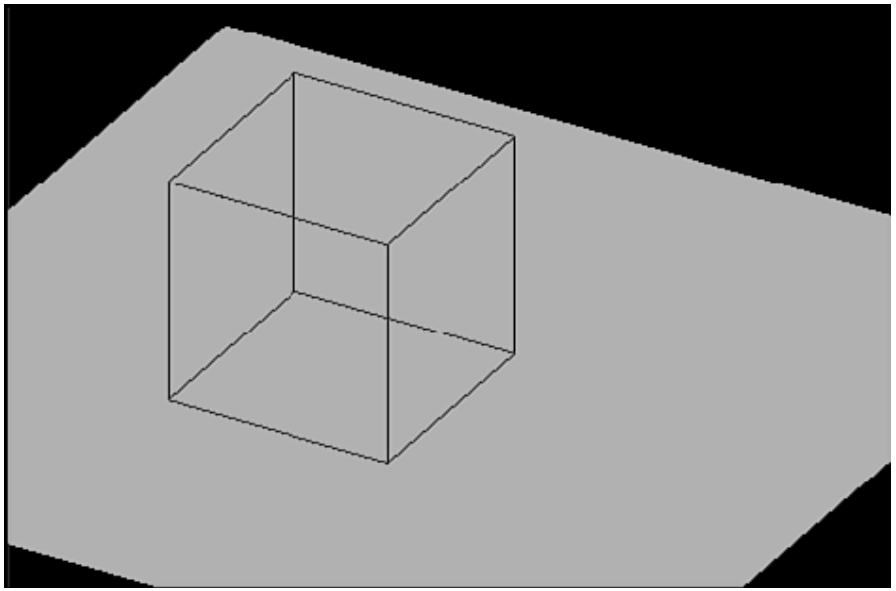
A Survey of 3D Effects

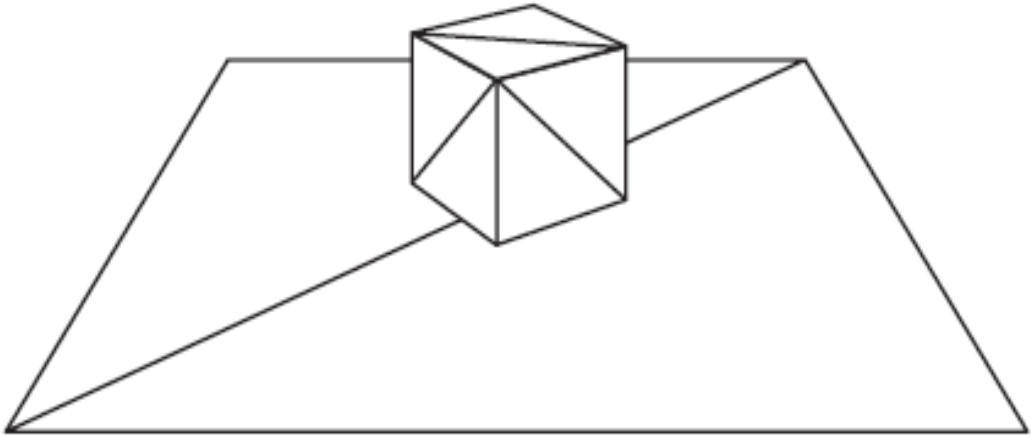
- The illusion of 3D is created on a flat computer screen by means of a bag full of perspective and artistic tricks.
  1. **Perspective**
  2. **Color**
  3. **Shading and Shadows**
  4. **Texture Mapping**
  5. **Fog**
  6. **Blending and Transparency**
  7. **Antialiasing**

1- Perspective

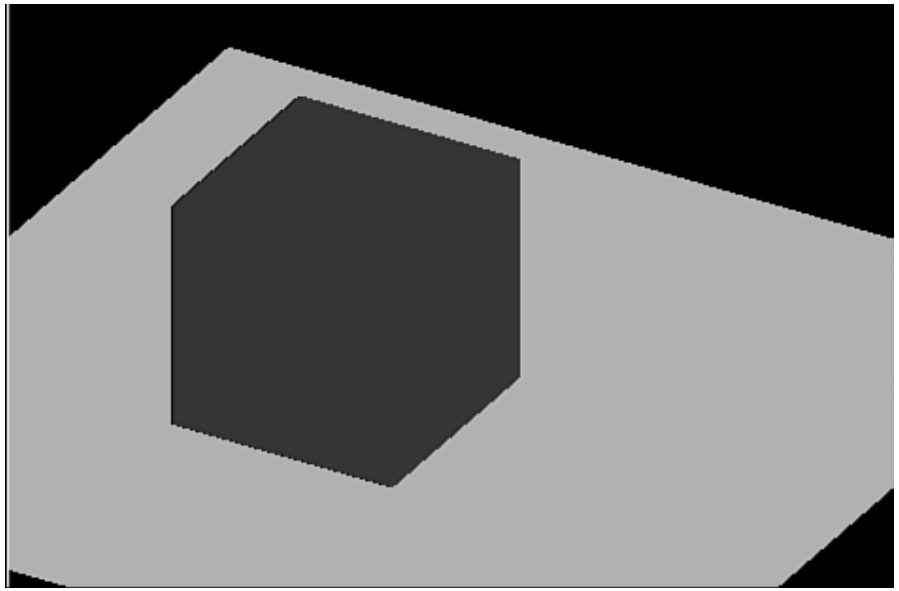


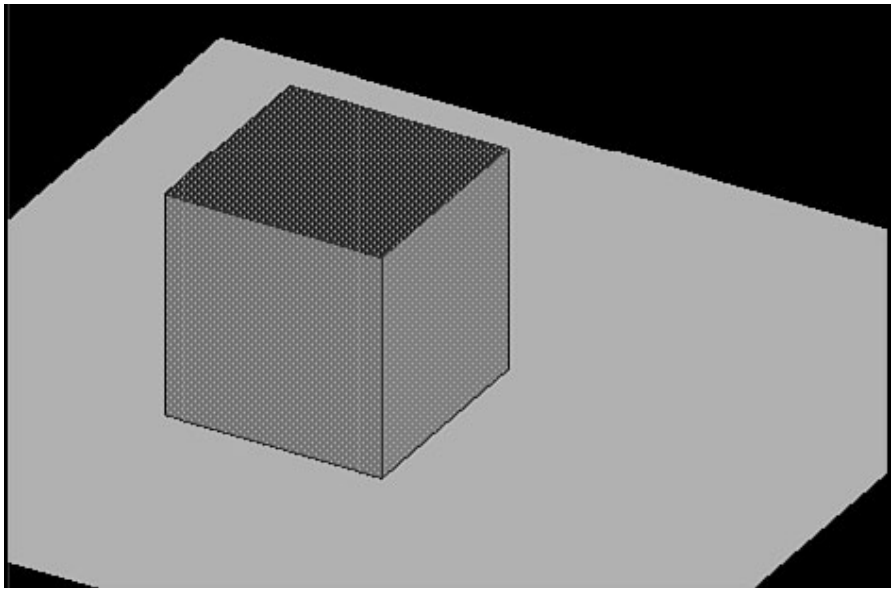
Perspective alone is enough to create the appearance of three dimensions.

