

**Computer Graphics**  
**Prof. Sukhendu Das**  
**Dept. of Computer Science and Engineering**  
**Indian Institute of Technology, Madras**  
**Lecture 40**  
**Graphics Programming using OpenGL**

Welcome to the lectures on computer graphics, today we are going to discuss the second part of OpenGL, the current standard of computer graphics which is accepted both by the industry and by any academician.

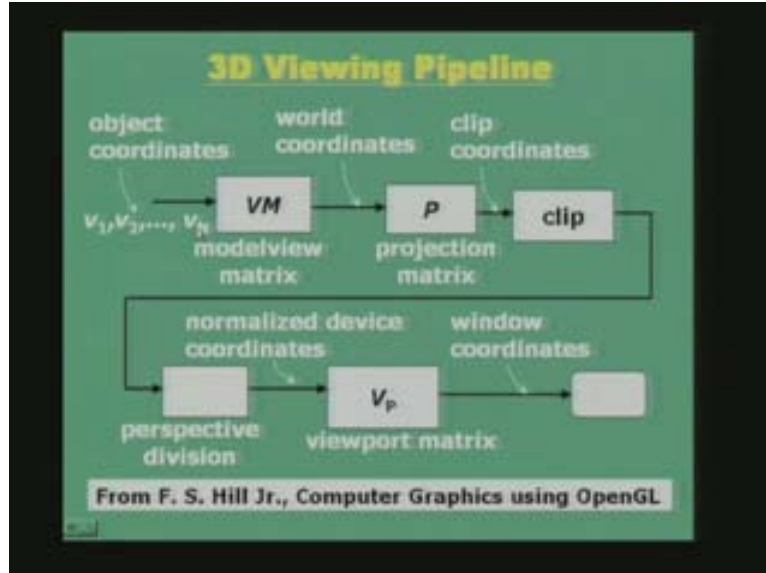
In the last lecture we have discussed quite a few points about OpenGL. We started with why OpenGL is accepted as a standard, what is OpenGL, what are the different features it supports starting from what you can draw, what you can shade, what you can clip, what you can render and what sort of transformations you can provide. These were the different features of OpenGL. And we also discussed that the OpenGL is in event driven. You can visualize it as a state machine it is an even driven mechanism. We have seen the framework of OpenGL the general block diagram in which we talked about per segment operation, per vertex, per fragment operations, pixel operations, geometric data based operations.

So we had seen that framework of OpenGL in the block diagram and of course towards the end we also discussed about the abstractions, different abstractions of OpenGL starting from GL, GLU and GLUT or the GLUT as it is called. We winded up the last lecture with the corresponding 3d viewing pipeline of OpenGL.

So we start from there where we left in the last class and if you look back into the slide right now the 3D viewing pipeline which has been borrowed from the book by F. S. Hill Junior computer graphics using OpenGL, I will give that reference at the end and the other two manuals on OpenGL which we have referred. So, at the beginning you see that this 3D viewing pipeline now OpenGL is similar to the viewing pipeline which we studied in 3D viewing transformations and we start with a set of object coordinates, it could be a set of vertices mostly and those vertices could be the vertices of a line, it could be the vertices of a polygon or it could be the vertices of a polyline or whatever it is.

So we start with the set of vertices it could be points by itself. So starting with object coordinates which are the vertices let us say it passes through what we call as modelview matrix that is the term in OpenGL called the word modelview matrix multiplication of trume it is called vm matrix and that generates the world coordinates for you.

(Refer Slide Time: 03:32)

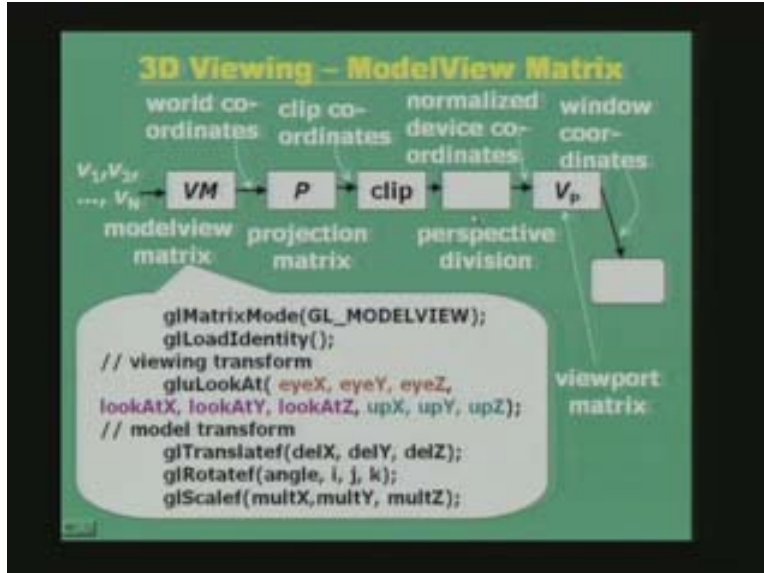


Those world coordinates pass through a projection matrix where you define the projection parameters, the perspective orthographic, you define the look up angle, look at angle, look at vector basically when the world coordinates will pass through the projection matrix it will generate the clip coordinates which have clip. So, there are clipping tools available for you provided by OpenGL. And then of course it goes through a perspective division operation in the canonical view volume to generate the normalized device coordinates ndc this is also not a new term for you.

You remember the canonical view volume, so normalized device coordinates and then of course it will pass through a viewport matrix  $v_p$  to generate the window coordinates and that will be put on the screen or the memory buffer or the frame buffer. That is what we see as the block diagram of the 3d viewing pipeline if not exactly same very similar to the viewing pipeline which we have studied earlier in the course when we talked about 3D viewing transformations and viewing pipeline. So we look back to the 3D viewing pipeline, we will go through a few stages of this as we go along.

Again I repeat, this is a block diagram taken from the book by F.S. Hill Junior computer graphics using OpenGL. So what does this viewing modelview matrix do? We will analyze one by one as much as possible. This is a same viewing pipeline which we have seen it is now condensed quite a lot for us to provide the transformations. This is the first time you are seeing OpenGL code. We will see a lot of good demonstrations today on OpenGL program starting from simple programs to form plot to a polygon filling and also we are going to have wireframe diagrams, shading, rotation with animation and things like that towards the end of the lecture where we will wind up.

(Refer Slide Time: 05:32)



So if you look back into the modelview matrix that consists of at least this following commands where we have a `glMatrixMode` this GL is the graphics library for the OpenGL that precedes any commands which are used for command functions or libraries used for the OpenGL matrix `glMatrixMode GL_MODELVIEW` where you define the model view and `glLoadIdentity` will load identity matrix for you.

The viewing transform `gluLookAt` that is the higher level function when you see a `glu` instead of a `gl` that is a higher level abstraction of `glu` instead of `gl` where you have a look at, where you talk about eye X Y Z the position of the eye X Y Z coordinates look at X Y that is the point at which you look at a particular point. You remember the concept of cvv and the direction of projection vector where you position yourself and keep looking at the particular point where your eye or the camera is at a certain position and you keep looking in a certain direction. That is what you have as `lookAtX lookAtY lookAtZ` and the `eyeX eyeY eyeZ` at the position of your eye or the camera.

You need the up vector `vup` if you remember when we discussed about 3D viewing pipeline so that `vup` vector is defined by the `upX upY upZ`. As you see all these three positions and these vectors are defined by the corresponding X Y Z components of a vector or coordinates as may be necessary. So that is the `gluLookAt` command and which is the viewing transform function and the model transform function is `glTranslatef` function.

We will see what this `f` stands for, `f` basically stands for the floating point values and similarly we will have `glRotatef` where `f` again stands for the floating point were you define what amount of rotation you want or what angle and `i j k` be the direction cosines again in float of the axis of rotation.

And of course if you need to scale this is another example you will scale, again this f stands for the floating point values of the factor X Y and Z along the diagonals of the scaling matrix. So you can provide different types of data types corresponding to the function, you can provide say data type integers floating point and different types one sign bytes and all that. Therefore, based on that the corresponding sign f you see here might change to b, if you provide an 8 bit integer this f which you see here will become b. It could become a short integer 16 bit then you should provide s. If you provide 32 bit integer which is a hint long then you can change it to i and f stands for a 32 bit floating point operation.

If you change f to d here in any one of these then you talking of a 64 bit floating points which is a double or a long float. And there are concepts called as gl double, gl float, gl integer, gl byte depending upon the type definition you need.

You can also use ub, us or ui unsigned integer, unsigned long integer and unsigned short integer respectively and corresponding to b s and i which are a byte, b stands for byte, s stands for short integer and i stands for long integer. These are the conventions which are used correspondingly you will have the gl double, gl short ,um gl byte, gl short and gl integers and typically you can similarly have the unsigned byte, unsigned short integer or unsigned long integer. So these are the different data types which are also supported by OpenGL you should of course look into the blue book and the red book manuals.

I will give the reference at the end of the class today but I mentioned this in the class earlier. You have to pick up sample program from the book either from the book by F. S. Hill or from public domain websites or even from those given in the reference map programming guides not reference manuals they will not have programs.

Reference manuals will have syntaxes of the commands to be used, programmers guide may have sample programs. Take that and write your own programs make your system OpenGL ready. My system is OpenGL ready we should be able to see the most today. And you should also be able to do the same on your system, make it OpenGL ready by downloading the DLLs driver files and the header files then write a short program and see how it works. So, coming back to the 3D viewing modelview matrix 3D viewing pipeline we are discussing modelview matrix which transform the vertices or the vertex coordinates to the world coordinates using some of these or all of these commands here.

