**INTRODUCTION**

Computer Graphics is concerned with all aspects of producing pictures or images using computer. A particular graphics software system called OpenGL, which has become a widely accepted standard for developing graphics applications.

The applications of computer graphics in some of the major areas are as follows:

1. Display of information
2. Design
3. Simulation and Animation
4. User Interfaces

OpenGL is a software interface to graphics hardware. This interface consists of about 150 distinct commands that are used to specify the objects and operation needed to produce interactive three-dimensional applications.

**HISTORY**

In the 1980s, developing software that could function with a wide range of graphics hardware was a real challenge. Software developers wrote custom interfaces and drivers for each piece of hardware. This was expensive and resulted in much duplication of effort.

By the early 1990s, **Silicon Graphics** (SGI) was a leader in 3D graphics for workstations. Their **IRIS GL** API was considered the state of the art and became the de facto industry standard, overshadowing the open standards-based [**PHIGS**](http://en.wikipedia.org/wiki/PHIGS). This was because IRIS GL was considered easier to use, and because it supported [**immediate mode**](http://en.wikipedia.org/wiki/Immediate_mode) rendering. By contrast, [**PHIGS**](http://en.wikipedia.org/wiki/PHIGS) was considered difficult to use and outdated in terms of functionality.

SGI's competitors (including **Sun Microsystems**, **Hewlett-Packard** and [**IBM**](http://en.wikipedia.org/wiki/IBM)) were also able to bring to market 3D hardware, supported by extensions made to the [**PHIGS**](http://en.wikipedia.org/wiki/PHIGS)standard. This in turn caused SGI market share to weaken as more 3D graphics hardware suppliers entered the market. In an effort to influence the market, SGI decided to turn the IrisGL API into an open standard.

SGI considered that the IrisGL API itself wasn't suitable for opening due to licensing and patent issues. Also, the IrisGL had API functions that were not relevant to 3D graphics. For example, it included a windowing, keyboard and mouse API, in part because it was developed before the [**X Window System**](http://en.wikipedia.org/wiki/X_Window_System)and Sun's [**News**](http://en.wikipedia.org/wiki/NeWS)systems were developed.

In addition, SGI had a large number of software customers; by changing to the OpenGL API they planned to keep their customers locked onto SGI (and IBM) hardware for a few years while market support for OpenGL matured. Meanwhile, SGI would continue to try to maintain their customers tied to SGI hardware by developing the advanced and proprietary [**Iris Inventor**](http://en.wikipedia.org/wiki/Open_Inventor) and [Iris Performer](http://en.wikipedia.org/wiki/Iris_Performer) programming APIs.As a result, SGI released the **OpenGL** standard.

The OpenGL standardized access to hardware, and pushed the development responsibility of hardware interface programs, sometimes called [**device drivers**](http://en.wikipedia.org/wiki/Device_driver)**,** to hardware manufacturers and delegated windowing functions to the underlying operating system. With so many different kinds of graphic hardware, getting them all to speak the same language in this way had a remarkable impact by giving software developers a higher level platform for 3D-software development.

In 1992, SGI led the creation of the OpenGL architectural review board (OpenGL ARB), the group of companies that would maintain and expand the OpenGL specification for years to come. OpenGL evolved from (and is very similar in style to) SGI's earlier 3D interface, *IrisGL*. One of the restrictions of IrisGL was that it only provided access to features supported by the underlying hardware. If the graphics hardware did not support a feature, then the application could not use it. OpenGL overcame this problem by providing support in software for features unsupported by hardware, allowing Applications to use advanced graphics on relatively low-powered systems. In 1994 SGI played with the idea of releasing something called "[**OpenGL++**](http://en.wikipedia.org/wiki/OpenGL%2B%2B)" which included elements such as a scene-graph API (presumably based on their [Performer](http://en.wikipedia.org/wiki/Performer_%28Computer_Graphics_API%29) technology). The specification was circulated among a few interested parties – but never turned into a product.

[**Microsoft**](http://en.wikipedia.org/wiki/Microsoft) released Direct3D in 1995, which would become the main competitor of OpenGL. On 17 December 1997, Microsoft and SGI initiated the Fahrenheit project, which was a joint effort with the goal of unifying the OpenGL and Direct3D interfaces (and adding a scene-graph API too). In 1998 Hewlett-Packard joined the project. It initially showed some promise of bringing order to the world of interactive 3D computer graphics APIs, but on account of financial constraints at SGI, strategic reasons at Microsoft, and general lack of industry support, it was abandoned in 1999. OpenGL releases are backward compatible.

**ABOUT OPENGL**

OpenGL is a software interface to graphics hardware. This interface consists of about 150 distinct commands that you use to specify the objects and operations needed to produce interactive three-dimensional applications.

OpenGL is designed as a streamlined, hardware-independent interface to be implemented on many different hardware platforms. To achieve these qualities, no commands for performing windowing tasks or obtaining user input are included in OpenGL; instead, we must work through whatever windowing system controls the particular hardware we are using. Similarly, OpenGL doesn’t provide high-level commands for describing models of three-dimensional objects. Such commands might allow you to specify relatively complicated shapes such as automobiles, parts of the body, airplanes, or molecules. With OpenGL, we must build up our desired model from a small set of *geometric primitives* - points, lines, and polygons. The OpenGL Utility Library (GLU) provides many of the modeling features, such as quadric surfaces and NURBS curves and surfaces. GLU is a standard part of every OpenGL implementation. Also, there is a higher-level, object-oriented toolkit, Open Inventor, which is built atop OpenGL, and is available separately for many implementations of OpenGL.

**CONCEPTS AND PRINCIPLES OF OPENGL**

The color plates give you an idea of the kinds of things you can do with the OpenGL graphics system. The following list briefly describes the major graphics operations which OpenGL performs to render an image on the screen.

**1.** Construct shapes from geometric primitives, thereby creating mathematical descriptions of objects. (OpenGL considers points, lines, polygons, images, and bitmaps to be primitives.)

**2.** Arrange the objects in three-dimensional space and select the desired vantage point for viewing the composed scene.

**3.** Calculate the color of all the objects. The color might be explicitly assigned by the application, determined from specified lighting conditions, obtained by pasting a texture onto the objects, or some combination of these three actions.

**4.** Convert the mathematical description of objects and their associated color information to pixels on During these stages, OpenGL might perform other operations, such as eliminating parts of objects that are hidden by other objects. In addition, after the scene is rasterized but before it’s drawn on the screen, you can perform some operations on the pixel data if you want.