Hiding the back sides of solid geometry enhances the 3D illusion



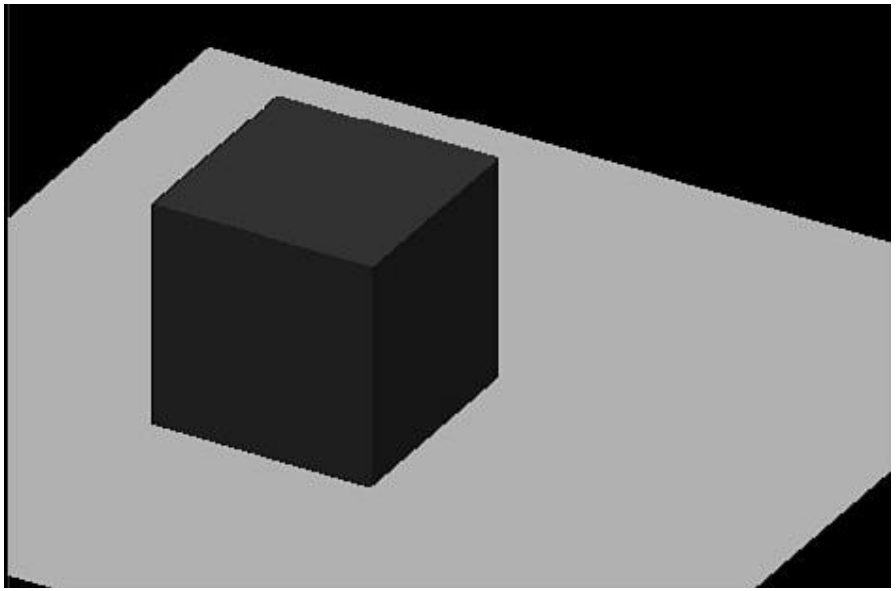
2- Color



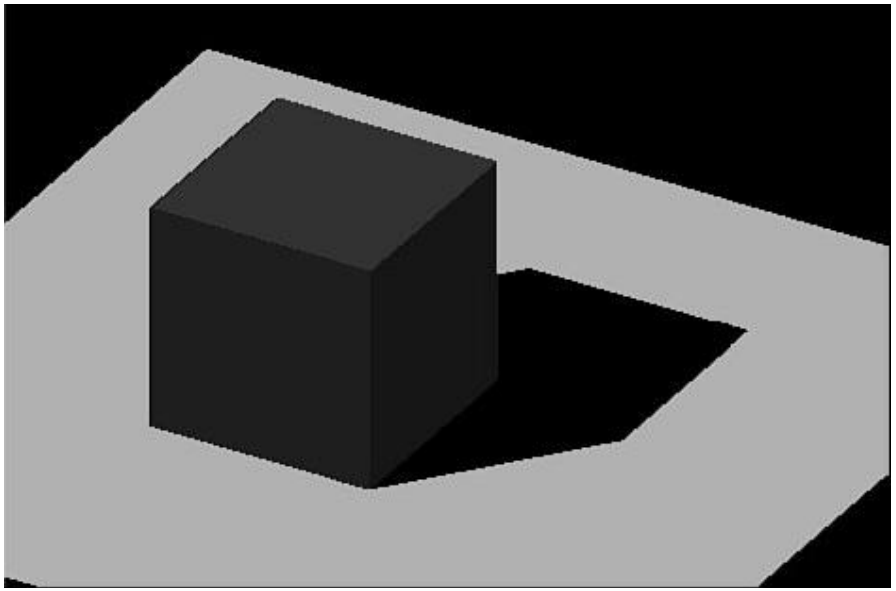


Adding different colors increases the illusion of three dimensions.

3- Shading and Shadows

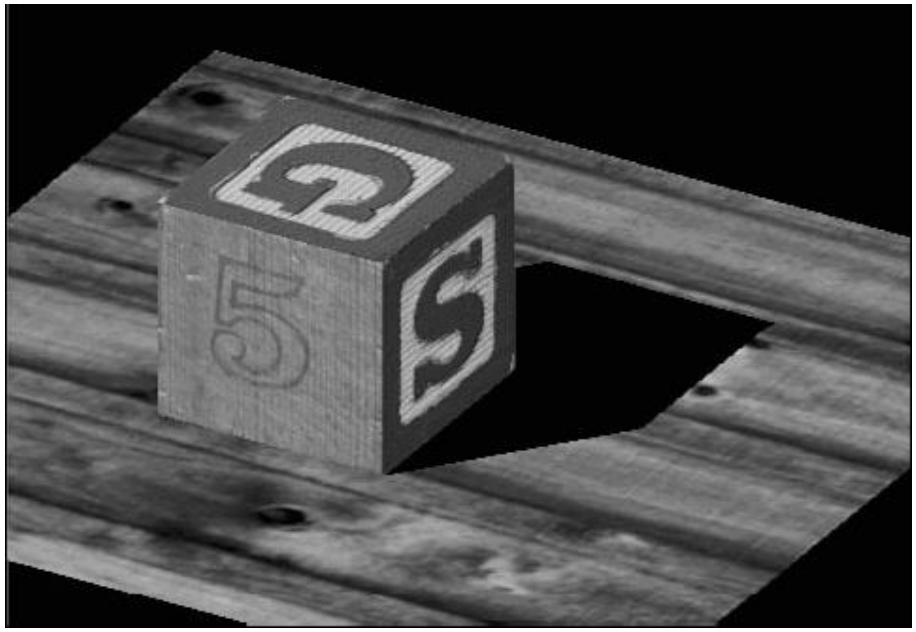


Proper shading creates the illusion of illumination.



Adding a shadow to further increase realism.

4-Texture Mapping

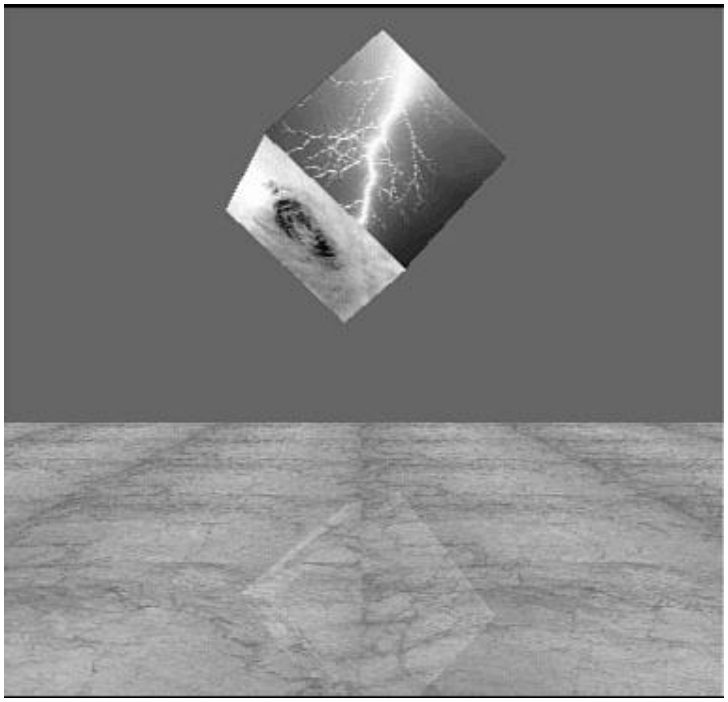


Texture mapping adds detail without adding additional geometry.

5- Fog

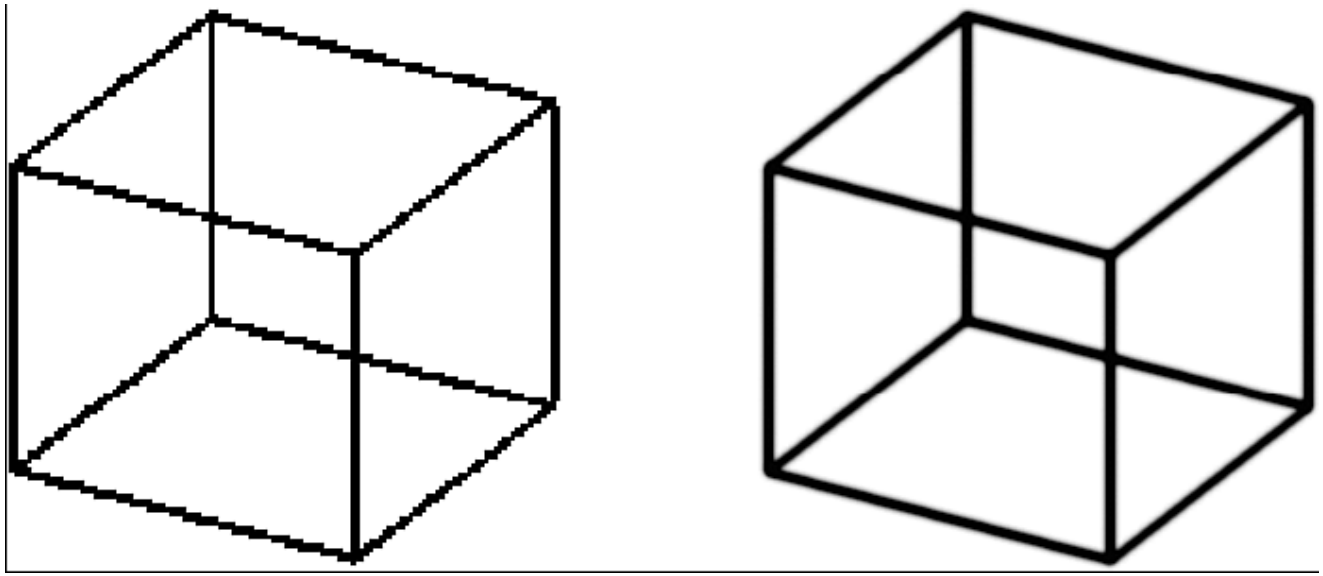
Fog effects provide a convincing illusion for wide-open spaces