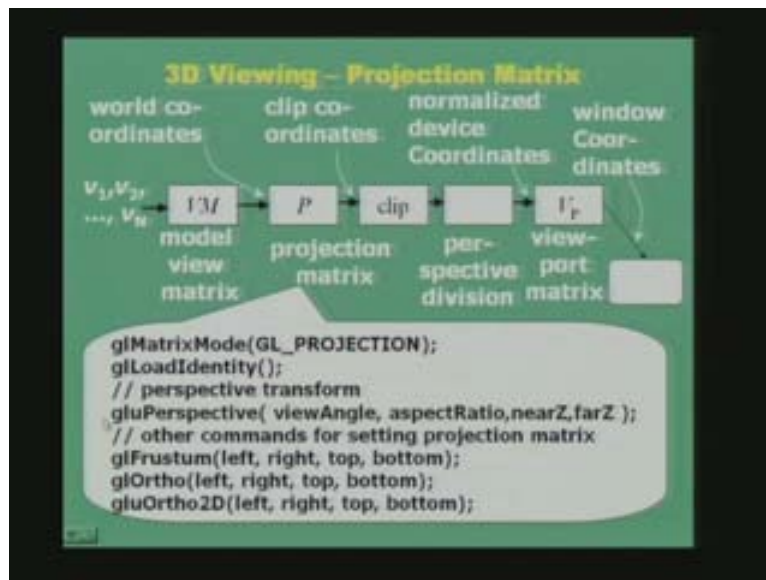
I repeat here the f stands for the corresponding data type used here and there are various types of data types which we already discussed starting from short integer, long integer to float double, unsigned byte, unsigned integer and so on.

So let us look into the next one which is the projection matrix. The same viewing pipeline we discussed the modelview matrix, an example. Let us look into the projection matrix now which is a glMatrix Mode GL\_PROJECTION which you need to define then a glLoadIdentity matrix again, a perspective projection functional call given by GLU that is a higher end abstraction of gl.

glPerspective which gives the viewAngle for the camera angle, aspectRatio, nearZ plane farZ plane etc and other commands for setting the projection matrix can also be as given in the following. So if you see the perspective here as I have used a viewAngle aspectRatio nearZ farZ is enough for you to define perspective, why?

If you remember the perspective geometry of course in terms of theory we have discussed at length, various types of equations and perspective transformation as well. Well some book tries to simplify, there are some literature which tries to simplify the perspective saying that since you anyway have to visualize a finite pyramid through which you are viewing or infinite pyramid and that of course infinite pyramid before the normalization is done to canonical view volume you can visualize infinite pyramid in perspective projection and that is made finite by a near plane and a far plane closer to you and at distance to you so you have a finite pyramid now.

(Refer Slide Time: 10:13)



Then instead of talking about direction of projection vector and all other parameters of course you have to define your view up vector but after that you can just define an angle, the angle of the subtended by the pyramid at the vertex. So this is visualized as called as a view angle. So view angle, if this is the perspective projection canonical view volume the pyramidal structure the finite one, the two sides of the two planes will actually meet at the vertex. You know how a pyramid looks like. So that angle subtended at the pyramid is called the view angle so that is what you specify and you also specify the nearZ and the farZ planes.

You can specify that the nearZ and the farZ planes and then you also specify the aspect ratio X, Y. So these are the parameters which can simply define your perspective projection.

So `glFrustum` you provide with the left, right, top and bottom for orthogonal projections orthogonal 2D or orthogonal almost similar commands where you provide your left, right, top, bottom planes to define your `GL_projection` matrix. You can use either perspective or orthographic and similar commands can be used.

So we will move forward to the structure or the first structure of a GLUT program or the GLUT program and the rest of the course coverage will be mainly based on example programs. We will talk about the general structure of the GLUT program and then take one or two examples and of course I have three or four demos which hopefully work in front of you today. So the structure of the GLUT program is something like this as you see on the screen. Of course I assume here that all of you have concepts of C or C++ based programming at least and C++ and this is nothing new to you we are talking of a GLUT.

You have to initialize the GLUT environment by a `GLUT init` command and then of course you have the `glutInitDisplayMode`. So you set up the display, there are various options which you can provide based on pipelining mode this is called the double buffering and the GLUT, you want the environment to be colored with RGB. Of course one of these are optional, you can actually switch off this and then it becomes black and white. And of course when you talk of GLUT DEPTH you are talking about z buffering to enable it in your display mode..

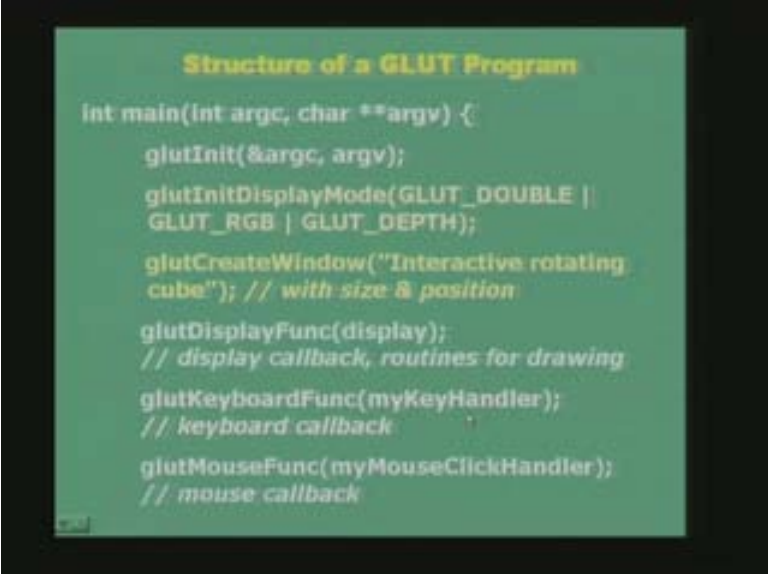
This is the main function which will create the window for you `glutCreateWindow`. And the title of that window will be as given whatever you put in double quotes interactive rotating cube will have rotating cube where we will see that this particular title bar will be available on the window. So you can provide this option with corresponding size and position of the window. Also you can provide to nowhere you want your window to be positioned. Otherwise of course there is always a default position which this GLUT window will be created.

I must admit here that I am quite thankful to a few of my research students who have helped me to make this OpenGL programs work for me and they worked for me for the last few weeks and to ensure that I have some sample programs to study and they work for you to visualize what exactly goes on in OpenGL.

So let us go back `glutCreateWindow` is what we have discussed after `glut init`. This is more or less a sequence. We will see a sample program which runs on the system and the `glut init` initializes then you initialize the display and then you create the window at a certain position and the size. And then of course this is the main function the `glutDisplayFunction` called as the display function or the display call back function where we have to replace the routines for drawing in a display routine which we will see. This will be called and all routines which you want to display used for displaying graphics are to be placed under that display function call. So this is the display call back function of `glut` higher level tool kit under `gl`.

Then of course you can also have `glutKeyboardFunction` `mykeyboradHandler` where you talked about keyboard call back. Of course we have not discussed much about interactive nature event driven and sample driven interactions in graphics but as we discussed that the OpenGL is an event driven system.

(Refer Slide Time: 16:31)



```
Structure of a GLUT Program

int main(int argc, char **argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE |
        GLUT_RGB | GLUT_DEPTH);
    glutCreateWindow("Interactive rotating
        cube"); // with size & position
    glutDisplayFunc(display);
    // display callback, routines for drawing
    glutKeyboardFunc(myKeyHandler);
    // keyboard callback
    glutMouseFunc(myMouseClickedHandler);
    // mouse callback
}
```

Although we have a pipeline and a framework it's an event driven system so you need to interact with the user, the user can interact, of course in this case we assume that you can use the keyboard or mouse. So you need a keyboard and a mouse handler which is provided by the glut also. So this is an example of a `glutKeyboardFunction` and this must all be in your main program. You need to call this `glutKeyboardFunction` as `myKeyHandler`, so this is considered as the keyboard call back function.

`glutDisplayFunction` was the display call back function and the `glutkeyboardFunction` is basically a keyboard call back function. Similarly, you can have a mouse call back function. At the bottom of the screen you see you have a `glutMouseFunction` called the `myMouseClickedHandler`.

So we will see what should be kept, all display routines under `display` then all functions for your keyboard interface should be in the `myKeyHandler` and all interactions with the mouse should be under the `myMouseClickedHandler`.

You can have `glutmotionFunction` for `myMouseMotionHandler` for mouse move call back that means you want to say create animations or create movements in the pictures so you must know how much the mouse moved in terms of positions X and Y coordinates in both vertical and horizontal directions. So you need to actually enable that and as you see here we are already seeing a very good feature of OpenGL that it is device independent.

Hardly it is hardware independent because nowhere I am saying that I am going to use this sort of a mouse or a keyboard or which has 101 key functions or a 104 key functions.

Typically the OpenGL driver files and the header files are suppose to handle all that. Once your system is OpenGL ready you absolutely need not worry about what your keyboard is, of course you really have to worry about the few special functions of the keys. But typically all the alpha, numeric keys will function as they behave in your typical editor.

Now the last one is your glutmotionFunction which will track the motion of the mouse and then of course towards the last you have a general init where you put all initialization commands necessary for your OpenGL function say you want to set some color for your background, some size of the window which you want to set so all the global variables or the constants which you need through out your program can be put under the init.

Of course there was a glutinit remember which was used for initializing the glut environment or the OpenGL environment. But this init is your initialization of the programmers interface. Whatever global variables or constants you need to initialize you can actually put it in under the init program. And then of course the last and main one is the glutMainLoop.

This glutMainLoop is the one that, once it enters whatever programs you have put under the display, you remember the function display let us go back, here glutDisplayFunction displays all routines for drawing will be event driven by the key or the mouse through the glutMainLoop. So the glutMainLoop will start triggering the routines which are there in the display.

The glutDisplayfunction and the display routines to be displayed will be displayed here and it will go into the loop unless the program asks you to come out through the glutMainLoop.

Then of course as I said before you have to define these functions display, all routines must be kept here, all display routines myKeyHandler you need to talk about a key value and the position X and Y then of course you need a mouseClickedHandler to find out which button you have clicked and of course which mouse pad you have clicked and the myMouseMotionHandler will talk about the incremental movement of the mouse so that is what you have.



