinikos 1999

[11] **OpenGL, GLU and GLUT specification manuals**

[12] **http://www.opengl.org**

[13] **http://www.gamasutra.com**

[14] many many more Internet sites

Copyright (C) Fotis Chatzinikos 1999