6-Blending and Transparency



Blending used to achieve a reflection effect

7-Antialiasing



Cube with jagged lines versus cube with smooth lines

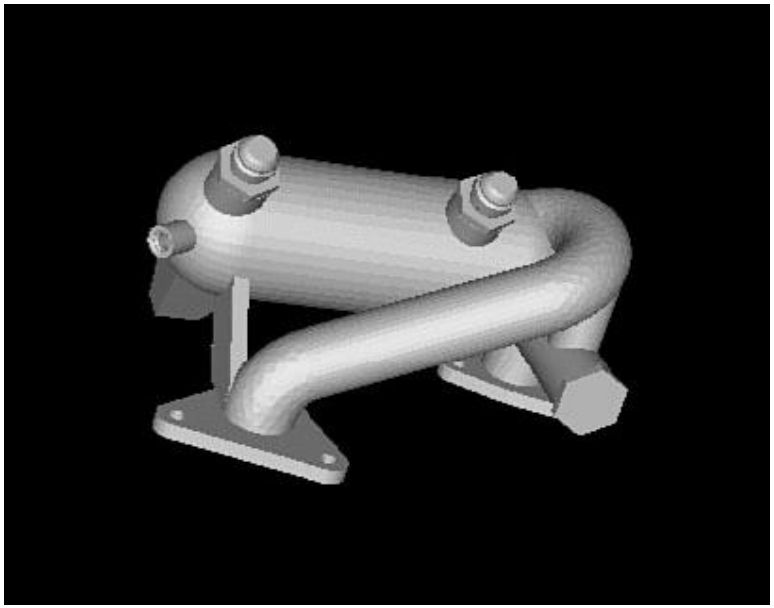
Common Uses for 3D Graphics

- **Real-Time 3D**
  - Flight simulator
  - Computer-aided design (CAD)
  - Architectural or civil planning
  - Medical imaging applications
  - Scientific visualization
  - Games

- **Non–Real-Time 3D**
  - Given more processing time, you can generate higher quality 3D graphics.
  - Rendering a single frame for a movie such as *Toy Story* or *Shrek* could take hours on a very fast computer.

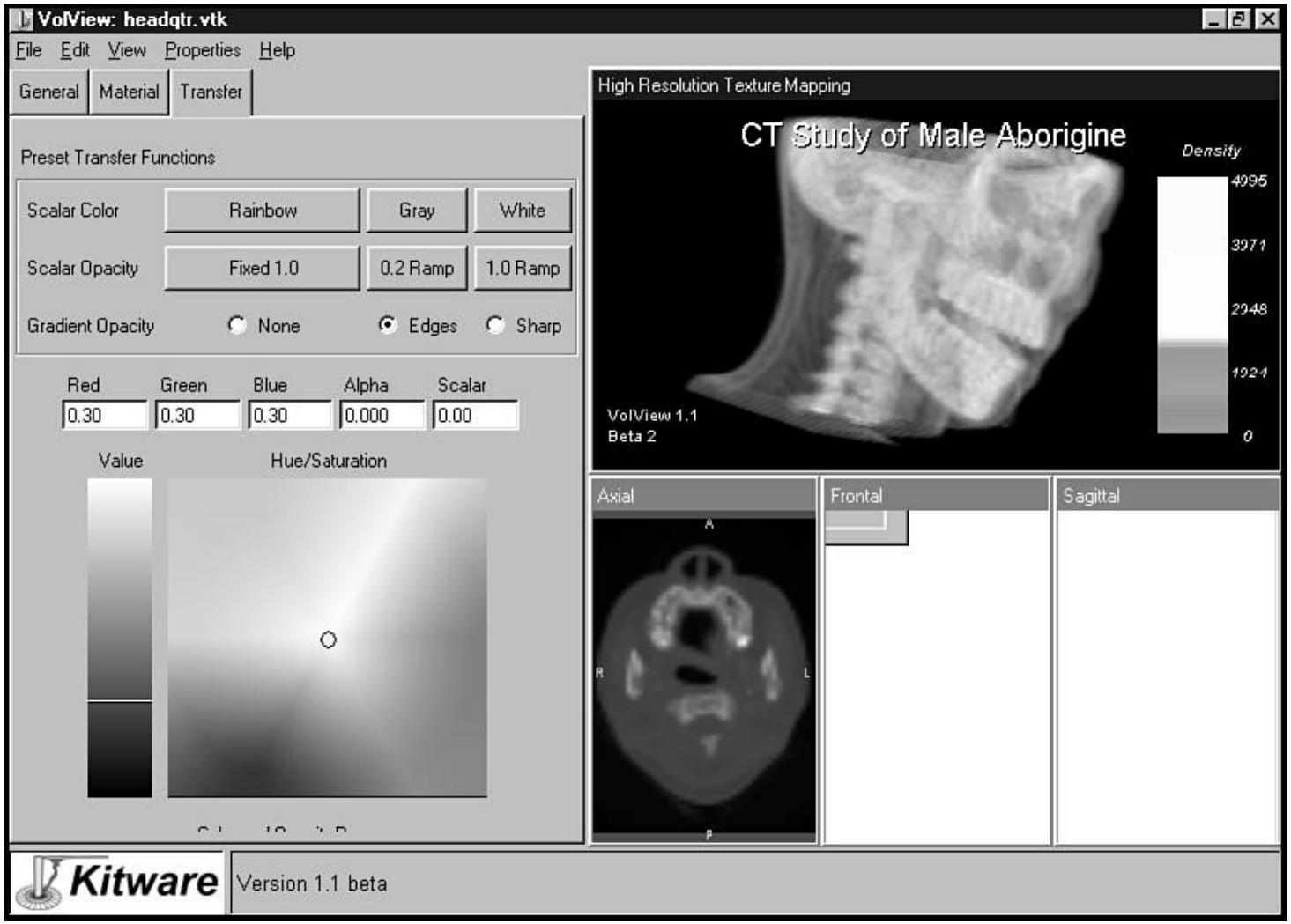
A popular OpenGL-based flight simulator from Flight Gear

3D graphics used for computer-aided design (CAD)