(Refer Slide Time: 19:30)

```
    glutMotionFunc(myMouseMotionHandler);  
    // mouse move callback  
    init();  
    glutMainLoop();  
}  
void display() {...}  
void myKeyHandler( unsigned char key, int x,  
int y) {...}  
void myMouseClickedHandler( int button, int  
state, int x, int y ) {...}  
void myMouseMotionHandler( int x, int y ) {...}
```

These are the minimum functions which you need. This is the overall structure of the OpenGL program this is not a sample program do not take it as sample program.

We will see sample programs and see its working on the system but this is what we call as the overall structure of the C based structure of an OpenGL program typically. I mean you should have a init, gluinit, glutcreatewindow, glutMainLoop and a function draw these must be there.

A program may not have a motion function or it may not have a keyboard handler it may not have a mouse handler but typically the glutinit, create window, display function init and glutmainloop are necessary and this will event drive, event based we talked about this as a state machine where it will be driven by the interface. But even if there is no interface a glutmainloop will actually trigger the functions in the display. As you see here the void display function will be triggered by the glutmainloop. So let us try to look at some sample programs today and of course we will run a few sample programs today to see how things go on.

So we look into viewing in 2D, let us look at an example of viewing in 2D. In 2D now again this is just trying to elaborate and not only give structure but some examples of how these structures must be used to write a program. We are looking in to the init program and let us say what I was mentioning earlier that all initializations in terms of the background clear color then you might have to define a foreground color with a 3f remember this f has come again. It basically means that this is a floating point value then you can define your point size, your matrix mode and identity. So these are some examples which you can put in your init program for initializing the environment before the OpenGL starts triggering.

This is an example of a `gluOrtho2D` function which I can use. It basically sets up my viewport in 2D starting from 0, 0 as my left. Remember, the `gluOrtho2D` is defined in terms of floating point values as 0. 0 screenwidth depending upon the left to right 0 to screen width plus 1 is what you will have and of course you will have 0 to screenHeight bottom to top.

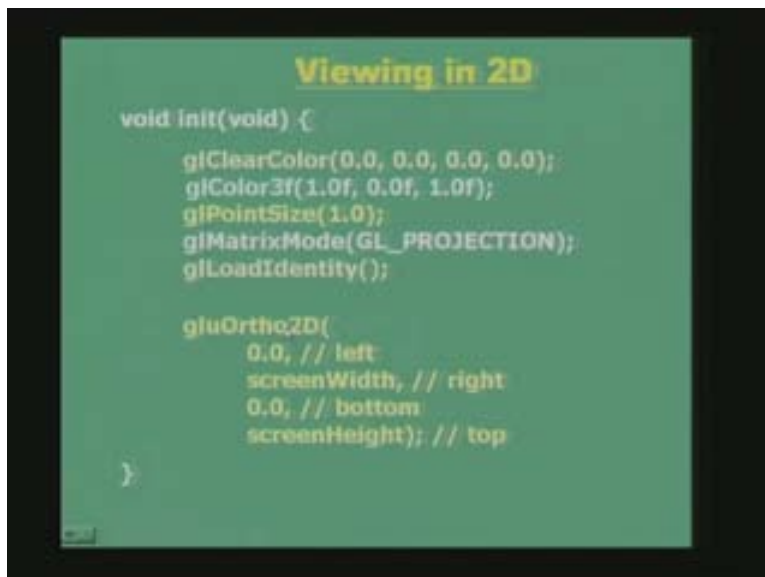
Left to right is what you have, the horizontal is the number of left to right virtual coordinates in the screen and the screen vertical coordinates bottom to top is 0 to screenHeight.

It is the floating point value, remember it could be from 0 to 1 or it could be 0 to 10 the OpenGL will map it automatically to device coordinates so you do not need to bother.

You do not need to bother but depending on the size of your screen which you specified if you need to basically resize the screen you have to use some other functions but this is the user defined coordinates the `umin umax the vmin vmax` is defined from 0 to screenwidth, 0 to screenHeight and that has to map on to certain coordinates based on the window viewport which you have defined using your function for initializing the screen. So, say that could be from 0 to 640 by 480 window size or you could have 320 by 480.

That is an integer because those are integer pixels on the screens or the frame buffer. So the corresponding mapping of the users view port to the device screen will be automatically handle by OpenGL. You work in your user specified coordinate whatever the programmer specifies and the corresponding mapping will be done by OpenGL. That is the example we saw as viewing in 2D. Let us look into this; this was the example of viewing in 2D. Those who want to note down some of these commands can note it down.

(Refer Slide Time: 23:37)



```
void Init(void) {
    glClearColor(0.0, 0.0, 0.0, 0.0);
    glColor3f(1.0f, 0.0f, 1.0f);
    glPointSize(1.0);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    gluOrtho2D(
        0.0, // left
        screenWidth, // right
        0.0, // bottom
        screenHeight); // top
}
```

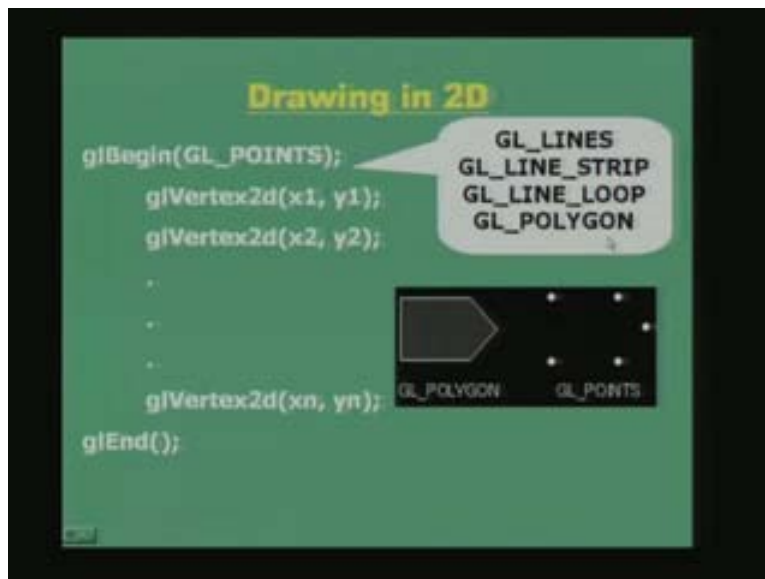
This is only just a trivial example for you. Let us look into another example drawing in 2D where this is what we mean by a set of commands defined by gl points starting with glBegin and glEnd. So this is a particular set of command forming in an event together. So that could be triggered by gl points so gl points defined by glVertex 2D.

This d stands for the data type double. You remember, the f stands for the floating point data type floating point and I mentioned that you can use integers, short integer, long integer, float double, unsigned byte, unsigned integer, all those and gl double, gl integer and all those types of datatypes are allowed in gl double. I am not giving a compliancy list of those, you please look into the manual because this is not a coaching in OpenGL.

You have to learn OpenGL, I am only triggering interest in to you so that you know the overall structure. I repeat you have to make your system OpenGL ready, have the blue book and preferably also the red book with you and then take sample programs and then start writing your own program either in Linux or Windows based environment and see how it works.

So going back towards an example in drawing in 2D we are talking of sequence of vertexes in 2D X1 Y1 X2 Y2 and so on up to Xn Yn assuming that these are defined earlier as constants or it could be array pointers that does not matter and you thus define it so basically you have a sequence of points.

(Refer Slide Time: 25:11)



And the way you draw it I mean with this set of points, here we will see with an example that you can only plot certain points or connect them using..... and set up a polyline or in fact filling between a set of polygon. So we are talking of a sequence of points here and that is what we define by drawing in 2D.

As I was seeing here this function can help you to simulate or emulate gl lines, gl line strip, gl line loop or GL\_POLYGON instead of gl points because as you see here I can define a set of points, I can define a set of lines or different line strips or a line loop or a gl poly line or a polygon. Different types of primitives are available for you to draw at a very lower level for the gl or OpenGL.

This is an example figure borrowed from one of the designers of the book where we discussed of the difference of the sequence of points generating either gl points or GL\_POLYGONs. So if you say it is a gl points then we have a set and assuming n to be 5 so you define say x1 y1 x2 y2 and there are sequence of five such commands here of glVertex 2D from x1 y1 x2 y2 up to so on up to say x5 y5. Suppose n is equal to 5 then this five points if they are set according to as the x y coordinates as given in the figure and if they are gl points then you only have this set of points plotted. If this is corresponding to a GL\_POLYGON then you can have the polygon filled with the certain shade or color or texture as you would have to specify somewhere else.

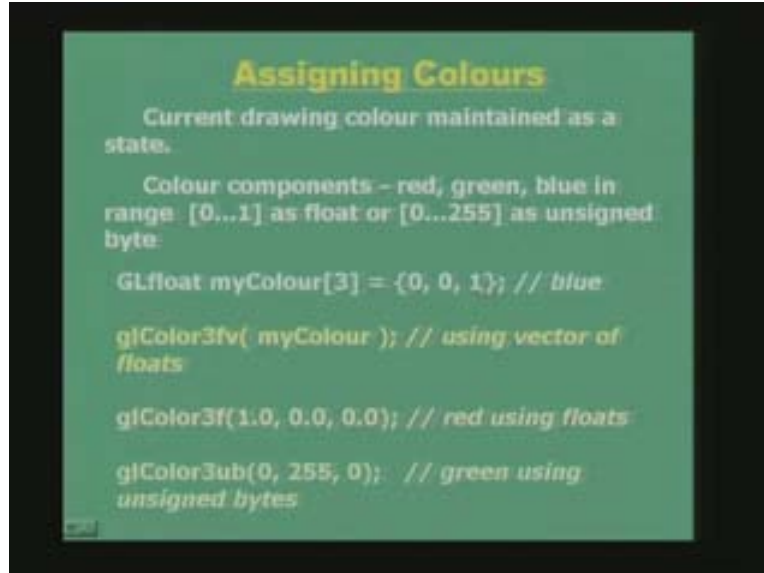
Of course the thickness of the point here and the shade which is inside the polygon also the texture inside the polygon or the color which you want to shade has to be defined a prairie and it has to be defined earlier and that will drive the state depending upon what types of primitives you are using here. The attributes for those primitives will be obtained from the current state of the OpenGL. So let us move to another important aspect of assigning colors. The current drawing color is maintained as a state.