In some implementations (such as with the X Window System), OpenGL is designed to work even if the computer that displays the graphics you create isn’t the computer that runs your graphics program. This might be the case if you work in a networked computer environment where many computers are connected to one another by a digital network. In this situation, the computer on which your program runs and issues OpenGL drawing commands is called the client, and the computer that receives those commands and performs the drawing is called the server. The format for transmitting OpenGL commands (called the *protocol*) from the client to the server is always the same, so OpenGL programs can work across a network even if the client and server are different kinds of computers. If an OpenGL program isn’t running across a network, then there’s only one computer, and it is both the client and the server.

**Display Lists**

All data, whether it describes geometry or pixels, can be saved in a display list for current or later use. (The alternative to retaining data in a display list is processing the data immediately - also known as immediate mode.) When a display list is executed, the retained data is sent from the display list just as if it were sent by the application in immediate mode lists.)

**Evaluators**

All geometric primitives are eventually described by vertices. Parametric curves and surfaces may be initially described by control points and polynomial functions called basis functions. Evaluators provide a method to derive the vertices used to represent the surface from the control points. The method is a polynomial mapping, which can produce surface normal, texture coordinates, colors, and spatial coordinate values from the control points.

# Per-Vertex Operations

For vertex data, next is the “per-vertex operations” stage, which converts the vertices into primitives.Some vertex data (for example, spatial coordinates) are transformed by 4 x 4 floating-point matrices.Spatial coordinates are projected from a position in the 3D world to a position on your screen. If advanced features are enabled, this stage is even busier. If texturing is used, texture coordinates may be generated and transformed here. If lighting is enabled, the lighting calculations are performed using the transformed vertex, surface normal, light source position, material properties, and other lighting information to produce a color value.

**Primitive Assembly**

Clipping, a major part of primitive assembly, is the elimination of portions of geometry which fall outside a half-space, defined by a plane. Point clipping simply passes or rejects vertices; line or polygon clipping can add additional vertices depending upon how the line or polygon is clipped. In some cases, this is followed by perspective division, which makes distant geometric objects appear smaller than closer objects. Then viewport and depth (z coordinate) operations are applied.

**Pixel Operations**

While geometric data takes one path through the OpenGL rendering pipeline, pixel data takes a different route. Pixels from an array in system memory are first unpacked from one of a variety of formats into the proper number of components. Next the data is scaled, biased, and processed by a pixel map. The results are clamped and then either written into texture memory or sent to the rasterization step. If pixel data is read from the frame buffer, pixel-transfer operations (scale, bias, mapping, and clamping) are performed. Then these results are packed into an appropriate format and returned to an array in system memory.

**Texture Assembly**

An OpenGL application may wish to apply texture images onto geometric objects to make them look more realistic. If several texture images are used, it’s wise to put them into texture objects so that you can easily switch among them. Some OpenGL implementations may have special resources to accelerate texture performance. There may be specialized, high-performance texture memory. If this memory is available, the texture objects may be prioritized to control the use of this limited and valuable resource.

# Rasterization

Rasterization is the conversion of both geometric and pixel data into fragments. Each fragment square corresponds to a pixel in the frame buffer. Line and polygon stipples, line width, point size, shading model, and coverage calculations to support antialiasing are taken into consideration as vertices are connected into lines or the interior pixels are calculated for a filled polygon. Color and depth values are assigned for each fragment square. Fragment Operations Before values are actually stored into the frame buffer, a series of operations are performed that may alter or even throw out fragments. All these operations can be enabled or disabled. The first operation which may be encountered is texturing, where a Texel (texture element) is generated from texture memory for each fragment and applied to the fragment. Then fog calculations may be applied, followed by the scissor test, the alpha test, the stencil test, and the depth-buffer test (the depth buffer is for hidden-surface removal).

**OPENGL RELATED LIBRARIES**

OpenGL provides a powerful but primitive set of rendering commands, and all higher-level drawing must be done in terms of these commands. Also, OpenGL programs have to use the underlying mechanisms of the windowing system. A number of libraries exist to allow you to simplify your programming tasks, including the following: The OpenGL Utility Library (GLU) contains several routines that use lower-level,OpenGL commands to perform such tasks as setting up matrices for specific viewing orientations and projections, performing polygon tessellation, and rendering surfaces. This library is provided as part of every OpenGL implementation. GLU routines use the prefix **glu**.

For every window system, there is a library that extends the functionality of that window system to support OpenGL rendering. For machines that use the X Window System, the OpenGL Extension to the X Window System (GLX) is provided as an adjunct to OpenGL. GLX routines use the prefix **glX**. For Microsoft Windows, the WGL routines provide the Windows to OpenGL interface. The OpenGL Utility Toolkit (GLUT) is a window system-independent toolkit, written by Mark Kilgard, to hide the complexities of differing window system APIs.

Open Inventor is an object-oriented toolkit based on OpenGL which provides objects and methods for creating interactive three-dimensional graphics applications. Open Inventor, which is written in C++, provides prebuilt objects and a built-in event model for user interaction, high-level application components for creating and editing three-dimensional scenes, and the ability to print objects and exchange data in other graphics formats. Open Inventor is separate from OpenGL.

**WINDOW MANAGEMENT IN OPENGL**

Five routines perform tasks necessary to initialize a window.

* **glutInit** (int \**argc*, char \*\**argv*) initializes GLUT and processes any command line arguments (for X, this would be options like -display and -geometry). **glutInit ()** should be called before any other GLUT routine.
* **glutInitDisplayMode (**unsigned int *mode*) specifies whether to use an *RGBA* or color-index color model. You can also specify whether you want a single- or double-buffered window. (If you’re working in color-index mode, you’ll want to load certain colors into the color map; use
* **glutSetColor ()** to do this. Finally, you can use this routine to indicate that you want the window to have an associated depth, stencil, and/or accumulation

buffer. For example, if you want a window with double buffering, the RGBA color model, and a depth buffer.