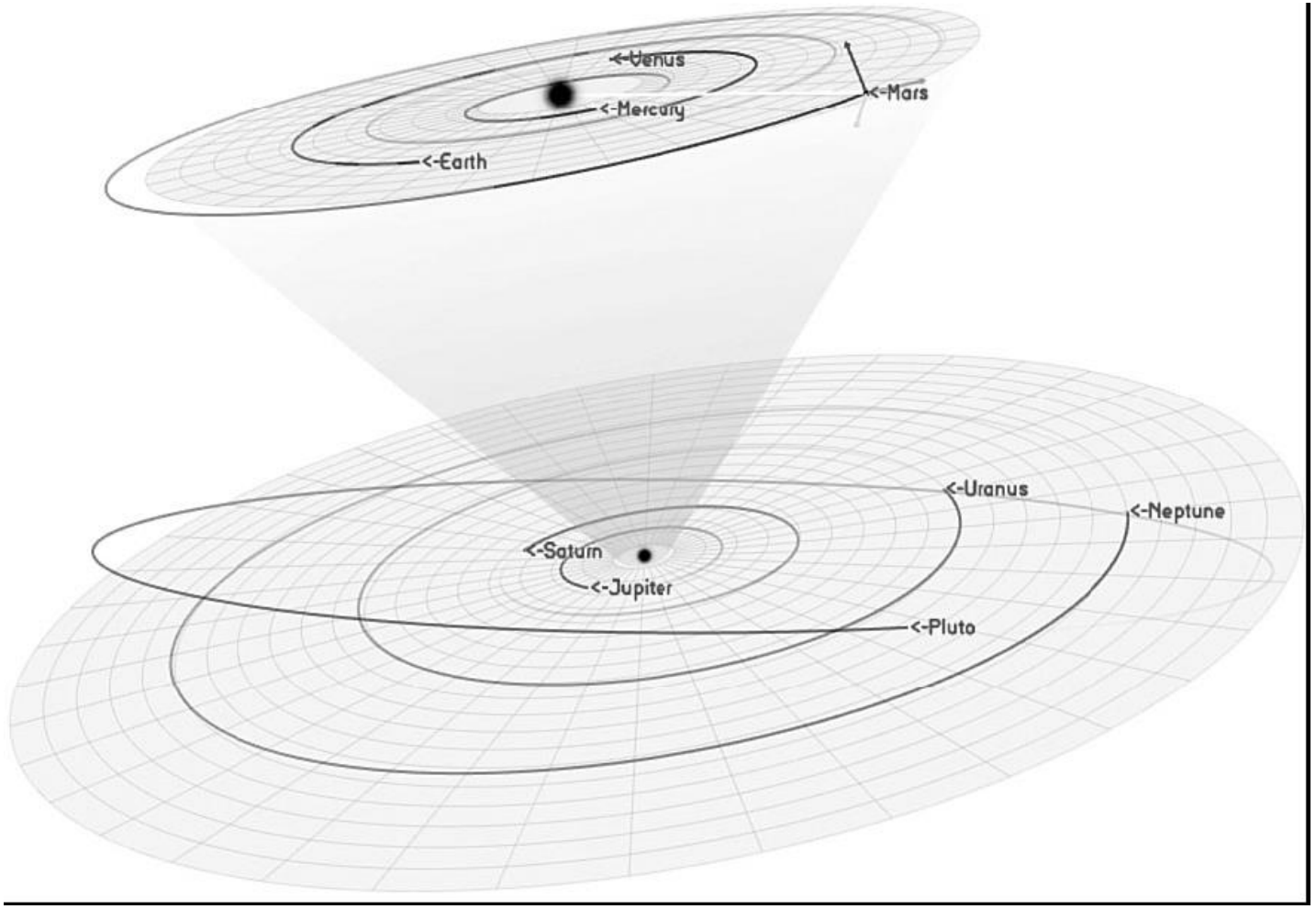


3D graphics used for architectural or civil planning

3D graphics used for medical imaging applications



3D graphics used for scientific visualization



3D graphics used for entertainment

Shaders

- Graphics cards are no longer dumb rendering chips, but highly programmable rendering computers

- Like the term CPU (central processing unit), the term GPU has been coined, meaning graphics processing unit, referring to the programmable chips on today's graphics cards.

Shaders allow for unprecedented real-time realism

OpenGL

- **What Is OpenGL?**
- OpenGL is a software interface to graphics hardware.
- OpenGL is designed as hardware-independent interface to be implemented on many different hardware platforms.
- OpenGL is not a programming language like C or C++. It is more like the C runtime library.
- 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, you must build up your desired model from a small set of geometric primitive - points, lines, and polygons.

Using GLUT

- OpenGL does not have a single function or command relating to window or screen management.
- OpenGL has no functions for keyboard input or mouse interaction.
  - You do not ask your graphics card if the user has pressed the enter key!
- In the beginning, there was AUX, the OpenGL auxiliary library. The AUX library was created to facilitate the learning and writing of OpenGL programs.
- You wouldn’t write “final” code when using AUX; it was more of a preliminary staging ground for testing your ideas.

Using GLUT (Cont.)

- AUX has since been replaced by the GLUT library for cross-platform programming examples and demonstrations. GLUT stands for *OpenGL utility toolkit.*

- GLUT is widely available on most UNIX distributions (including Linux), and is natively supported by Mac OS X.
  - On Windows, GLUT development has been discontinued!

- A new GLUT implementation, *freeglut.*

The API Wars

- When low-cost 3D graphics accelerators began to become available for the PC, many hardware vendors and game developers were attracted to OpenGL for its ease of use compared to Microsoft's Direct 3D.
- Microsoft provided a driver kit that made it very easy to make an OpenGL driver for Windows 98.

- Just before Windows 98 was released, Microsoft announced that it would not extend the OpenGL driver code license beyond the Windows 98 beta period, and that hardware vendors were forbidden to release their drivers.
- Virtually every PC hardware vendor had a robust and fast OpenGL driver ready to roll for consumer PCs, but couldn’t ship them.
- Hardware vendors with some help from Silicon Graphics, Inc. SGI continued to support OpenGL with new drivers.

- OpenGL’s popularity, however, has continued to grow as an alternative to Windows-specific rendering technology and is now widely supported across all major operating systems and hardware devices.

- Even cellphones with 3D graphics technology support a subset of OpenGL, called OpenGL ES.

- A hardware implementation is often referred to as an *accelerated implementation* because hardware-assisted 3D graphics usually far outperform software-only implementations.

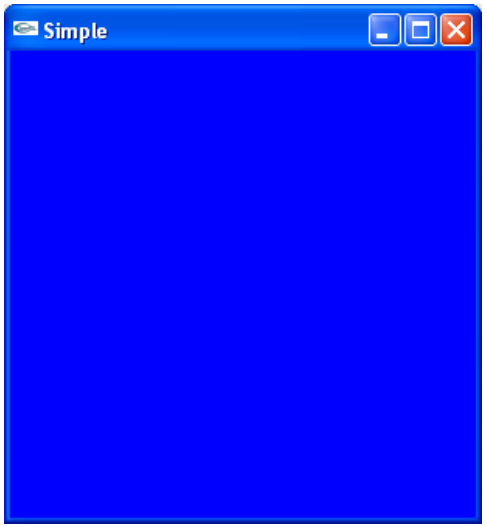
Download and Install GLUT

http://www.xmission.com/~nate/glut.html

- **Install Glut into the following directories:**
  - *glut32.dll* -> C:\Windows\System32
  - *glut32.lib* -> C:\Program Files\Microsoft Visual Studio .NET\Vc7\PlatformSDK\lib
  - *glut.h* -> C:\Program Files\Microsoft Visual Studio .NET\Vc7\PlatformSDK\Include\gl

- **Note:**
  In order for someone else on a different machine to run your application you must include the *glut32.dll* file with the program. If they do not have this file in the same directory as your application or in their *C:\Windows\System* folder and error will occur and the program will not run.

YourFirstProgram.c




```
#include <GL/glut.h>
void RenderScene(void) {
    glClear(GL_COLOR_BUFFER_BIT);
    glFlush(); // Flush drawing commands
}

int main(int argc, char* argv[]) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGBA);
    glutCreateWindow("Simple");
    glutDisplayFunc(RenderScene);
    glClearColor(0.0f, 0.0f, 1.0f, 1.0f);
    glutMainLoop();
    return 0;
}
```

Clearing the Window

- glClearColor(0.0, 0.0, 0.0, 0.0);
- glClear(GL_COLOR_BUFFER_BIT);
- The first line sets the clearing color to black, and the next command clears the entire window to the current clearing color.