The current drawing color is always maintained as a state in your OpenGL and the color components are red, green and blue in the range 0 to 1. So it is a floating point number as float or it could be in 0 to 255 range that is 0 xff in hexadecimal form as unsigned byte. So you can use unsigned byte 0 to 255 or you can use 0 to float from 0 to 1 as you prefer. So these are the color components, remember of course that we are using red, green, blue or the RGB format for display and of course there are various types of color standards available.

Some people may prefer the RGB, some prefer the HSV or the LAB the LAB or the HS or the CYB let us say, there are various types of color formats and color schemes available but as OpenGL will stick to RGB. So the color components are red, green and blue in the range 0 to 1 as float or between 0 to 255 which is 8 bit all ones and as a unsigned byte. So these are some of the commands which you can use for GLfloat myColour, the GLfloat myColour three component 0 0 1 R G and B.

So since this is blue this is a comment which is put here to show that this will give you the blue color or you can use any other mycolor which is defined earlier and use it under glColor 3fv where 3fv stands for floating point vector form myColour using a vector of floats.

(Refer Slide Time: 28:44)



So you are using a vector of floats is this format fv stands for in OpenGL glColor 3fv or you can use like the myColour3 defined earlier. If you use the same myColour3 here under this command then of course you will have the blue color.

However, if you have the glColor3f defined as 1 0 0 then you will have the color red. of course I am not putting the pure red color but I am putting a color of this statement as close as red that is possible, this was as close to blue and this is as close as green as possible. You can use C a different format glColor3ub, what does ub stands for? Unsigned byte 8 bit unsigned integer or unsigned byte ub. So we have seen f, of course we would have seen d, now you have seen ub. So these are the different formats for using data types in OpenGL.

Therefore, coming back as you can see this is a vector of floats sb. This is simply a set of floating point components of the color or you can use three unsigned byte or three unsigned integers from 0 to 255 ranges for your color.

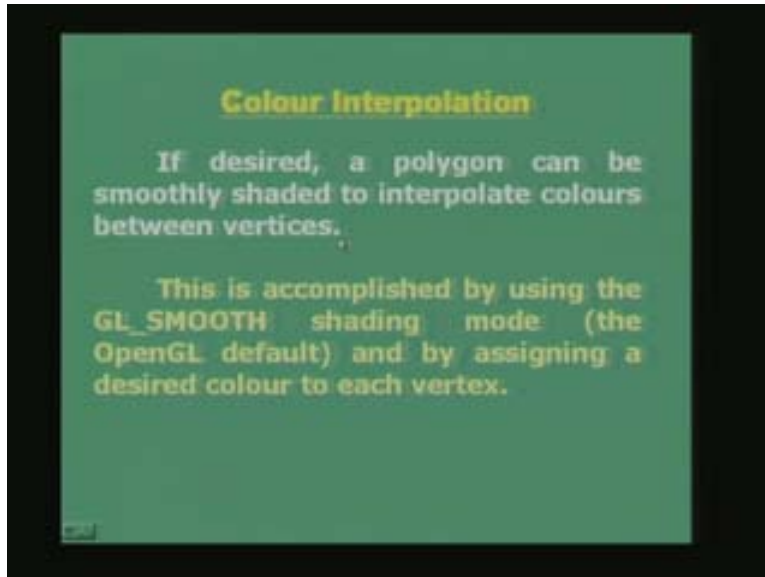
So you can use any one of these types of formats as I was talking about color components in red, green and blue the corresponding three components in any one of these parameters which you see here and they should be in the range 0 to 1 for if it is float and 0 to 255 for an unsigned byte. Color interpolation is a very interesting concept. So if desired a polygon can be smoothly shaded to interpolate colours between vertices. This is similar to the concept of Gouraud shading. If you remember, Gouraud shading we talked about interpolation.

If you remember the last section of shading and illumination which we studied towards the end after the form model we talked about the Gouraud shading model where the three vertices of a polygon we took a triangle example but it could be n vertices. The normals of the three vertices were different they were not the same and those three normals gave

three different colors and we interpolated along the edge and then for each scanline from one edge intersection to other you also did an interpolations scheme and the formulas were given to you at that time itself.

But you can use OpenGL itself, you do not have to write or code the formula where if you want if desired a polygon can be smoothly shaded to interpolate colours between the vertices.

(Refer Slide Time: 31:34)



And this is accomplished by using the GL\_SMOOTH shading mode which is the default OpenGL shading that is interesting. This is accomplished by using the GL\_SMOOTH shading mode option which is the OpenGL default option and by assigning a desired color to each vertex so that will be interesting.

You assign a different color for each vertex and then while filling up the polygon you use a GL\_SMOOTH option which is in fact the default option

If you do not want smooth, you want flat shading which we have seen, we will see the difference between flat shading and smooth shading with OpenGL today when we see a different type of primitive structures supported by OpenGL and we will shade the polygons for those 3D vertices and 3D objects which are primitives defined in OpenGL.

We will see the difference between the smooth shading and flat shading. We look into the slide and see the glShadeModel which talks about GL\_SMOOTH as an option as opposed to GL\_FLAT which you can use. And this is section of a code again under GL\_POLYGON which will be event driven and there are three types of vertices with three different colors. The first pair of statements will actually generate a vertex at 0, 0 in 2D, 0, x and y coordinates with color red.

I have given three different colors, I hope it is visible for the respective colors for the respective vertices here in the code itself. Similarly, you have the blue color for a vertex at 1, 0 and a green color because RGB those are the three components.

(Refer Slide Time: 33:07)

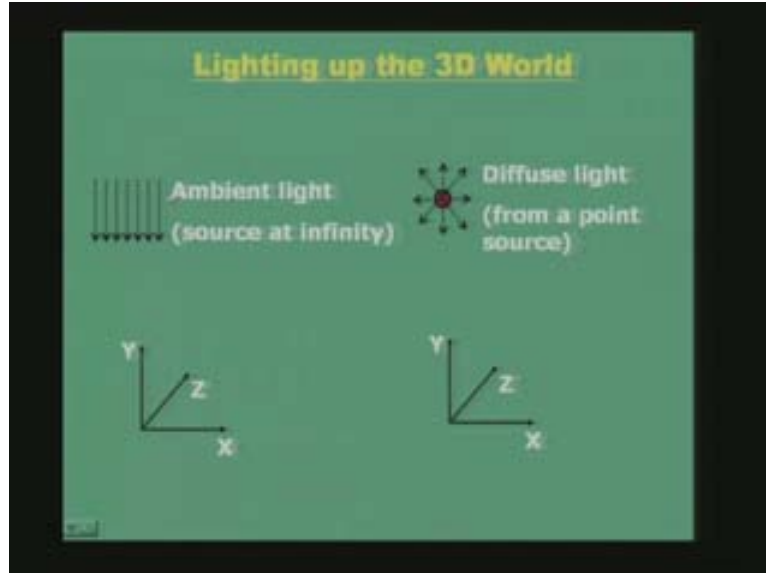
```
glShadeModel(GL_SMOOTH);  
// as opposed to GL_FLAT  
  
glBegin(GL_POLYGON);  
    glColor3f(1.0, 0, 0); // red  
    glVertex2d(0, 0);  
    glColor3f(0, 0, 1.0); // blue  
    glVertex2d(1, 0);  
    glColor3f(0, 1.0, 0); // green  
    glVertex2d(1, 1);  
    glColor3f(1.0, 1.0, 1.0); // white  
    glVertex2d(0, 1);  
glEnd();
```

So, you have r, then g here and the blue here and this will be the vertex green at 1, 1, at the coordinates 1, so at 1 the vertex will be green. And of course you can also specify that a white color will be at a vertex 0, 1 so that will give you a polygon with four different vertices, four different colors. And of course when you put a GL\_SMOOTH the color will be interpolate. We will see an example of how this output looks like.

Lighting up the 3D world we can talk about ambient light with a source at infinity or a diffused light from a point source where the light is of course going in all directions.

In case of ambient light of course it is an approximation of inter reflected rays, we know that. The diffused light is actually the case of where the light equally spreads in all directions from the point source and that source could also be assumed to be at infinity or at a finite distance.

(Refer Slide Time: 33:45)



Those options are provided to you by the OpenGL. So these are the two different figures which talked about the ambient light coming from a source at infinity or the diffused light going in all directions from a point source at a finite distance.

Based on this we can talk about a light zero color telling the color you want r g b components and an alpha. This is the blending option which is provided. We discussed about this in the previous class and the position 0 1 0. So this would have been finite, if this value was finite this is like a homogenous Cartesian coordinates system and you know in homogenous Cartesian coordinates system when that homogenous factor  $w$  is put to 0 what happens? When the homogenous factor is put to 0 the values are basically pushed to infinity. So that is what will happen with the light zero position here and with 0 1 0 and so this is a position at that particular point and also you have a setting up light type and position.

You can say that I will have a light vector with floating point format gl lights, a first type of light which is of ambient type and the corresponding color borrowed from here, the light zero color is defined here so it is put in as this parameter. So this `GL_LIGHT0` is the label of the first light and you can have either ambient or diffused. You can use `GL_DIFFUSE` for diffuse or `GL_AMBIENT` if you want ambient.

You can also define the same light at a certain position here given. Thus, light zero position with a GL position here and the `GL_LIGHT0` position values will be put at this particular point.