* g**lutInitDisplayMode** (*GLUT\_DOUBLE | GLUT\_RGB | GLUT\_DEPTH*). **glutInitWindowPosition** (int *x*, int *y*) specifies the screen location for the upper-left corner of your window.
* **glutInitWindowSize (**int *width*, int *size*) specifies the size, in pixels, of your window.
* Int **glutCreateWindow** (char \**string*) creates a window with an OpenGL context. It returns a unique identifier for the new window. Be warned

**glutMainLoop ()** is called, the window is not yet displayed.

# THE CALLBACKS

**glutDisplayFunc** (void (\* *func*) (void)) is the first and most important event callback function you will see. Whenever GLUT determines the contents of the window need to be redisplayed, the callback function registered by **glutDisplayFunc ()** is executed. If your program changes the contents of the window, sometimes you will have to call **glutPostRedisplay** (void), which gives **glutMainLoop ()** a nudge to call the registered display callback at its next opportunity.

## RUNNING THE PROGRAM

The very last thing you must do is call glutMainLoop (void). All windows that have been created are now shown, and rendering to those windows is now effective. Event processing begins, and the registered display callback is triggered. Once this loop is entered, it is never exited!. Note the restructuring of the code. To maximize efficiency, operations that need only be called once (setting the background color and coordinate system) are now in a procedure called init(). Operations to render (and possibly re-render) the scene are in the display() procedure, which is the registered GLUT display callback.

1. **Program to display point.**

#include <glfw3.h>

#include <glut.h>

void init(void)

{

glClearColor(0.0, 0.0, 0.0, 0.0); // Set display-window color to white.

glMatrixMode(GL\_PROJECTION); // Set projection parameters.

gluOrtho2D(0.0, 200.0, 0.0, 250.0);

}

void lineSegment(void)

{

glClear(GL\_COLOR\_BUFFER\_BIT); // Clear display window.

glColor3f(1.0, 0.0, 0.0); // Set line segment color to green.

glBegin(GL\_POINTS);

glVertex2i(50, 100);

glVertex2i(75, 150);

glVertex2i(100, 200);

glEnd();

glFlush(); // Process all OpenGL routines as quickly as possible.

}

int main(int argc, char\*\* argv)

{

glutInit(&argc, argv); // Initialize GLUT.

glutInitDisplayMode(GLUT\_SINGLE | GLUT\_RGB); // Set display mode.

glutInitWindowPosition(50, 100); // Set top-left display-window position.

glutInitWindowSize(400, 300); // Set display-window width and height.

glutCreateWindow("An Example OpenGL Program"); // Create display window.

init(); // Execute initialization procedure.

glutDisplayFunc(lineSegment); // Send graphics to display window.

glutMainLoop(); // Display everything and wait.

return 0;

}

1. **Program to display points.**

#include <glfw3.h>

#include <glut.h>

void init(void)

{

glClearColor(0.0, 0.0, 0.0, 0.0); // Set display-window color to white.

glMatrixMode(GL\_PROJECTION); // Set projection parameters.

gluOrtho2D(0.0, 200.0, 0.0, 250.0);

}

void lineSegment(void)

{

glClear(GL\_COLOR\_BUFFER\_BIT); // Clear display window.

glColor3f(1.0, 0.0, 0.0); // Set line segment color to green.

int point1[] = { 50, 100 };

int point2[] = { 75, 150 };

int point3[] = { 100, 200 };

glBegin(GL\_POINTS);

glVertex2iv(point1);

glVertex2iv(point2);

glVertex2iv(point3);

glEnd();

glFlush(); // Process all OpenGL routines as quickly as possible.

}

int main(int argc, char\*\* argv)

{

glutInit(&argc, argv); // Initialize GLUT.

glutInitDisplayMode(GLUT\_SINGLE | GLUT\_RGB); // Set display mode.

glutInitWindowPosition(50, 100); // Set top-left display-window position.

glutInitWindowSize(400, 300); // Set display-window width and height.

glutCreateWindow("An Example OpenGL Program"); // Create display window.

init(); // Execute initialization procedure.

glutDisplayFunc(lineSegment); // Send graphics to display window.

glutMainLoop(); // Display everything and wait.

return 0;

}

1. **Program to display line.**

#include <gl/glut.h>

void init(void)

{

glClearColor(1.0, 1.0, 1.0, 0.0); // Set display-window color to white.

glMatrixMode(GL\_PROJECTION); // Set projection parameters.

gluOrtho2D(0.0, 200.0, 0.0, 150.0);

}

void lineSegment(void)

{

glClear(GL\_COLOR\_BUFFER\_BIT); // Clear display window.

glColor3f(1.0, 0.4, 0.2); // Set line segment color to green.

glBegin(GL\_TRIANGLE\_STRIP);

glVertex2i(180, 15); // Specify line-segment geometry.

glVertex2i(10, 145);

glEnd();

glFlush(); // Process all OpenGL routines as quickly as possible.

}

int main(int argc, char\*\* argv)

{

glutInit(&argc, argv); // Initialize GLUT.

glutInitDisplayMode(GLUT\_SINGLE | GLUT\_RGB); // Set display mode.

glutInitWindowPosition(50, 100); // Set top-left display-window position.

glutInitWindowSize(400, 300); // Set display-window width and height.

glutCreateWindow("An Example OpenGL Program"); // Create display window.

init(); // Execute initialization procedure.

glutDisplayFunc(lineSegment); // Send graphics to display window.

glutMainLoop(); // Display everything and wait.

return 0;

}

1. **Program to display Line Loop**

#include <glut.h>

void init(void)

{

glClearColor(0.0, 0.0, 0.0, 0.0); // Set display-window color to white.

glMatrixMode(GL\_PROJECTION); // Set projection parameters.

gluOrtho2D(-2.0, 2.0, -2.0, 2.0);

}

void lineSegment(void)

{

glClear(GL\_COLOR\_BUFFER\_BIT); // Clear display window.

glColor3f(1.0, 0.0, 0.0); // Set line segment color to green.

glBegin(GL\_LINE\_LOOP);

glVertex3f(0.2, 0.2, 0.0);

glVertex3f(0.8, 0.2, 0.0);

glVertex3f(0.2, 0.5, 0.0);

glVertex3f(0.8, 0.5, 0.0);

glVertex3f(0.2, 0.8, 0.0);

glVertex3f(0.8, 0.8, 0.0);

glEnd();

glFlush(); // Process all OpenGL routines as quickly as possible.