- glClearColor(0.0, 0.0, 0.0, 0.0);
- glClearDepth(0.0);
- glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

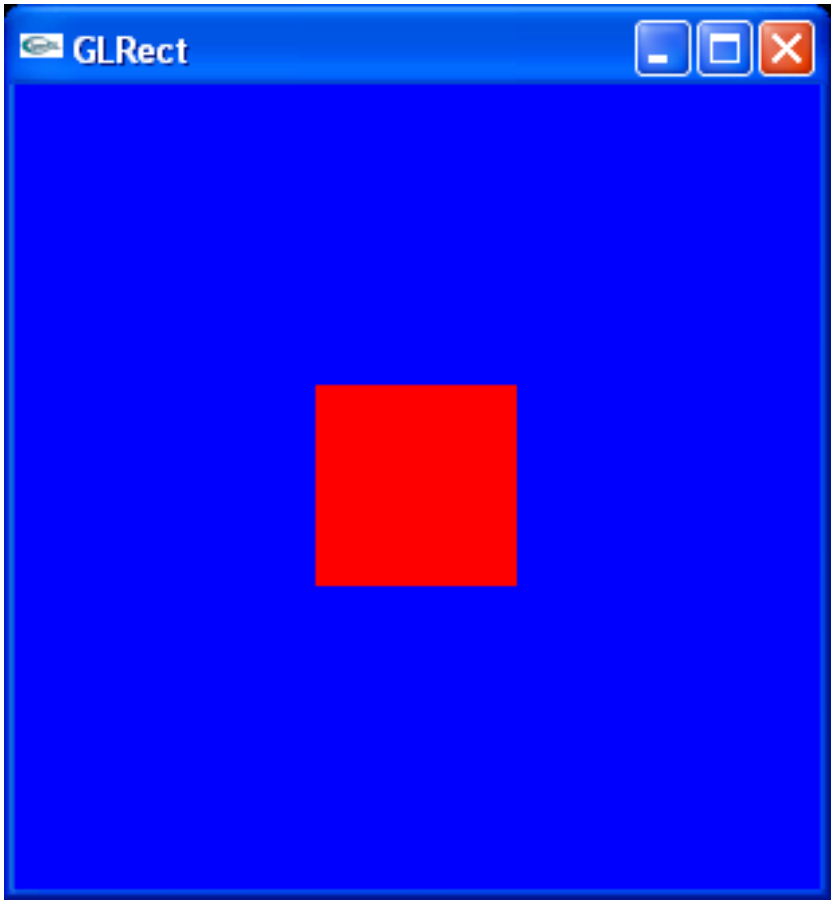
- glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

- and

- glClear(GL_COLOR_BUFFER_BIT);
- glClear(GL_DEPTH_BUFFER_BIT);

- both have the same final effect, the first example might run faster on many machines.

A Simple OpenGL Program



DrawingaCenteredRectanglewithOpenGL.c

```
#include <gl/glut.h>
void RenderScene(void){
    // Clear the window with current clearing color
    glClear(GL_COLOR_BUFFER_BIT);
    // Set current drawing color to red
    // R G B
    glColor3f(1.0f, 0.0f, 0.0f);
    // Draw a filled rectangle with current color
    // void glRectf(GLfloat x1, GLfloat y1, GLfloat x2, GLfloat y2);
    //The first pair represents the upper-left corner of the rectangle, and
    //the second pair represents the lower-right corner.

    glRectf(-25.0f, 25.0f, 25.0f, -25.0f);
    // Flush drawing commands
    glFlush();
}
// Set up the rendering state
void SetupRC(void) {
    // Set clear color to blue
    glClearColor(0.0f, 0.0f, 1.0f, 1.0f);
}
```

```
// Called by GLUT library when the window has chanaged size
void ChangeSize(GLsizei w, GLsizei h) {
    GLfloat aspectRatio;
    // Prevent a divide by zero
    if(h == 0) h = 1;
    // Set Viewport to window dimensions
    glViewport(0, 0, w, h);
    // Reset coordinate system
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    // Establish clipping volume (left, right, bottom, top, near, far)
    aspectRatio = (GLfloat)w / (GLfloat)h;
    if (w <= h)
        glOrtho (-100.0, 100.0, -100 / aspectRatio, 100.0 / aspectRatio,1.0, -
    1.0);
    else
        glOrtho (-100.0 * aspectRatio, 100.0 * aspectRatio,-100.0, 100.0,
    1.0, -1.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}
```

```
// Main program entry point
int main(int argc, char* argv[])
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutCreateWindow("GLRect");
    glutDisplayFunc(RenderScene);
    glutReshapeFunc(ChangeSize);
    SetupRC();
    glutMainLoop();
    return 0;
}
```

OpenGL Command Syntax

- OpenGL commands use the prefix **gl**

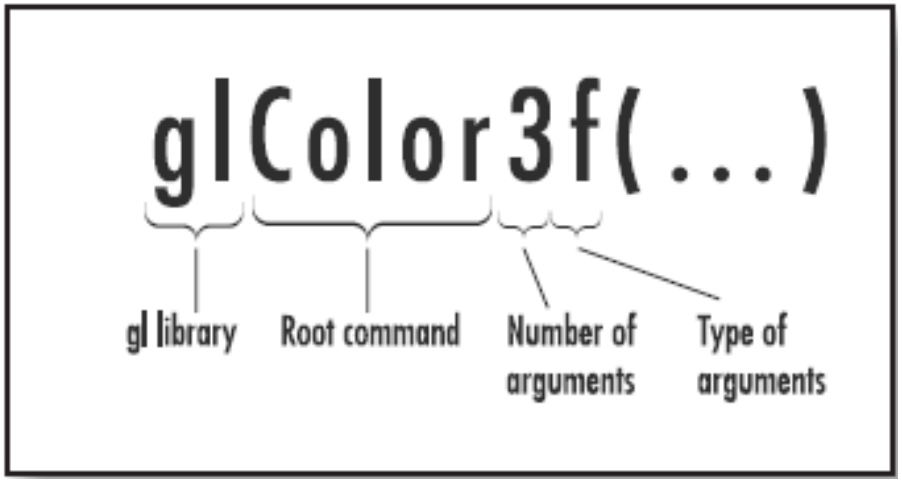
Table 1-1 : Command Suffixes and Argument Data Types

Suffix	**Data Type**		**OpenGL Type Definition**
b	8-bit integer	signed char	GLbyte
s	16-bit integer	short	GLshort
i	32-bit integer	long	GLint, GLsizei
f	32-bit floating-point	float	GLfloat, GLclampf
d	64-bit floating-point	double	GLdouble, GLclampd
ub	8-bit unsigned integer	unsigned char	GLubyte, GLboolean
us	16-bit unsigned integer	unsigned short	GLushort
ui	32-bit unsigned integer	unsigned long	GLuint, GLenum, GLbitfield

- Thus, the two commands: glVertex2i(1, 3); glVertex2f(1.0, 3.0);are equivalent

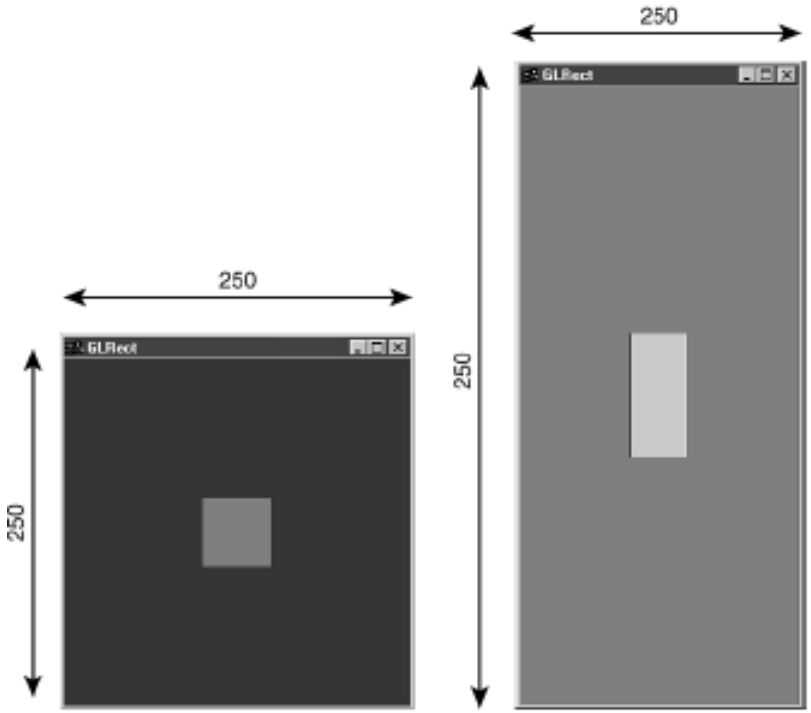
- The following lines show how you might use a vector and a nonvector version of the command that sets the current color:

- glColor3f(1.0, 0.0, 0.0);
- float color_array[] = {1.0, 0.0, 0.0};
- glColor3fv(color_array);