(Refer Slide Time: 35:36)

```
GLfloat light0_colour[] = {1, 1.0, 1, 1.0};
GLfloat light0_position[] = {0.0, 1.0, 0.0, 0.0};

// Setting up light type and position
glLightfv(GL_LIGHT0, GL_AMBIENT,
light0_colour); // use GL_DIFFUSE for diffuse

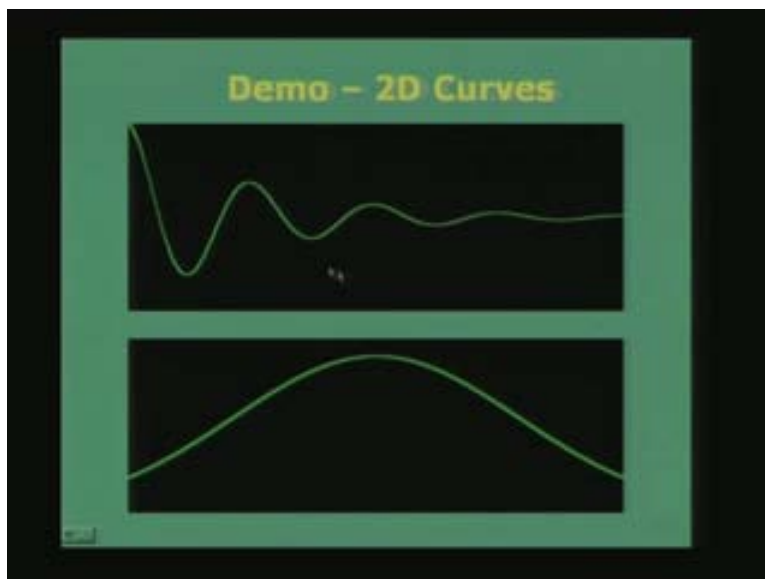
glLightfv(GL_LIGHT0, GL_POSITION,
light0_position);

// Enable light
glEnable(GL_LIGHT0); // can have other lights
glEnable(GL_LIGHTING);
glShadeModel(GL_SMOOTH);
```

You need to enable the light before you start drawing the pictures and that is done by enabling the light.

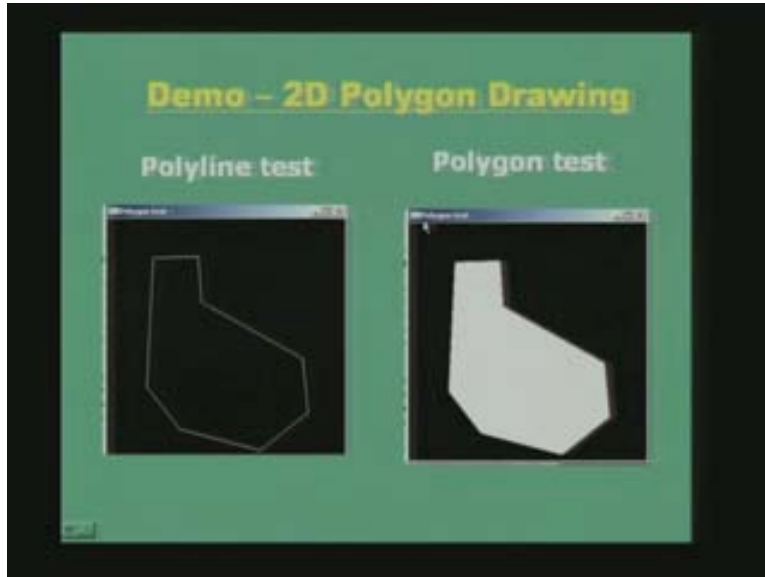
You can have other light sources `GL_LIGHT0` this label means you can have the first light, second, third and so on. And of course `glEnable GL_LIGHTING` you need to put the `glShadeModel, GL_SMOOTH` for smoothing.

(Refer Slide Time: 35:51)



These are the examples of demo 2D curves. We will run the program and go through the sample program for drawing this demo graph at least for this decaying sinusoid, amplitude is exponentially decaying sinusoid.

(Refer Slide Time: 36:07)



We will see this curve and this is another demo for the polyline test and a polygon filling. This is just the polyline, this outer skeleton boundary is drawn, this is a case where the polygon will be filled. We will actually run this program under the Visual C++ Microsoft environment in a XP. And of course this is the color which I was talking about. So this is the section of the code which we have already seen and I will not execute this program.

(Refer Slide Time: 36:46)

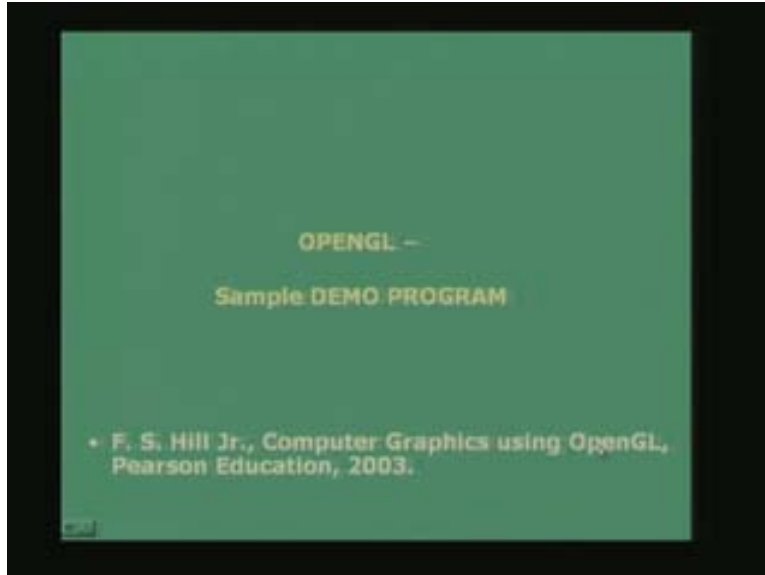


But this is the program output that means you have defined four vertices red, green, blue and white, if you remember 0 0 0 1 1 0 and 1 1 these were the four and you had at the four red, green, blue and white. And this is the interpolated shading which you will have for this particular polygon. It gives you a completely different picture of which you would not have imagined unless you would have interpolated.

Of course when you want to have flat shading all the points inside the polygon will have the same color it could be full red, green and blue or white or any intermediate color which you have to specify and the state model will put the same color if you provide `GL_FLAT` instead of `GL_SMOOTH`. So this is the demo color here.

Let us look in to an OpenGL sample demo program which I have borrowed from the, in fact most of the programs have been taken from the computer graphics book using OpenGL by F. S. Hill under Pearson education 2003.

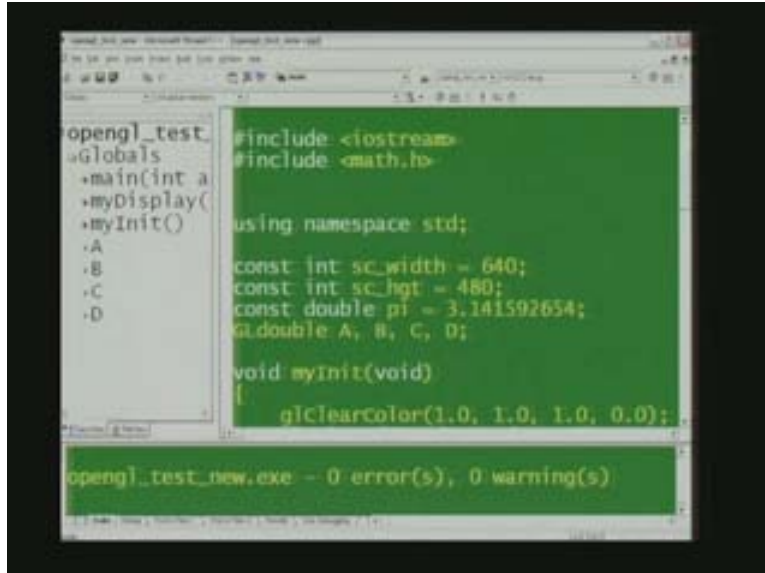
(Refer Slide Time: 37:24)



This is a simple OpenGL program which will draw that sinusoidal exponential indicator form for you. We have just used the Microsoft visual c plus plus environment OpenGL test new .CPP is just a single file.

We have the corresponding external dependencies for IO and the glut. And if you look at the class functions you will have the main and the mydisplay and the myinit and of course a few global variables. So you need to include this gl dot h and the glut dot h depending upon what you want. So this is the higher utility tool kit and the gl dot h. Of course you need a few other header files for C and C plus plus. I am skipping those quite for a windows and IO scheme and Mac and of course we are using the name space and you define your screenwidth and screenheight here as 640 480, 680.

(Refer Slide Time: 38:16)



```
opengl_test
Globals
+main(int a
+myDisplay(
+myInit()
.A
.B
.C
.D

#include <iostream>
#include <math.h>

using namespace std;

const int sc_width = 640;
const int sc_hgt = 480;
const double pi = 3.141592654;
double A, B, C, D;

void myInit(void)
{
    glClearColor(1.0, 1.0, 1.0, 0.0);
}

opengl_test_new.exe -- 0 error(s), 0 warning(s)
```

So I hope you able to see the cursor 640 480 you define pi for the curve and a set of global variables A B C D.

Look at the myinit program now. If you look at the myinit program now there is ClearColor that means you set it up to the foreground white. Then you have to define your color as black, point size is big 3.0, matrix mode is gl projection is in fact not necessary in this in a case we are not projecting anything.

gluortho2D is basically what we define here as a screen width 0, 0 and then of course you have gl double screenheight, 0 to screenwidth you see the parameters gluortho2D. The gluortho2D here you see am discussing this section gluortho2D is from 0 to screenwidth which is here 0 to screenwidth and then 0 to screenheight, that is the gl double both. And these are all defined earlier screenwidth screen height are defined here as you can see a 640 by 480. And then of course you need to define some global constants will see A is width by 4 and height by 2 and then of course this is your mydisplay which will clear the buffer bit and this is your glBegin to glEnd, this section is important.

gl again begin to glEnd which starts from gl points and then as you move here from gl xx to less than 4 with the small increments of .005 you are trying to plot. What are you trying to plot? This is the function, gl double function is the function which you have, e to the power of minus x into a cosine 2pi x function. So it is basically an exponential e to the power minus x function will be a slowly decaying function and then of course you have a cosine or you could have even put a sinusoidal function.

So, if a sinusoid modulated amplitude modulated by an exponential decay so that is what we are trying to see here. That is the function and of course there you use the gl vertex2D command for the plots in the x and scale in terms of y.