}

int main(int argc, char\*\* argv)

{

glutInit(&argc, argv); // Initialize GLUT.

glutInitDisplayMode(GLUT\_SINGLE | GLUT\_RGB); // Set display mode.

glutInitWindowPosition(50, 100); // Set top-left display-window position.

glutInitWindowSize(400, 300); // Set display-window width and height.

glutCreateWindow("Point Program"); // Create display window.

init(); // Execute initialization procedure.

glutDisplayFunc(lineSegment); // Send graphics to display window.

glutMainLoop(); // Display everything and wait.

return 0;

}

1. **Program to Implement DDA Line algorithm.**

#include <stdio.h>

#include <math.h>

#include <GL/glut.h>

#include <iostream>

using namespace std;

double X1, Y1, X2, Y2;

float round\_value(float v)

{

return floor(v + 0.5);

}

void LineDDA(void)

{

double dx = (X2 - X1);

double dy = (Y2 - Y1);

double steps;

float xInc, yInc, x = X1, y = Y1;

/\* Find out whether to increment x or y \*/

steps = (abs(dx) > abs(dy)) ? (abs(dx)) : (abs(dy));

xInc = dx / (float)steps;

yInc = dy / (float)steps;

/\* Clears buffers to preset values \*/

glClear(GL\_COLOR\_BUFFER\_BIT);

/\* Plot the points \*/

glBegin(GL\_POINTS);

/\* Plot the first point \*/

glVertex2d(x, y);

int k;

/\* For every step, find an intermediate vertex \*/

for (k = 0; k < steps; k++)

{

x += xInc;

y += yInc;

/\* printf("%0.6lf %0.6lf\n",floor(x), floor(y)); \*/

glVertex2d(round\_value(x), round\_value(y));

}

glEnd();

glFlush();}

void Init()

{

/\* Set clear color to white \*/

glClearColor(1.0, 1.0, 1.0, 0);

/\* Set fill color to black \*/

glColor3f(0.0, 0.0, 0.0);

/\* glViewport(0 , 0 , 640 , 480); \*/

/\* glMatrixMode(GL\_PROJECTION); \*/

/\* glLoadIdentity(); \*/

gluOrtho2D(0, 640, 0, 480);

}

int main(int argc, char\*\* argv)

{

cout<<"Enter two end points of the line to be drawn:\n";

cout<<"\nEnter Point1( X1 , Y1):\n";

cin >> X1;

cin >> Y1;

cout<<"\nEnter Point1( X2 , Y2):\n";

cin>> X2;

cin >> Y2;

/\* Initialise GLUT library \*/

glutInit(&argc, argv);

/\* Set the initial display mode \*/

glutInitDisplayMode(GLUT\_SINGLE | GLUT\_RGB);

/\* Set the initial window position and size \*/

glutInitWindowPosition(0, 0);

glutInitWindowSize(640, 480);

/\* Create the window with title "DDA\_Line" \*/

glutCreateWindow("DDA\_Line");

/\* Initialize drawing colors \*/

Init();

/\* Call the displaying function \*/

glutDisplayFunc(LineDDA);

/\* Keep displaying untill the program is closed \*/

glutMainLoop();

}

1. **Program to implement Cylindrical and parallelepiped structure.**

#include <GL/glut.h>

#include <math.h>

#include <stdio.h>

void draw\_pixel(GLint cx, GLint cy)

{ glColor3f(1.0,0.0,0.0);

glBegin(GL\_POINTS);

glVertex2i(cx,cy);

glEnd();

}

void plotpixels(GLint h, GLint k, GLint x, GLint y)

{

draw\_pixel(x+h,y+k);

draw\_pixel(-x+h,y+k);

draw\_pixel(x+h,-y+k);

draw\_pixel(-x+h,-y+k);

draw\_pixel(y+h,x+k);

draw\_pixel(-y+h,x+k);

draw\_pixel(y+h,-x+k);

draw\_pixel(-y+h,-x+k);

}

void Circle\_draw(GLint h, GLint k, GLint r)

{

GLint d = 1-r, x=0, y=r;

while(y > x)

{

plotpixels(h,k,x,y);

if(d < 0) d+=2\*x+3;

else

{d+=2\*(x-y)+5;

--y;

}

++x;

}

plotpixels(h,k,x,y);

}

void Cylinder\_draw()

{

GLint xc=100, yc=100, r=50;

GLint i,n=50;

for(i=0;i<n;i+=3)

{

Circle\_draw(xc,yc+i,r);

}

}

void parallelepiped(int x1, int x2,int y1, int y2, int y3, int y4)

{

glColor3f(0.0, 0.0, 1.0);

glPointSize(2.0);

glBegin(GL\_LINE\_LOOP);

glVertex2i(x1,y1);

glVertex2i(x2,y3);

glVertex2i(x2,y4);

glVertex2i(x1,y2);

glEnd();

}

void parallelepiped\_draw()

{

int x1=200,x2=300,y1=100,y2=175,y3=100,y4=175;

GLint i,n=40;

for(i=0;i<n;i+=2)

{

parallelepiped(x1+i,x2+i,y1+i,y2+i,y3+i,y4+i);

}

}

void init(void)

{

glClearColor(1.0,1.0,1.0,0.0);

glMatrixMode(GL\_PROJECTION);

gluOrtho2D(0.0,400.0,0.0,300.0);

}

void display(void)

{ glClear(GL\_COLOR\_BUFFER\_BIT);

glColor3f(1.0,0.0,0.0);)

glPointSize(2.0);

Cylinder\_draw();

parallelepiped\_draw();

glFlush();

}

void main(int argc, char \*\*argv)

{ glutInit(&argc,argv); // Initialize GLUT

glutInitDisplayMode(GLUT\_SINGLE | GLUT\_RGB); // Set Display mode

glutInitWindowPosition(50,50); // Set top left window position

glutInitWindowSize(400,300); // Set Display window width and height

glutCreateWindow("Cylinder and parallelePiped Display by extruding

Circle and Quadrilaterl ");

init();

glutDisplayFunc(display); // Send the graphics to Display Window

glutMainLoop();

}

1. **Program to implement Liang-Barsky line clipping algorithm.**

#include <stdio.h>

#include <GL/glut.h>

double xmin=50,ymin=50, xmax=100,ymax=100;

double xvmin=200,yvmin=200,xvmax=300,yvmax=300;

int cliptest(double p, double q, double \*t1, double \*t2)

{

double t=q/p;

if(p < 0.0) // potentially enry point, update te

{

if( t > \*t1) \*t1=t;

if( t > \*t2) return(false); // line portion is outside

}

else

if(p > 0.0) // Potentially leaving point, update tl

{

if( t < \*t2) \*t2=t;

if( t < \*t1) return(false); // line portion is outside

}

else

if(p == 0.0)