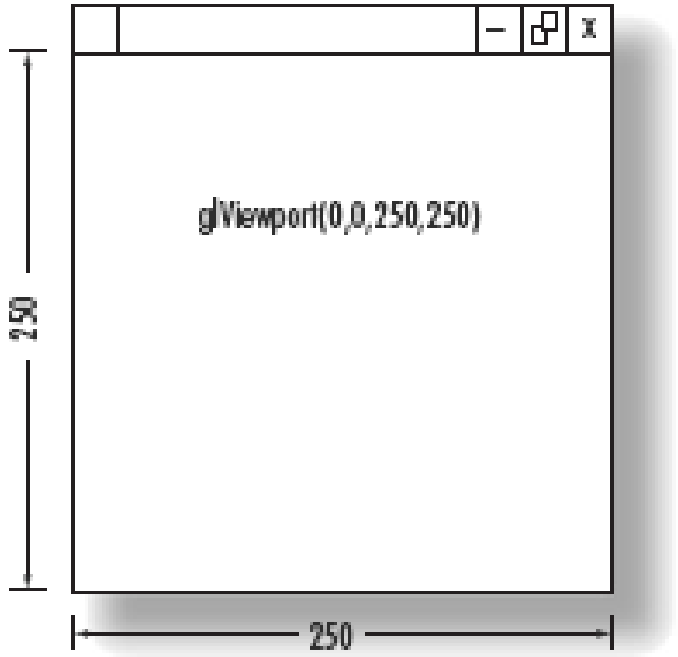
An OpenGL function.

- Whenever the window size changes, the viewport and clipping volume must be redefined for the new window dimensions.
- Otherwise, you see an effect like the one shown in the following Figure,
  - where the mapping of the coordinate system to screen coordinates stays the same regardless of the window size.

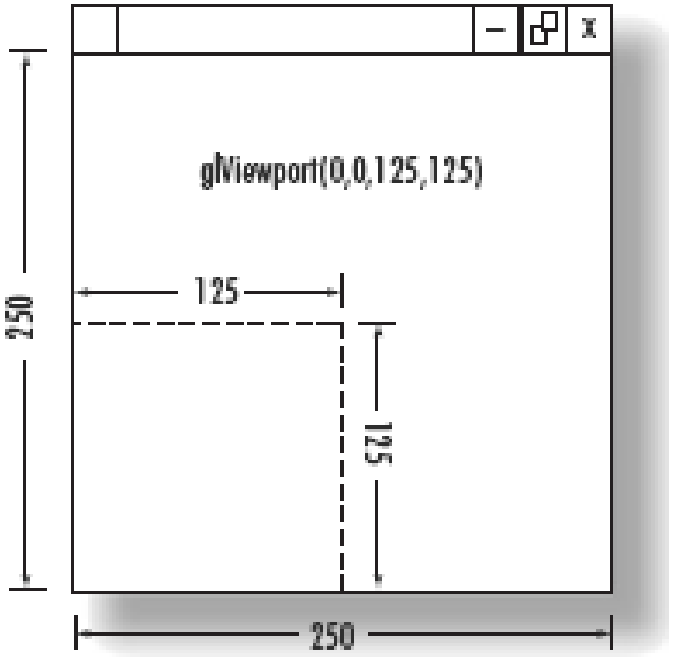


Viewport-to-window mapping.

void glViewport(GLint *x*, GLint *y*, GLsizei *width*, GLsizei *height*);



Window and viewport are same



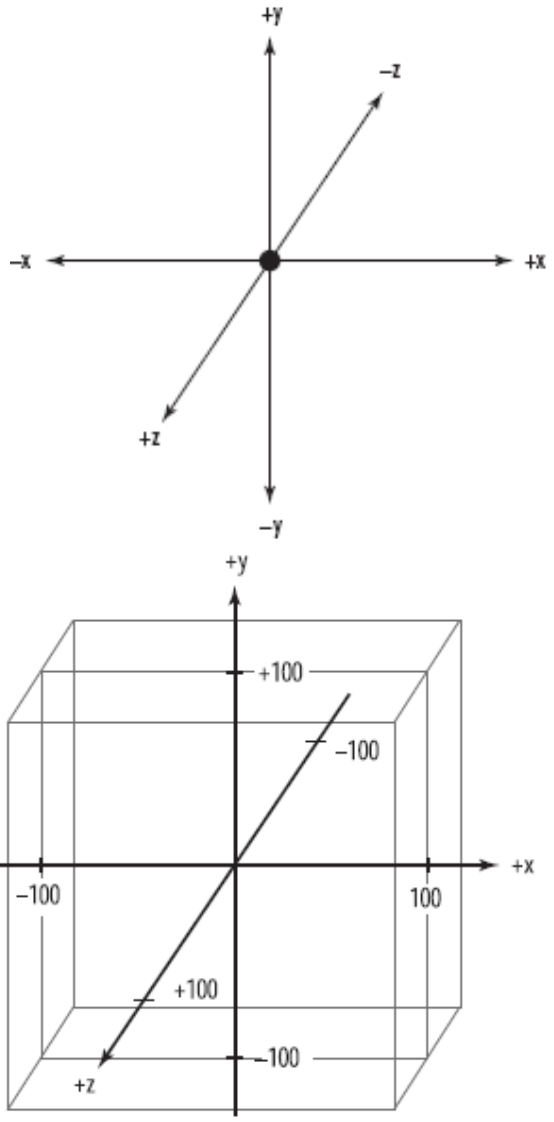
Viewport 1/2 size of window

Defining the Clipped Viewing Volume

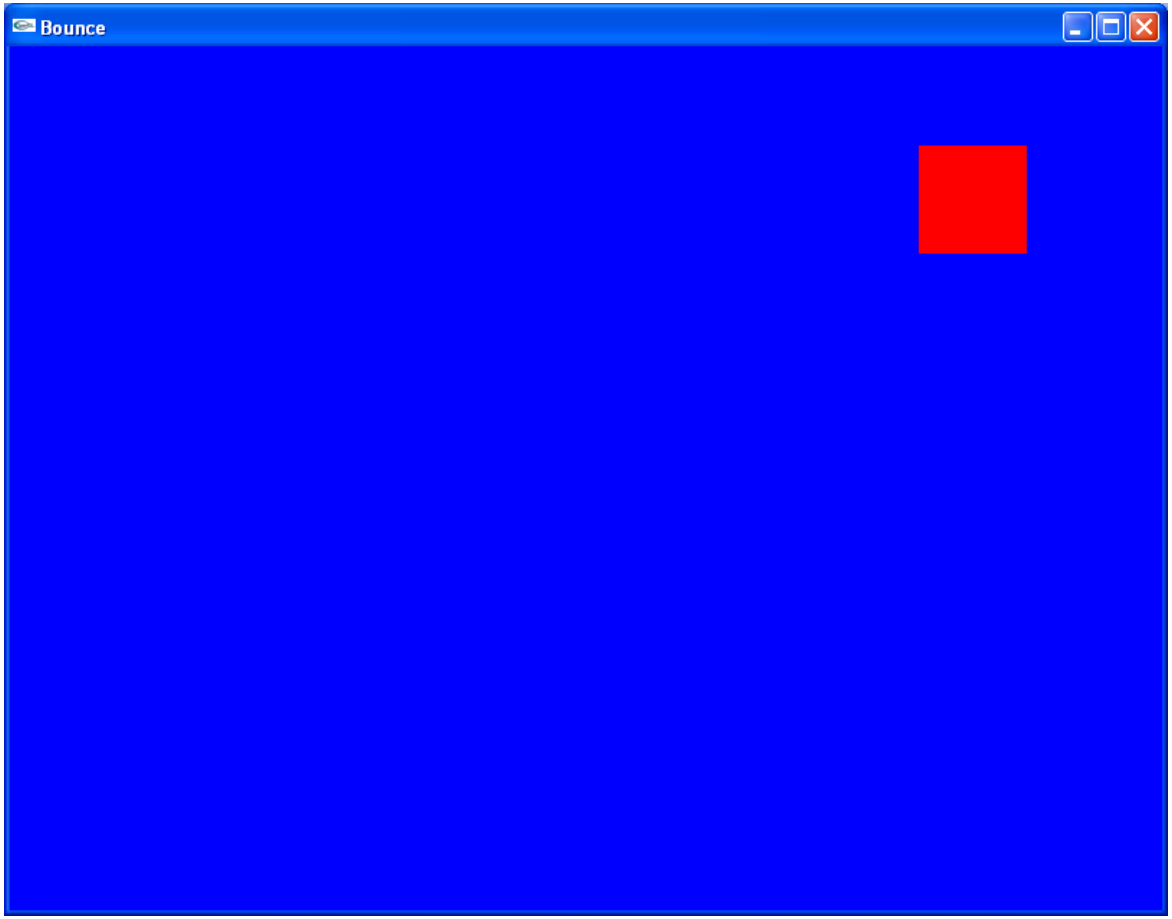
- The last requirement of our ChangeSize function is to redefine the clipping volume so that the aspect ratio remains square.