This A B C D is used for scaling the corresponding x y coordinates. And of course glBegin to glEnd is the sequence of your code for my display start with a clear and then you flush out the points in the screen. This is the main program. Let us go to the main program here. So this is your main program and that is just an output for you, you do not need this c out, conventionally any sample program starts with the hello world program so I just put that.

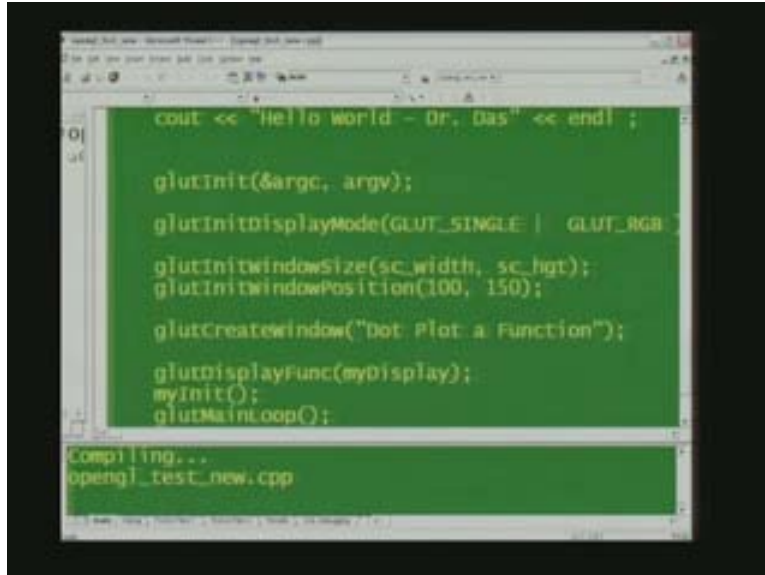
Then you have a glutinit, you have the gluinit first then of course you have the glutinitdisplaymode with single and r g b. Then you have the glutinitwindow size, you specify the certain width and height and you say in which position of the screen you want this, you also specify which position you want in the screen. And this glutcreatewindow will open up a window of size screenwidth by screenheight this is defined as global variables earlier at a position 100, 150 at the top left and the title of that window will be dot plot a function as given here, dot plot a function will be the title we will see that when we run. Then of course you need to provide the glutdisplay.

This is the main section of the code glutdisplayfunction mydisplay will run the mydisplay. What this will run? mydisplay part will be executed here which is the section given here. This was the mydisplay section of the code which will be executed in the main program after it visits the glutdispaly function, it will be enabled and the myinit we have already seen how to use this myinit, it does the global variables and initialization and this is the myinit function.

We have visited this already earlier void myinit function of the code is this section of the code. This section of code is myinit so myinit was defined, my display is also defined. So they are again call back function for display and this is what is used here.

glutmainloop will actually drive after myinit. After the myinit is over the glutmainloop will drive the mydisplay callback functions and keep on processing. It does not have any interaction. These do not have any mousehandler, keyhandler. So let us see what happens when we compile this function.

(Refer Slide Time: 43:04)



```
cout << "Hello world - Dr. Das" << endl ;

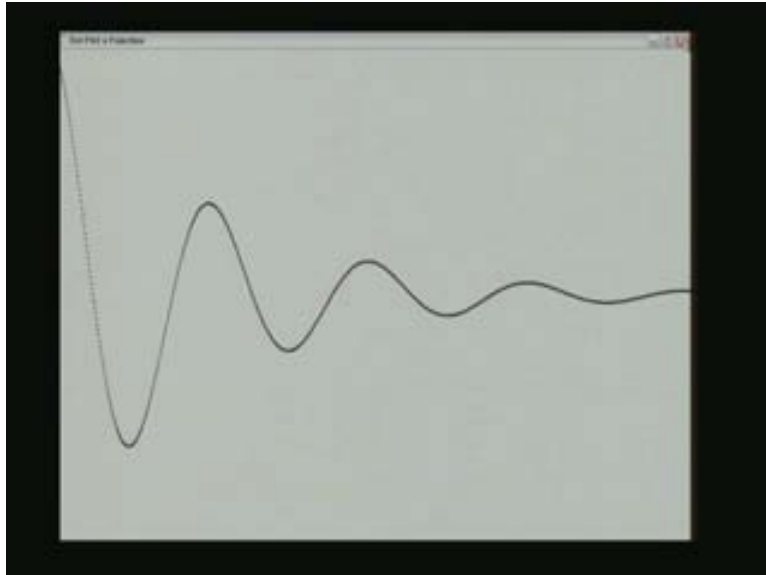
glutInit(&argc, argv);
glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
glutInitWindowSize(sc_width, sc_hgt);
glutInitWindowPosition(100, 150);
glutCreateWindow("Dot Plot a Function");
glutDisplayFunc(myDisplay);
myInit();
glutMainLoop();

Compiling...
opengl_test_new.cpp
```

So let us run it now so this is the function which you have. This is the function which was talking about, the cosine function with its amplitude modulated by an exponential function.

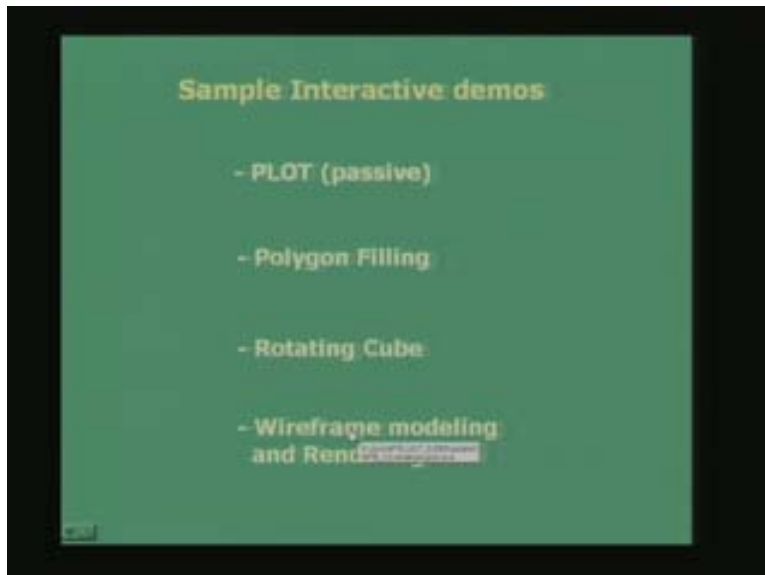
This is the exponential decay of the amplitude as you can see and the oscillation of the cosine and the sine. And you can see the coarseness and the separation of the points is very coarse here, very large here in fact and it is very finite at this particular point. As because you move with  $x$ , the  $y$  variability is very high at the beginning and very less at the small. That is why the interpixel gap does not give a very pleasing effect and this can happen in the plots unless you take some special care.

(Refer Slide Time: 43:40)



We have a set of sample interactive demos, it starts with this plot which we have seen already that is a passive plot.

(Refer Slide Time: 44:00)

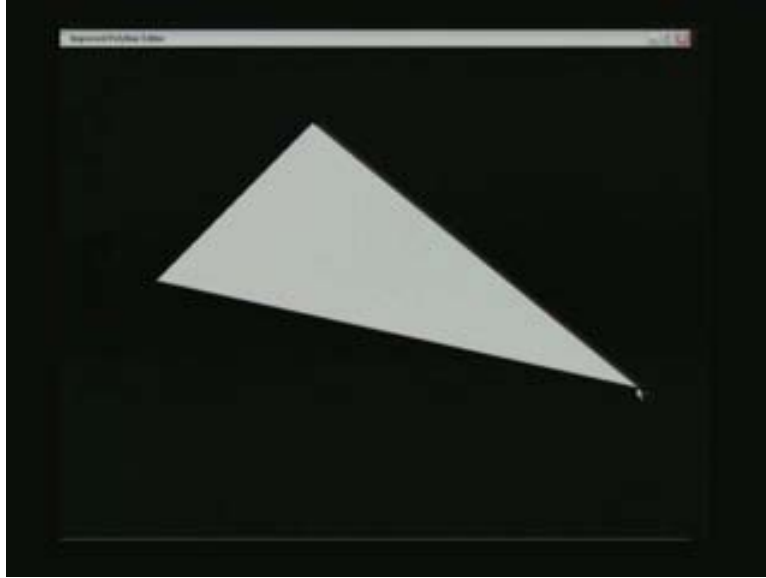


If you run it again it will have the same option that is what you have, it plots that function which we have seen with the example programs. I may not have time to go through the other programs I have three other demos which we will go through and if time permits we might take up one of the example programs at the end.



The next one is a polygon filling algorithm and let us run this program and see. As you see here that is an improved polyline editor.

(Refer Slide Time: 44:31)



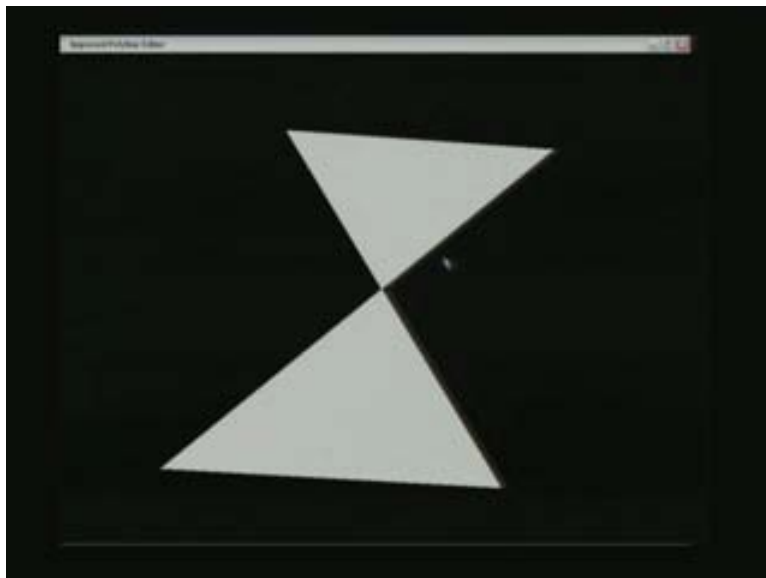
And let us start with the simple triangle. So that is filled, let us run it again. Now you see the lines being drawn. As the mouse moves, it keeps on clicking the vertices and I am trying to construct a polygon as complex as ever one can imagine. I hope for the time being the sample program **for c** borrowed from books but generated with the help of my research students, they have sat with me, it works really well. As you can see here all the interiors, you can visually select any point and do an inside outside test and see that the interior part of the polygon is shaded by a color white and the background being dark.