{

if( q < 0.0) return(false); // line parallel to edge but outside

}

return(true);

}

void LiangBarskyLineClipAndDraw (double x0, double y0,double x1, double y1)

{

double dx=x1-x0, dy=y1-y0, te=0.0, tl=1.0;

if(cliptest(-dx,x0-xmin,&te,&tl))

if(cliptest(dx,xmax-x0,&te,&tl))

if(cliptest(-dy,y0-ymin,&te,&tl))

if(cliptest(dy,ymax-y0,&te,&tl))

{

if( tl < 1.0 )

{

x1 = x0 + tl\*dx;

y1 = y0 + tl\*dy;

}

if( te > 0.0 )

{ x0 = x0 + te\*dx;

y0 = y0 + te\*dy;

}

double sx=(xvmax-xvmin)/(xmax-xmin);

double sy=(yvmax-yvmin)/(ymax-ymin);

double vx0=xvmin+(x0-xmin)\*sx;

double vy0=yvmin+(y0-ymin)\*sy;

double vx1=xvmin+(x1-xmin)\*sx;

double vy1=yvmin+(y1-ymin)\*sy;

glColor3f(1.0, 0.0, 0.0);

glBegin(GL\_LINE\_LOOP);

glVertex2f(xvmin, yvmin);

glVertex2f(xvmax, yvmin);

glVertex2f(xvmax, yvmax);

glVertex2f(xvmin, yvmax);

glEnd();

glColor3f(0.0,0.0,1.0);

glBegin(GL\_LINES);

glVertex2d (vx0, vy0);

glVertex2d (vx1, vy1);

glEnd();

}

}

void display()

{

double x0=60,y0=20,x1=80,y1=120;

glClear(GL\_COLOR\_BUFFER\_BIT);

glColor3f(1.0,0.0,0.0);

glBegin(GL\_LINES);

glVertex2d (x0, y0);

glVertex2d (x1, y1);

glEnd();

glColor3f(0.0, 0.0, 1.0);

glBegin(GL\_LINE\_LOOP);

glVertex2f(xmin, ymin);

glVertex2f(xmax, ymin);

glVertex2f(xmax, ymax);

glVertex2f(xmin, ymax);

glEnd();

LiangBarskyLineClipAndDraw(x0,y0,x1,y1);

glFlush();

}

void myinit()

{

glClearColor(1.0,1.0,1.0,1.0);

glColor3f(1.0,0.0,0.0);

glPointSize(1.0);

glMatrixMode(GL\_PROJECTION);

glLoadIdentity();

gluOrtho2D(0.0,499.0,0.0,499.0);

}

void main(int argc, char\*\* argv)

{

glutInit(&argc,argv);

glutInitDisplayMode(GLUT\_SINGLE|GLUT\_RGB);

glutInitWindowSize(500,500);

glutInitWindowPosition(0,0);

glutCreateWindow("Liang Barsky Line Clipping Algorithm");

glutDisplayFunc(display);

myinit();

glutMainLoop();

}

1. **Program to implement the Cohen-Sutherland line clipping algorithm. Make provision to specify the input line, window for clipping and viewport for displaying the clipped image.**

#include <stdio.h>

#include <GL/glut.h>

#define outcode int

double xmin=50,ymin=50, xmax=100,ymax=100;

double xvmin=200,yvmin=200,xvmax=300,yvmax=300;

const int RIGHT = 8;

const int LEFT = 2;

const int TOP = 4;

const int BOTTOM = 1;

outcode ComputeOutCode (double x, double y);

void CohenSutherlandLineClipAndDraw (double x0, double y0,double x1, double y1)

{

outcode outcode0, outcode1, outcodeOut;

bool accept = false, done = false;

outcode0 = ComputeOutCode (x0, y0);

outcode1 = ComputeOutCode (x1, y1);

do{

if (!(outcode0 | outcode1))

{

accept = true;

done = true;

}

else if (outcode0 & outcode1)

done = true;

else

{

double x, y;

outcodeOut = outcode0? outcode0: outcode1;

if (outcodeOut & TOP)

{

x = x0 + (x1 - x0) \* (ymax - y0)/(y1 - y0);

y = ymax;

}

else if (outcodeOut & BOTTOM)

{

x = x0 + (x1 - x0) \* (ymin - y0)/(y1 - y0);

y = ymin;

}

else if (outcodeOut & RIGHT)

{

y = y0 + (y1 - y0) \* (xmax - x0)/(x1 - x0);

x = xmax;

}

else

{

y = y0 + (y1 - y0) \* (xmin - x0)/(x1 - x0);

x = xmin;

}

if (outcodeOut == outcode0)

{

x0 = x;

y0 = y;

outcode0 = ComputeOutCode (x0, y0);

}

else

{

x1 = x;

y1 = y;

outcode1 = ComputeOutCode (x1, y1);

}

}

}while (!done);

if (accept)

{

double sx=(xvmax-xvmin)/(xmax-xmin);

double sy=(yvmax-yvmin)/(ymax-ymin);

double vx0=xvmin+(x0-xmin)\*sx;

double vy0=yvmin+(y0-ymin)\*sy;

double vx1=xvmin+(x1-xmin)\*sx;

double vy1=yvmin+(y1-ymin)\*sy;

glColor3f(1.0, 0.0, 0.0);

glBegin(GL\_LINE\_LOOP);

glVertex2f(xvmin, yvmin);

glVertex2f(xvmax, yvmin);

glVertex2f(xvmax, yvmax);

glVertex2f(xvmin, yvmax);

glEnd();

glColor3f(0.0,0.0,1.0);

glBegin(GL\_LINES);

glVertex2d (vx0, vy0);

glVertex2d (vx1, vy1);

glEnd();

}

}

outcode ComputeOutCode (double x, double y)

{

outcode code = 0;

if (y > ymax) //above the clip window

code |= TOP;

else if (y < ymin) //below the clip window

code |= BOTTOM;

if (x > xmax) //to the right of clip window

code |= RIGHT;

else if (x < xmin) //to the left of clip window

code |= LEFT;

return code;

}

void display()

{

double x0=120,y0=10,x1=40,y1=130;

glClear(GL\_COLOR\_BUFFER\_BIT);

glColor3f(1.0,0.0,0.0);

glBegin(GL\_LINES);

glVertex2d (x0, y0);

glVertex2d (x1, y1);

glVertex2d (60,20);

glVertex2d (80,120);

glEnd();

glColor3f(0.0, 0.0, 1.0);

glBegin(GL\_LINE\_LOOP);

glVertex2f(xmin, ymin);

glVertex2f(xmax, ymin);

glVertex2f(xmax, ymax);

glVertex2f(xmin, ymax);

glEnd();