- If you specify a viewport that is not square and it is mapped to a square clipping volume, the image will be distorted.

- void glOrtho(GLdouble *left*, GLdouble *right*, GLdouble *bottom*, GLdouble *top*, GLdouble *near*, GLdouble *far*);
  - The *left* and *right* values specify the minimum and maximum coordinate value displayed along the x-axis.
  - *Bottom* and *top* are for the y-axis.
  - The *near* and *far* parameters are for the z-axis, generally with negative values extending away from the viewer



Animated Bouncing Square



Bounce.cpp (ch2)

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

// Initial square position and size
GLfloat x = 0.0f;
GLfloat y = 0.0f;
GLfloat rsize = 25;

// Step size in x and y directions
// (number of pixels to move each time)
GLfloat xstep = 1.0f;
GLfloat ystep = 1.0f;

// Keep track of windows changing width and height
GLfloat windowWidth;
GLfloat windowHeight;
```

```
void RenderScene(void)
    {
    // Clear the window with current clearing color
    glClear(GL_COLOR_BUFFER_BIT);

    // Set current drawing color to red
    //                    R        G            B
    glColor3f(1.0f, 0.0f, 0.0f);

    // Draw a filled rectangle with current color
    glRectf(x, y, x + rsize, y - rsize);

   // Flush drawing commands and swap
   glutSwapBuffers();
    }
```

```
// Called by GLUT library when idle (window not being resized or moved)
void TimerFunction(int value)     {
    // Reverse direction when you reach left or right edge
    if(x > windowWidth-rsize || x < -windowWidth)        xstep = -xstep;

    // Reverse direction when you reach top or bottom edge
    if(y > windowHeight || y < -windowHeight + rsize)        ystep = -ystep;

      // Actually move the square
    x += xstep;     y += ystep;

    // Check bounds. This is in case the window is made smaller while the rectangle is
      //bouncing and the rectangle suddenly finds itself outside the new clipping volume

    if(x > (windowWidth-rsize + xstep))        x = windowWidth-rsize-1;
      else if(x < -(windowWidth + xstep))    x = -windowWidth -1;

    if(y > (windowHeight + ystep))        y = windowHeight-1;
      else if(y < -(windowHeight - rsize + ystep))       y = -windowHeight + rsize - 1;

     // Redraw the scene with new coordinates
    glutPostRedisplay();
    glutTimerFunc(33,TimerFunction, 1);
    }
```

```
void SetupRC(void)    {
    // Set clear color to blue
    glClearColor(0.0f, 0.0f, 1.0f, 1.0f);
    }
// Called by GLUT library when the window has chanaged size
void ChangeSize(int w, int h)    {
    GLfloat aspectRatio;

    // Prevent a divide by zero
    if(h == 0)        h = 1;
   glViewport(0, 0, w, h);

    // Reset coordinate system
    glMatrixMode(GL_PROJECTION);    glLoadIdentity();

    // Establish clipping volume (left, right, bottom, top, near, far)
    aspectRatio = (GLfloat)w / (GLfloat)h;
    if (w <= h)        {
        windowWidth = 100;        windowHeight = 100 / aspectRatio;
        glOrtho (-100.0, 100.0, -windowHeight, windowHeight, 1.0, -1.0);
        }
    else        {
        windowWidth = 100 * aspectRatio;        windowHeight = 100;
        glOrtho (-windowWidth, windowWidth, -100.0, 100.0, 1.0, -1.0);
        }

    glMatrixMode(GL_MODELVIEW);    glLoadIdentity();
    }
```

```
int main(int argc, char* argv[])
      {
      glutInit(&argc, argv);
      glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA);
      glutInitWindowSize(800,600);
     glutCreateWindow("Bounce");
      glutDisplayFunc(RenderScene);
     glutReshapeFunc(ChangeSize);
      glutTimerFunc(33, TimerFunction, 1);

      SetupRC();

      glutMainLoop();

     return 0;
     }
```

glutTimerFunc

- void glutTimerFunc(unsigned int *msecs*, void (*func)(int *value*), int *value*);

- This code sets up GLUT to wait *msecs* milliseconds before calling the function func. You can pass a user-defined value in the *value* parameter.
- The function called by the timer has the following prototype:
  - void TimerFunction(int *value*);

OpenGL as a State Machine

- OpenGL is a state machine. You put it into various states (or modes) that then remain in effect until you change them.
- As you've already seen, the current color is a state variable.
- Many state variables refer to modes that are enabled or disabled with the command **glEnable()** or **glDisable()**.
- glEnable(GL_LIGHTING);
- glDisable(GL_LIGHTING);

- Each state variable or mode has a default value, and at any point you can query the system for each variable's current value.

- Typically, you use one of the four following commands to do this:
  - void glGetBooleanv(GLenum *pname*, GLboolean **params*);
  - void glGetDoublev(GLenum *pname*, GLdouble **params*);
  - void glGetFloatv(GLenum *pname*, GLfloat **params*);
  - void glGetIntegerv(GLenum *pname*, GLint **params*);

- Some state variables have a more specific query command (such as **glGetLight*()**, **glGetError()**, or **glGetPolygonStipple()**).
- You can save and later restore the values of a collection of state variables on an attribute stack with the **glPushAttrib()** and **glPopAttrib()** commands.
  - A single OpenGL state value or a whole range of related state values can be pushed on the attribute stack with the following command:
    - void glPopAttrib(GLbitfield *mask*);
    - For example, you could save the lighting and texturing state with a single call like this:
      - glPushAttrib(GL_TEXTURE_BIT | GL_LIGHTING_BIT);

Specifying a Color