(Refer Slide Time: 45:45)



Let us draw a concave polygon and see what the result is. That is very simple for you to visualize. This is the one, this is what it will result in and that will also be done. But again the inside outside test is passed with this particular polygon.

(Refer Slide Time: 46:02)



Let us move ahead with another demo which will hopefully be more interesting to you because it will bring in some animation. We will look at this rotating cube example. Well it is a projection of a cube in perspective projection but since you are seeing one face of it then it will look like a square but let me rotate it now.

You can see at least three sides now and that is the maximum which you can see, three sides of the cube and there is an ambient component and the surface is defined as diffused with a corresponding light source.

The light source color is also yellow and as the mouse moves the mousekeyhandler traces the movement of mouse and keeps on rotating depending upon the way the motion has been specified in the program.

Rotations about two different axis depending upon the motion of the mouse. It is just the cube rotating about one or two axis depending upon the motion.

This is rotating about one axis, this is the rotation about the same, this is the rotation in another direction. So that is a simple example of OpenGL. So you have seen three demos and of course I must admit that some of them look trivial except the polygon filling one which is a very interesting example because we have spent lot of time in understanding the concept of polygon filling. Of course we brought the simplest example which used to describe was the plot function. And the rotating cube was the first example of animation.

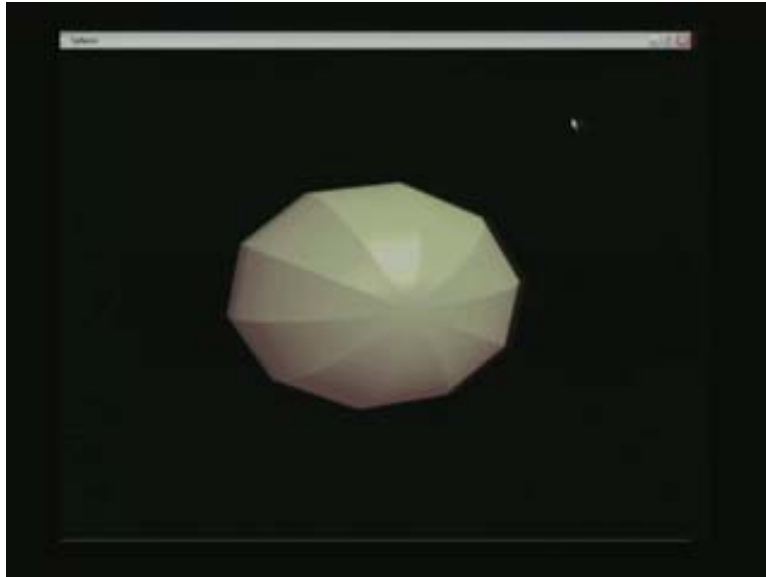
Now a cube is a primitive which is given by OpenGL. OpenGL helps you to define primitives. Cube is one, spheres, cylinder, cone and a very interesting example which you will see now in the next example is that you can even define higher level structures like say a teapot.

We will see that in the next demo for the wireframe modeling and rendering example. Let us see this wireframe modeling and rendering example which I hope you will like and then after seeing this we will probably open up one of these examples to glance through the rotating cube. We will see because that has the examples of projection matrix, shading, lighting unlike the passive example of plot which you have seen.

Let us look at this example let us start by a simple line diagram for a sphere. Although we have seen a cube, cone, cylinder, torus we might see a torus. We will see the sphere and then see the torus. That is the sphere, a sphere with a wireframe diagram non-uniformly sweep representation in two directions. The steps along one direction are much larger than the steps in the other direction purposefully given for visualization otherwise it could have been very dense.

Let us give it flat shading, this sphere, this is what it will look like in terms of flat shading and not smooth. And I am able to rotate it using similar concept as I have did for the cube so that was the example of flat shading.

(Refer Slide Time: 49:20)



Let us see the smooth shading on the sphere. This is the smooth shading on the sphere. Remember, this sphere is smooth along one, it is coarse along the other direction because purposefully for the wireframe it looks like a highly approximated sphere in one direction and smooth in the other.

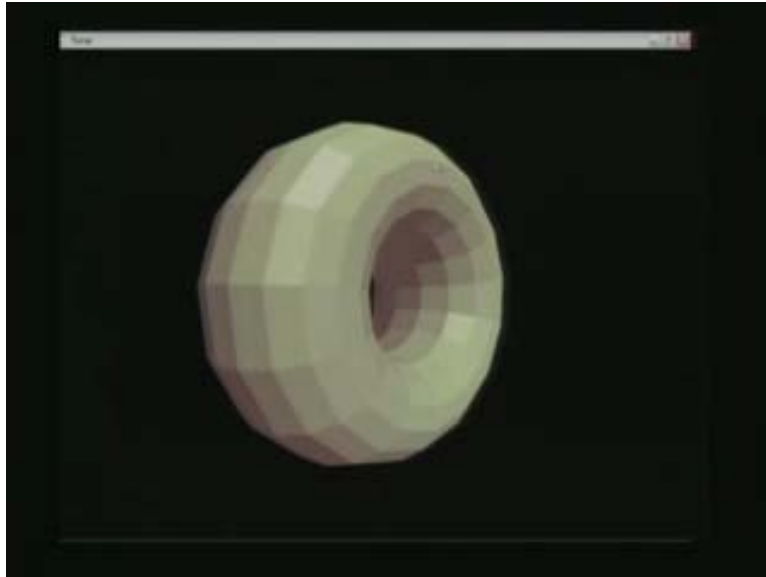
That is if you visualize this in  $r$  theta  $\pi$  domain then in one direction along theta the steps for generating the wireframe tessellation is very fine and it is coarse in the other direction. But you can see the smoothness in terms of the shading here whereas let me restore back the flat shading or we will move on to another structure, say, the torus.

(Refer Slide Time: 50:00)



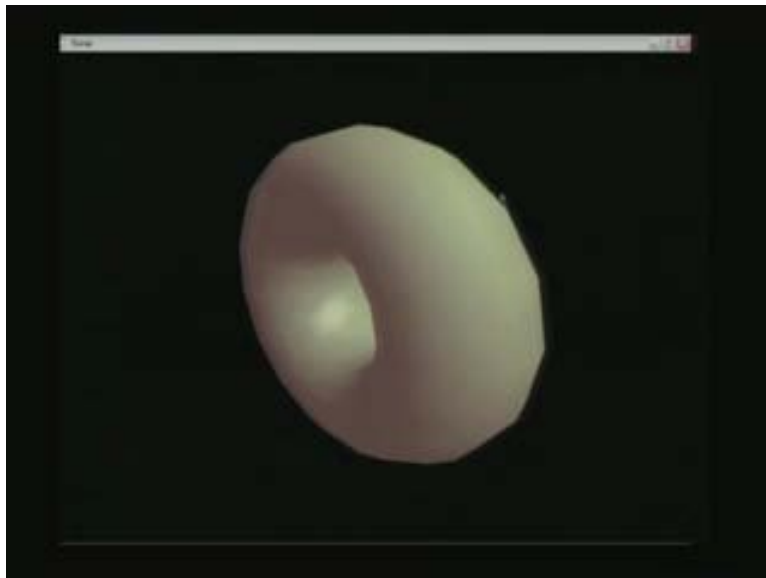
The teapot will be coming. This is the flat shading; we will probably look at the wireframe of this. We will look at the wire frame for the torus, this is the wireframe for the torus. I hope you are able to have visualization. Let us have flat shading on the same wireframe diagram of the torus.

(Refer Slide Time: 50:46)



Let us change to smooth shading from flat for the same torus. You see the change now of the Gouraud shading.

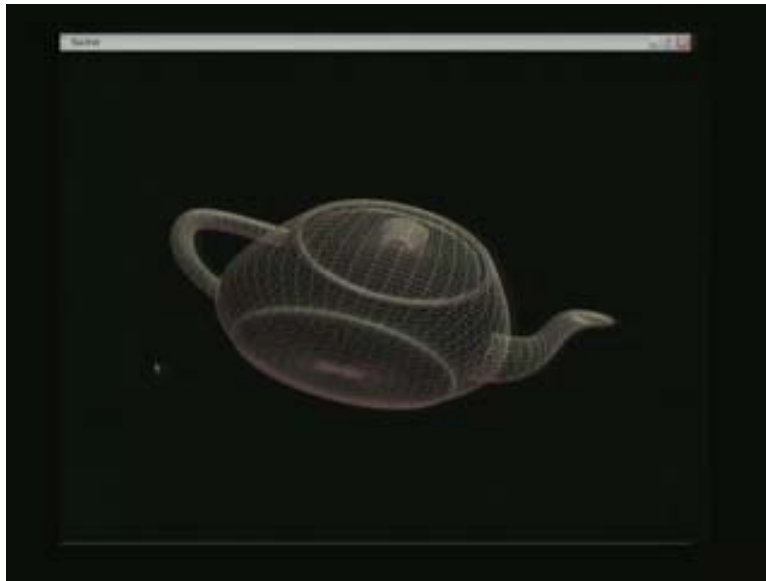
(Refer Slide Time: 51:00)



The surface looks much more shiny and smooth compared to the flat shading. I hope this illustrates much, of course we have seen examples earlier when we discussed rendering and illumination models but along with it animation and a good lighting model. The light is coming from the top right of you towards the object as you can see and you have certain sections which are dark and certain sections which are very bright. This is also a primitive, the torus wireframe is a primitive defined as a feature provided by OpenGL in fact which you can use.