CohenSutherlandLineClipAndDraw(x0,y0,x1,y1);

CohenSutherlandLineClipAndDraw(60,20,80,120);

glFlush();

}

void myinit()

{

glClearColor(1.0,1.0,1.0,1.0);

glColor3f(1.0,0.0,0.0);

glPointSize(1.0);

glMatrixMode(GL\_PROJECTION);

glLoadIdentity();

gluOrtho2D(0.0,499.0,0.0,499.0);

}

void main(int argc, char\*\* argv)

{

glutInit(&argc,argv);

glutInitDisplayMode(GLUT\_SINGLE|GLUT\_RGB);

glutInitWindowSize(500,500);

glutInitWindowPosition(0,0);

glutCreateWindow("Cohen Suderland Line Clipping Algorithm");

glutDisplayFunc(display);

myinit();

glutMainLoop();

}

1. **Program to create a house like figure and rotate it about a given fixed point using openGL functions.**

#include <stdio.h>

#include <math.h>

#include <GL/glut.h>

GLfloat house[3][9]={{100.0,100.0,175.0,250.0,250.0,150.0,150.0,200.0,200.0},   
 {100.0,300.0,400.0,300.0,100.0,100.0,150.0,150.0,100.0},

{1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0}};

GLfloat rot\_mat[3][3]={{0},{0},{0}};

GLfloat result[3][9]={{0}, {0}, {0}};

GLfloat h=100.0;

GLfloat k=100.0;

GLfloat theta;

void multiply()

{ int i,j,l;

for(i=0;i<3;i++)

for(j=0;j<9;j++)

{

result[i][j]=0;

for(l=0;l<3;l++)

result[i][j]=result[i][j]+rot\_mat[i][l]\*house[l][j];

}

}

void rotate()

{

GLfloat m,n;

m=-h\*(cos(theta)-1)+k\*(sin(theta));

n=-k\*(cos(theta)-1)-h\*(sin(theta));

rot\_mat[0][0]=cos(theta);

rot\_mat[0][1]=-sin(theta);

rot\_mat[0][2]=m;

rot\_mat[1][0]=sin(theta);

rot\_mat[1][1]=cos(theta);

rot\_mat[1][2]=n;

rot\_mat[2][0]=0;

rot\_mat[2][1]=0;

rot\_mat[2][2]=1;

multiply();

}

void drawhouse()

{

glColor3f(0.0, 0.0, 1.0);

glBegin(GL\_LINE\_LOOP);

glVertex2f(house[0][0],house[1][0]);

glVertex2f(house[0][1],house[1][1]);

glVertex2f(house[0][3],house[1][3]);

glVertex2f(house[0][4],house[1][4]);

glEnd();

glColor3f(1.0,0.0,0.0);

glBegin(GL\_LINE\_LOOP);

glVertex2f(house[0][5],house[1][5]);

glVertex2f(house[0][6],house[1][6]);

glVertex2f(house[0][7],house[1][7]);

glVertex2f(house[0][8],house[1][8]);

glEnd();

glColor3f(0.0, 0.0, 1.0);

glBegin(GL\_LINE\_LOOP);

glVertex2f(house[0][1],house[1][1]);

glVertex2f(house[0][2],house[1][2]);

glVertex2f(house[0][3],house[1][3]);

glEnd();

}

void drawrotatedhouse()

{

glColor3f(0.0, 0.0, 1.0);

glBegin(GL\_LINE\_LOOP);

glVertex2f(result[0][0],result[1][0]);

glVertex2f(result[0][1],result[1][1]);

glVertex2f(result[0][3],result[1][3]);

glVertex2f(result[0][4],result[1][4]);

glEnd();

glColor3f(1.0,0.0,0.0);

glBegin(GL\_LINE\_LOOP);

glVertex2f(result[0][5],result[1][5]);

glVertex2f(result[0][6],result[1][6]);

glVertex2f(result[0][7],result[1][7]);

glVertex2f(result[0][8],result[1][8]);

glEnd();

glColor3f(0.0, 0.0, 1.0);

glBegin(GL\_LINE\_LOOP);

glVertex2f(result[0][1],result[1][1]);

glVertex2f(result[0][2],result[1][2]);

glVertex2f(result[0][3],result[1][3]);

glEnd();

}

void display()

{

glClear(GL\_COLOR\_BUFFER\_BIT);

drawhouse();

rotate();

drawrotatedhouse();

glFlush();

}

void myinit()

{

glClearColor(1.0,1.0,1.0,1.0);

glColor3f(1.0,0.0,0.0);

glPointSize(1.0);

glMatrixMode(GL\_PROJECTION);

glLoadIdentity();

gluOrtho2D(0.0,499.0,0.0,499.0);

}

void main(int argc, char\*\* argv)

{

printf("Enter the rotation angle\n");

scanf("%f", &theta);

glutInit(&argc,argv);

glutInitDisplayMode(GLUT\_SINGLE|GLUT\_RGB);

glutInitWindowSize(500,500);

glutInitWindowPosition(0,0);

glutCreateWindow("house rotation");

glutDisplayFunc(display);

myinit();

glutMainLoop();

}

1. **Program to display a set of values {fij} as a rectangular mesh.**

#include <stdlib.h>

#include <GL/glut.h>

#define maxx 20

#define maxy 25

#define dx 15

#define dy 10

GLfloat x[maxx]={0.0},y[maxy]={0.0};

GLfloat x0=50,y0=50; // initial values for x, y

GLint i,j;

void init()

{

glClearColor(1.0,1.0,1.0,1.0);

glColor3f(1.0,0.0,0.0);

glPointSize(5.0);

glMatrixMode(GL\_PROJECTION);

glLoadIdentity();

gluOrtho2D(0.0,499.0,0.0,499.0);

glutPostRedisplay(); // request redisplay

}

void display(void)

{

/\* clear window \*/

glClear(GL\_COLOR\_BUFFER\_BIT);

glColor3f(0.0, 0.0, 1.0); // set color to blue

/\* draw rectangles \*/

for(i=0;i<maxx;i++)

x[i]=x0+i\*dx; // compute x[i]

for(j=0;j<maxy;j++)

y[j]=y0+j\*dy; // compute y[i]

glColor3f(0.0, 0.0, 1.0);

for(i=0;i<maxx-1;i++)

for(j=0;j<maxy-1;j++)