Let us look at an interesting wireframe diagram for the teapot which is also a very good feature that is supported. Let us look at this tea pot that is a wireframe diagram for you. I hope you are able to visualize the structure in spite of the dense wireframe diagram of the tea pot.

(Refer Slide Time: 52:20)



The top lid is coming up so let us provide flat shading for this structure. This is the flat shading model. That means all the polygons and triangles within this particular structure are shaded flat. Let us forget the bottom part of it and let us look at the front as you can see the flatness here now.

You can see the effect of aliasing due to flat shading on the surface of the structure of the tea pot. I hope you have a fair idea of the coarseness. As you see here at each point you can see the aliasing effect of the line, here you can almost see the coarseness of the line once again.

(Refer Slide Time: 53:20)



And let us now put a smooth shading of this. You can see here the smooth structure, you can see the glossy nature of the surface absolutely no aliasing effect or the blocking fact effect which you were seeing due to the flat shading it has completely disappeared. This is the tea pot for you. You can see the glossy nature with no effect whatsoever of the blocking effect on the surface which looks very shining. These are the four different demonstrations which we have. We probably quickly go through an example code may be one within the time frame available to us for the rotating cube.

This is the example for the rotating cube. Well we have a lot of functions, display, drawbox, init main, keyboardhandler and mouseclickhandler, the mouse motion and of course some of the globals here. And then of course we have single file external dependencies glut dot h and the source file is just cube dot c.

So we will just put everything in one program so that although that may not be a good programming from a different perspective but as you see here we are talking of a diffused light source after the initial discussions of the header files we look into a diffused light source with color red component one here 0.8 which is the green and the blue is 0 and the alpha blending is 1.

And then of course we defined a set of surface normals and faces and vertices. I cannot spend much time on that. These are the points which define my cube, vertices, plane normals and then of course the amount of rotation for the alpha theta given by the mouse.

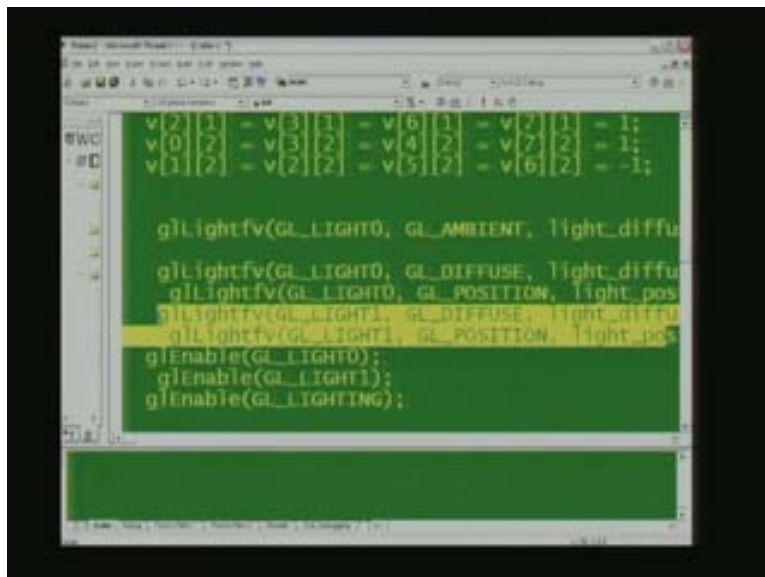
Then in this you have the drawbox function where you have the matrixmode model view, then you have the rotate by an alpha beta about x axis then z axis depending upon the motion, mouse handler will return the alpha and the beta for corresponding rotation movement along x and y and it will rotate by an amount alpha by x axis and then about

the y axis. And then of course this is the one which drives in the drawbox where it defines the faces with gl vertex normals and vertex floating point vectors. And then of course you have the simple display routine here where you clearly have a push and pop of the matrix which may be necessary otherwise you have the drawbox function.

The drawbox function which you see here is the one which you have seen which is just defined here. The drawbox function is defined here up to this point. Then you have the init which defines the light position. There are two lights defined, set of vertices indefinite to define the vertex coordinates here 8 of those and then of course you define the light ambient, I decided to put an ambient light and a diffused light at a certain position.

Two of these GL\_LIGHT0 for one light here and the second light here light 0 and light one and enable both of them.

(Refer Slide Time: 57:11)

A screenshot of a code editor window with a dark background and light-colored text. The code is written in C++ and defines two lights, GL\_LIGHT0 and GL\_LIGHT1. It includes vertex coordinates for a cube, sets ambient and diffuse colors for both lights, and enables both lights and the lighting system. The code is as follows:

```
v[2][1] = v[3][1] = v[6][1] = v[7][1] = 1;
v[0][2] = v[3][2] = v[4][2] = v[7][2] = 1;
v[1][2] = v[2][2] = v[5][2] = v[6][2] = -1;

glLightfv(GL_LIGHT0, GL_AMBIENT, light_diffu);
glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffu);
glLightfv(GL_LIGHT0, GL_POSITION, light_pos);
glLightfv(GL_LIGHT1, GL_DIFFUSE, light_diffu);
glLightfv(GL_LIGHT1, GL_POSITION, light_pos);

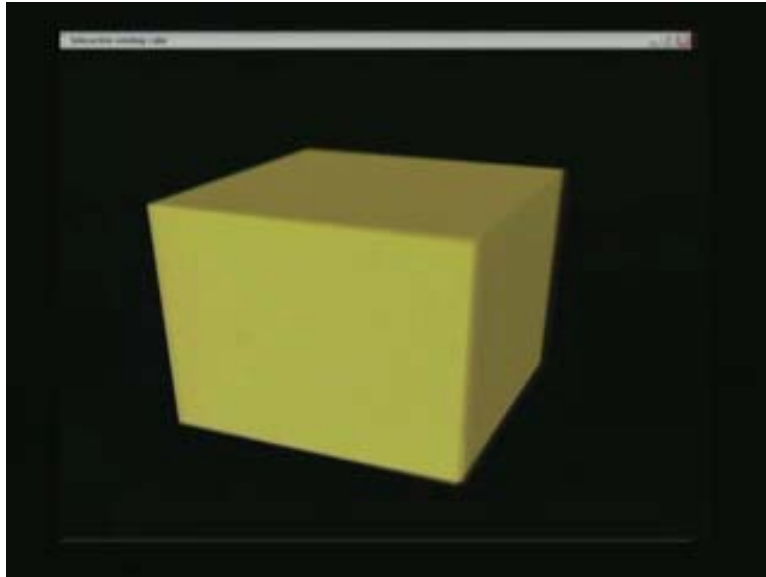
glEnable(GL_LIGHT0);
glEnable(GL_LIGHT1);
glEnable(GL_LIGHTING);
```

Once that is done depth test is enabled and then you define a look up vector, define eye, where is your eye? At 0 0 5 center looking at 0 0 and then of course you have a look up vector as well.

Then you have the glMatrix perspective here, the model view and you translate it and position it somewhere. Of course now you have to define your keyhandler, the mouse click handler with its mouse down or up and then based on the mouse motion you define the amount of increment you need to give it to the angle alpha and beta for the amount of rotation. And if you run that this is the same box which you have seen in the demo earlier.



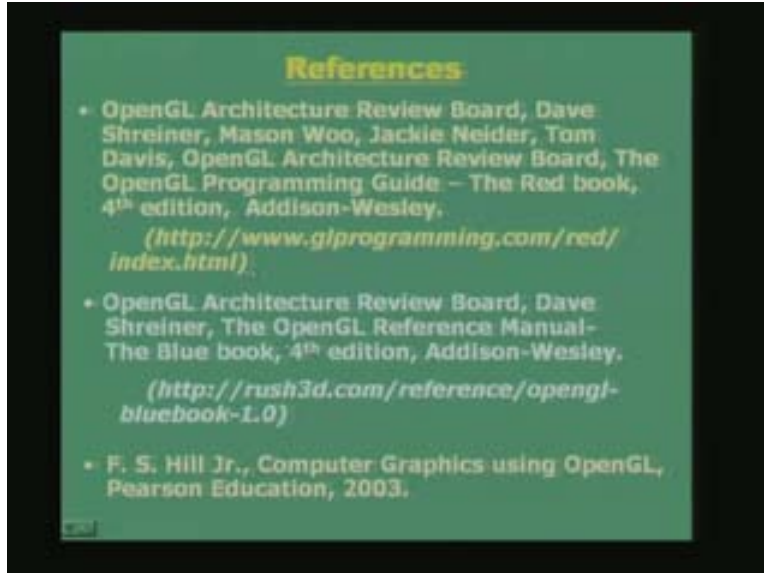
(Refer Slide Time: 57:58)



So I am glad that most of this in fact all of the demos ran very well. I am thankful to my students I would refer their names once again Mr. Vinod and Deepthi they have really worked hard to get this demos done although some of them were quietly well and simple obtained from books. But to ensure that they are working on a system and make a system OpenGL ready is not a very trivial task in certain cases.

I give a list of reference books which you must obtain to use programs, obtain, know about OpenGL, know about OpenGL features and also get the syntax of the functions and data types etc about the red book. This is the red book by Addison Wesley you can download it from this website.

(Refer Slide Time: 58:51)



Hopefully this will be available when you are viewing these lectures and also you have them in the blue book of OpenGL as given here. Of course if the sites or the URL changes please go to some search engines and try the red book and blue book here the blue book and the red book for OpenGL.

And if you are going to get a link and of course most of the contents and programs have been borrowed from F.S. Hill Junior. So, that brings us to the end of the lectures on graphics programming using OpenGL. Thanks a lot for the sequence and your patient hearing and a sequence of lectures starting from the very beginning of introduction to display devices and all different types of transformations clipping, drawing, rendering, shading, VSD and finally the industrial standards and some examples of coding of OpenGL.

I hope you have enjoyed the last lecture today and that will trigger the interest in you to learn about OpenGL and use them and slowly become experts as well, thank you very much.