{

glColor3f(0.0, 0.0, 1.0);

glBegin(GL\_LINE\_LOOP);

glVertex2f(x[i],y[j]);

glVertex2f(x[i],y[j+1]);

glVertex2f(x[i+1],y[j+1]);

glVertex2f(x[i+1],y[j]);

glEnd();

glFlush();

}

glFlush();

}

void main(int argc, char\*\* argv)

{

glutInit(&argc, argv); // OpenGL initializations

glutInitDisplayMode(GLUT\_SINGLE | GLUT\_RGB);// single buffering and RGB

glutInitWindowSize(500, 400); // create a 500x400 window

glutInitWindowPosition(0, 0); // ...in the upper left

glutCreateWindow("Rectangular Mesh"); // create the window

glutDisplayFunc(display); // setup callbacks

init();

glutMainLoop(); // start it running

}

1. **Program to recursively subdivide a tetrahedron to from 3D Sierpinski gasket. The number of recursive steps is to be specified by the user.**

#include <stdlib.h>

#include <stdio.h>

#include <GL/glut.h>

typedef float point[3];

point v[]={{0.0, 0.0, 1.0}, {0.0, 0.942809, -0.33333},

{-0.816497, -0.471405, -0.333333}, {0.816497, -0.471405, -0.333333}};

static GLfloat theta[] = {0.0,0.0,0.0};

int n=2;

void triangle( point a, point b, point c)

{

glBegin(GL\_POLYGON);

glNormal3fv(a);

glVertex3fv(a);

glVertex3fv(b);

glVertex3fv(c);

glEnd();

}

void divide\_triangle(point a, point b, point c, int m)

{

point v1, v2, v3;

int j;

if(m>0)

{

for(j=0; j<3; j++) v1[j]=(a[j]+b[j])/2;

for(j=0; j<3; j++) v2[j]=(a[j]+c[j])/2;

for(j=0; j<3; j++) v3[j]=(b[j]+c[j])/2;

divide\_triangle(a, v1, v2, m-1);

divide\_triangle(c, v2, v3, m-1);

divide\_triangle(b, v3, v1, m-1);

}

else(triangle(a,b,c));

}

void tetrahedron( int m)

{

glColor3f(1.0,0.0,0.0);

divide\_triangle(v[0], v[1], v[2], m);

glColor3f(0.0,1.0,0.0);

divide\_triangle(v[3], v[2], v[1], m);

glColor3f(0.0,0.0,1.0);

divide\_triangle(v[0], v[3], v[1], m);

glColor3f(0.0,0.0,0.0);

divide\_triangle(v[0], v[2], v[3], m);

}

void display(void)

{

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

glLoadIdentity();

tetrahedron(n);

glFlush();

}

void myReshape(int w, int h)

{

glViewport(0, 0, w, h);

glMatrixMode(GL\_PROJECTION);

glLoadIdentity();

if (w <= h)

glOrtho(-2.0, 2.0, -2.0 \* (GLfloat) h / (GLfloat) w,

2.0 \* (GLfloat) h / (GLfloat) w, -10.0, 10.0);

else

glOrtho(-2.0 \* (GLfloat) w / (GLfloat) h,

2.0 \* (GLfloat) w / (GLfloat) h, -2.0, 2.0, -10.0, 10.0);

glMatrixMode(GL\_MODELVIEW);

glutPostRedisplay();

}

void main(int argc, char \*\*argv)

{

glutInit(&argc, argv);

glutInitDisplayMode(GLUT\_SINGLE | GLUT\_RGB | GLUT\_DEPTH);

glutInitWindowSize(500, 500);

glutCreateWindow("3D Gasket");

glutReshapeFunc(myReshape);

glutDisplayFunc(display);

glEnable(GL\_DEPTH\_TEST);

glClearColor (1.0, 1.0, 1.0, 1.0);

glutMainLoop();

}

1. **Program to draw a color cube and spin it using openGL transformation matrices.**

#include <stdlib.h>

#include <GL/glut.h>

GLfloat vertices[][3] = {{-1.0,-1.0,-1.0},{1.0,-1.0,-1.0},

{1.0,1.0,-1.0}, {-1.0,1.0,-1.0}, {-1.0,-1.0,1.0},

{1.0,-1.0,1.0}, {1.0,1.0,1.0}, {-1.0,1.0,1.0}};

GLfloat normals[][3] = {{-1.0,-1.0,-1.0},{1.0,-1.0,-1.0},

{1.0,1.0,-1.0}, {-1.0,1.0,-1.0}, {-1.0,-1.0,1.0},

{1.0,-1.0,1.0}, {1.0,1.0,1.0}, {-1.0,1.0,1.0}};

GLfloat colors[][3] = {{0.0,0.0,0.0},{1.0,0.0,0.0},

{1.0,1.0,0.0}, {0.0,1.0,0.0}, {0.0,0.0,1.0},

{1.0,0.0,1.0}, {1.0,1.0,1.0}, {0.0,1.0,1.0}};

void polygon(int a, int b, int c , int d)

{

glBegin(GL\_POLYGON);

glColor3fv(colors[a]);

glNormal3fv(normals[a]);

glVertex3fv(vertices[a]);

glColor3fv(colors[b]);

glNormal3fv(normals[b]);

glVertex3fv(vertices[b]);

glColor3fv(colors[c]);

glNormal3fv(normals[c]);

glVertex3fv(vertices[c]);

glColor3fv(colors[d]);

glNormal3fv(normals[d]);

glVertex3fv(vertices[d]);

glEnd();

}

void colorcube(void)

{

polygon(0,3,2,1);

polygon(2,3,7,6);

polygon(0,4,7,3);

polygon(1,2,6,5);

polygon(4,5,6,7);

polygon(0,1,5,4);

}

static GLfloat theta[] = {0.0,0.0,0.0};

static GLint axis = 2;

void display(void)

{

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

glLoadIdentity();

glRotatef(theta[0], 1.0, 0.0, 0.0);

glRotatef(theta[1], 0.0, 1.0, 0.0);

glRotatef(theta[2], 0.0, 0.0, 1.0);

colorcube();

glFlush();

glutSwapBuffers();

}

void spinCube()

{

theta[axis] += 1.0;

if( theta[axis] > 360.0 ) theta[axis] -= 360.0;

glutPostRedisplay();

}

void mouse(int btn, int state, int x, int y)

{

if(btn==GLUT\_LEFT\_BUTTON && state == GLUT\_DOWN) axis = 0;

if(btn==GLUT\_MIDDLE\_BUTTON && state == GLUT\_DOWN) axis = 1;

if(btn==GLUT\_RIGHT\_BUTTON && state == GLUT\_DOWN) axis = 2;

}

void myReshape(int w, int h)

{

glViewport(0, 0, w, h);

glMatrixMode(GL\_PROJECTION);

glLoadIdentity();

if (w <= h)

glOrtho(-2.0, 2.0, -2.0 \* (GLfloat) h / (GLfloat) w,

2.0 \* (GLfloat) h / (GLfloat) w, -10.0, 10.0);

else

glOrtho(-2.0 \* (GLfloat) w / (GLfloat) h,

2.0 \* (GLfloat) w / (GLfloat) h, -2.0, 2.0, -10.0, 10.0);

glMatrixMode(GL\_MODELVIEW);

}

void main(int argc, char \*\*argv)

{

glutInit(&argc, argv);

glutInitDisplayMode(GLUT\_DOUBLE | GLUT\_RGB | GLUT\_DEPTH);

glutInitWindowSize(500, 500);

glutCreateWindow("Rotating a Color Cube");

glutReshapeFunc(myReshape);

glutDisplayFunc(display);

glutIdleFunc(spinCube);

glutMouseFunc(mouse);

glEnable(GL\_DEPTH\_TEST); /\* Enable hidden--surface--removal \*/

glutMainLoop();

}

1. **Program, using openGL functions, to draw a simple shaded scene consisting of a teapot on a table. Define suitably the position and properties of the light source along with the properties of the surfaces of the solid object used in the scene.**

#include <GL/glut.h>

void wall (double thickness)

{

//draw thin wall with top = xz-plane, corner at origin

glPushMatrix();

glTranslated (0.5, 0.5 \* thickness, 0.5);

glScaled (1.0, thickness, 1.0);

glutSolidCube (1.0);

glPopMatrix();

}

//draw one table leg

void tableLeg (double thick, double len)

{

glPushMatrix();

glTranslated (0, len/2, 0);

glScaled (thick, len, thick);

glutSolidCube (1.0);

glPopMatrix();

}

void table (double topWid, double topThick, double legThick, double legLen)

{

//draw the table - a top and four legs

//draw the top first

glPushMatrix();

glTranslated (0, legLen, 0);

glScaled(topWid, topThick, topWid);

glutSolidCube (1.0);

glPopMatrix();

double dist = 0.95 \* topWid/2.0 - legThick/2.0;

glPushMatrix();

glTranslated (dist, 0, dist);

tableLeg (legThick, legLen);

glTranslated (0.0, 0.0, -2 \* dist);

tableLeg (legThick, legLen);

glTranslated (-2\*dist, 0, 2 \*dist);

tableLeg (legThick, legLen);

glTranslated(0, 0, -2\*dist);

tableLeg (legThick, legLen);

glPopMatrix();

}

void displaySolid (void)

{

//set properties of the surface material

GLfloat mat\_ambient[] = {0.7f, 0.7f, 0.7f, 1.0f}; // gray

GLfloat mat\_diffuse[] = {.5f, .5f, .5f, 1.0f};

GLfloat mat\_specular[] = {1.0f, 1.0f, 1.0f, 1.0f};

GLfloat mat\_shininess[] = {50.0f};

glMaterialfv (GL\_FRONT, GL\_AMBIENT, mat\_ambient);

glMaterialfv (GL\_FRONT, GL\_DIFFUSE, mat\_diffuse);

glMaterialfv (GL\_FRONT, GL\_SPECULAR, mat\_specular);

glMaterialfv (GL\_FRONT, GL\_SHININESS, mat\_shininess);

//set the light source properties

GLfloat lightIntensity[] = {0.7f, 0.7f, 0.7f, 1.0f};

GLfloat light\_position[] = {2.0f, 6.0f, 3.0f, 0.0f};

glLightfv (GL\_LIGHT0, GL\_POSITION, light\_position);

glLightfv (GL\_LIGHT0, GL\_DIFFUSE, lightIntensity);

//set the camera

glMatrixMode (GL\_PROJECTION);

glLoadIdentity();

double winHt = 1.0; //half-height of window

glOrtho (-winHt \* 64/48.0, winHt\*64/48.0, -winHt, winHt, 0.1, 100.0);

glMatrixMode (GL\_MODELVIEW);

glLoadIdentity();

gluLookAt (2.3, 1.3, 2.0, 0.0, 0.25, 0.0, 0.0, 1.0, 0.0);

//start drawing

glClear (GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

glPushMatrix();

glTranslated (0.4, 0.4, 0.6);

glRotated (45, 0, 0, 1);

glScaled (0.08, 0.08, 0.08);

glPopMatrix();

glPushMatrix();

glTranslated (0.6, 0.38, 0.5);

glRotated (30, 0, 1, 0);

glutSolidTeapot (0.08);

glPopMatrix ();

glPushMatrix();

glTranslated (0.25, 0.42, 0.35);

//glutSolidSphere (0.1, 15, 15);

glPopMatrix();

glPushMatrix();

glTranslated (0.4, 0, 0.4);

table (0.6, 0.02, 0.02, 0.3);

glPopMatrix();

wall (0.02);

glPushMatrix();

glRotated (90.0, 0.0, 0.0, 1.0);

wall (0.02);

glPopMatrix();

glPushMatrix();

glRotated (-90.0, 1.0, 0.0, 0.0);

wall (0.02);

glPopMatrix();

glFlush();

}

void main (int argc, char \*\* argv)

{

glutInit (&argc, argv);

glutInitDisplayMode (GLUT\_SINGLE|GLUT\_RGB|GLUT\_DEPTH);

glutInitWindowSize (640, 480);

glutInitWindowPosition (100, 100);

glutCreateWindow ("simple shaded scene consisting of a tea pot on a table");

glutDisplayFunc (displaySolid);

glEnable (GL\_LIGHTING);

glEnable (GL\_LIGHT0);

glShadeModel (GL\_SMOOTH);

glEnable (GL\_DEPTH\_TEST);

glEnable (GL\_NORMALIZE);

glClearColor (0.1, 0.1, 0.1, 0.0);

glViewport (0, 0, 640, 480);

glutMainLoop();

}