```
set_current_color(red);
draw_object(A);
draw_object(B);
set_current_color(green);
set_current_color(blue);
draw_object(C);
```

- draws objects A and B in red, and object C in blue. The command on the fourth line that sets the current color to green is wasted.

- To set a color, use the command **glColor3f()**.

- glColor3f(0.0, 0.0, 0.0); black
- glColor3f(1.0, 0.0, 0.0); red
- glColor3f(0.0, 1.0, 0.0); green
- glColor3f(1.0, 1.0, 0.0); yellow
- glColor3f(0.0, 0.0, 1.0); blue
- glColor3f(1.0, 0.0, 1.0); magenta
- glColor3f(0.0, 1.0, 1.0); cyan
- glColor3f(1.0, 1.0, 1.0); white

Hidden-Surface Removal

- while (1) {

get_viewing_point_from_mouse_position();

glClear(GL_COLOR_BUFFER_BIT);

draw_3d_object_A();

draw_3d_object_B(); }

- it might be that for some mouse positions, object A obscures object B, and for others, the opposite relationship might hold. If nothing special is done, the preceding code always draws object B second, and thus on top of object A, no matter what viewing position is selected.

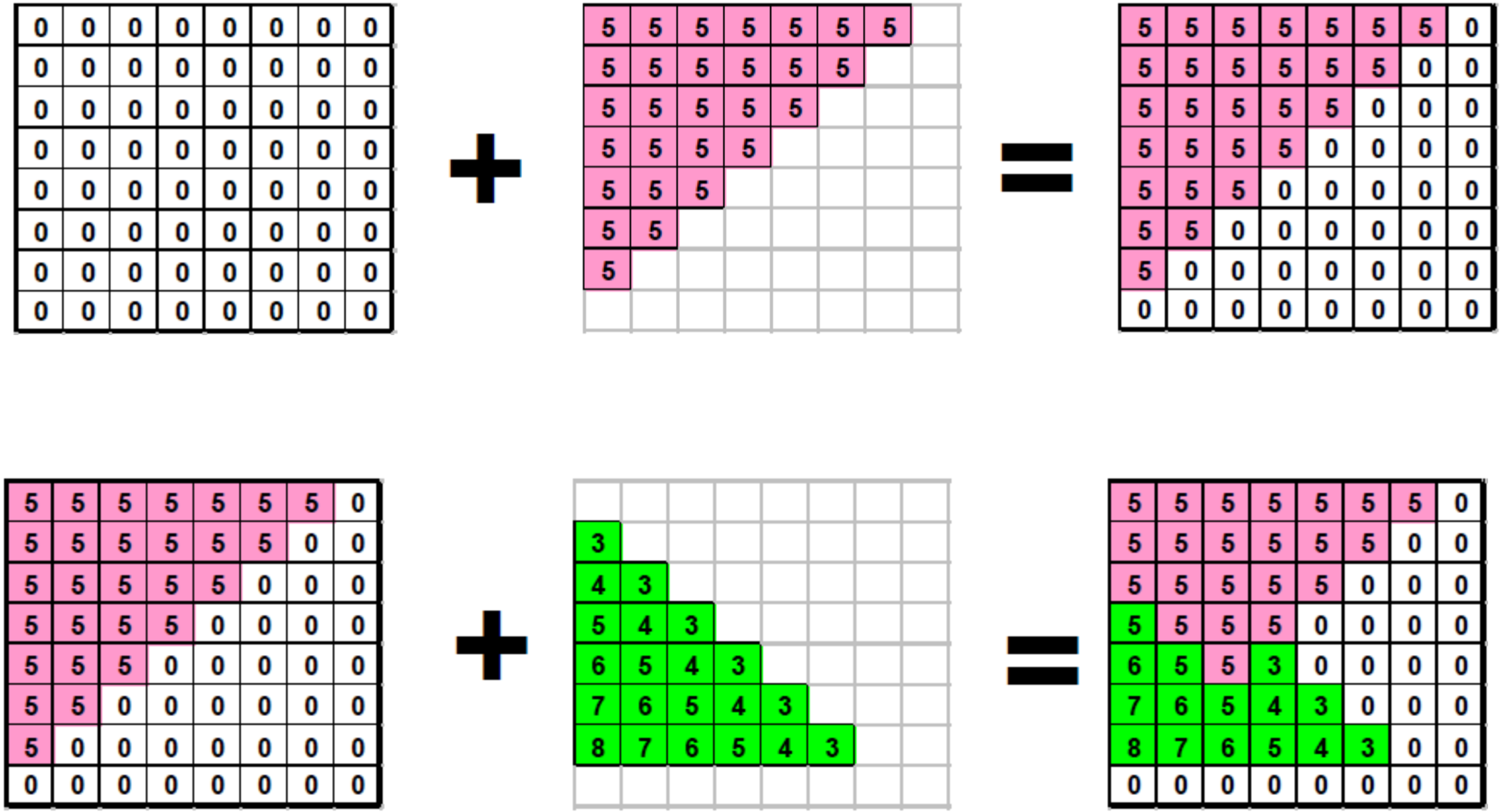
- A depth buffer works by associating a depth, or distance from the viewpoint, with each pixel on the window. Initially, the depth values for all pixels are set to the largest possible distance using the **glClear()** command with GL_DEPTH_BUFFER_BIT, and then the objects in the scene are drawn in any order.

- To convert the preceding program fragment so that it performs hidden-surface removal, modify it to the following:

```
glEnable(GL_DEPTH_TEST);
 ...
while (1) {
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
get_viewing_point_from_mouse_position();
draw_3d_object_A();
draw_3d_object_B(); }
```

- We want to see only those surfaces in front of other surfaces
- OpenGL uses a *hidden-surface* method called the *z*-buffer algorithm that saves depth information as objects are rendered so that only the front objects appear in the image

- Z-buffering (depth-buffering) is a visible surface detection algorithm
- Requires data structure (z-buffer) in addition to frame buffer.
- Z-buffer stores values [0 .. ZMAX] corresponding to depth of each point.
- If the point is closer than one in the buffers, it will replace the buffered values


0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
5 5 5 5 5 5 5
5 5 5 5 5 5
5 5 5 5 5
5 5 5 5
5 5 5
5 5
5
5 5 5 5 5 5 5 0
5 5 5 5 5 5 0 0
5 5 5 5 5 0 0 0
5 5 5 5 0 0 0 0
5 5 5 0 0 0 0 0
5 5 0 0 0 0 0 0
5 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
5 5 5 5 5 5 5 0
5 5 5 5 5 5 0 0
5 5 5 5 5 0 0 0
5 5 5 5 0 0 0 0
5 5 5 0 0 0 0 0
5 5 0 0 0 0 0 0
5 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
3
4 3
5 4 3
6 5 4 3
7 6 5 4 3
8 7 6 5 4 3
5 5 5 5 5 5 5 0
5 5 5 5 5 5 0 0
5 5 5 5 5 0 0 0
5 5 5 5 0 0 0 0
6 5 5 3 0 0 0 0
7 6 5 4 3 0 0 0
8 7 6 5 4 3 0 0
0 0 0 0 0 0 0 0

Z-buffering with front clipping

```
for (y = 0; y < YMAX; y++)
        for (x = 0; x < XMAX; x++) {
            F[x][y] = BACKGROUND_VALUE;
            Z[x][y] = -1; /* Back value in NPC */
}
for (each polygon)
    for (each pixel in polygon's projection) {
        pz = polygon's z-value at pixel coordinates (x,y)
        if (pz < FRONT && pz > Z[x][y]) { /* New point is
        behind front plane & closer than previous point */
            Z[x][y] = pz;
            F[x][y] = polygon's color at pixel coordinates (x,y)
    } }
```

Using the *z*-buffer algorithm

1. Requested in **main.c**

   glutInitDisplayMode (GLUT_SINGLE|GLUT_RGB|GLUT_DEPTH)

2. glEnable(GL_DEPTH_TEST)
3. Cleared in the display callback

   glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT)

OpenGL Errors

- Internally, OpenGL maintains a set of error flags. Each flag represents a different type of error. Whenever one of these errors occurs, the corresponding flag is set. To see whether any of these flags is set, call glGetError:
  - Glenum glGetError(void);

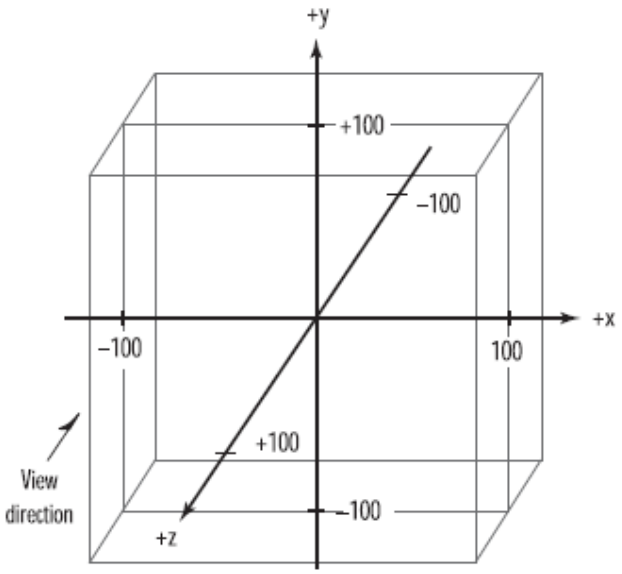
  - **Error Code Description**
    1. GL_INVALID_ENUM The enum argument is out of range.
    2. GL_INVALID_VALUE The numeric argument is out of range.
    3. GL_INVALID_OPERATION The operation is illegal in its current state.
    4. GL_STACK_OVERFLOW The command would cause a stack overflow.
    5. GL_STACK_UNDERFLOW The command would cause a stack underflow.
    6. GL_OUT_OF_MEMORY Not enough memory is left to execute the command.
    7. GL_TABLE_TOO_LARGE The specified table is too large.
    8. GL_NO_ERROR No error has occurred.

- If more than one of these flags is set, glGetError still returns only one distinct value. This value is then cleared when glGetError is called, and glGetError again will return either another error flag or GL_NO_ERROR.

- Usually, you want to call glGetError in a loop that continues checking for error flags until the return value is GL_NO_ERROR.

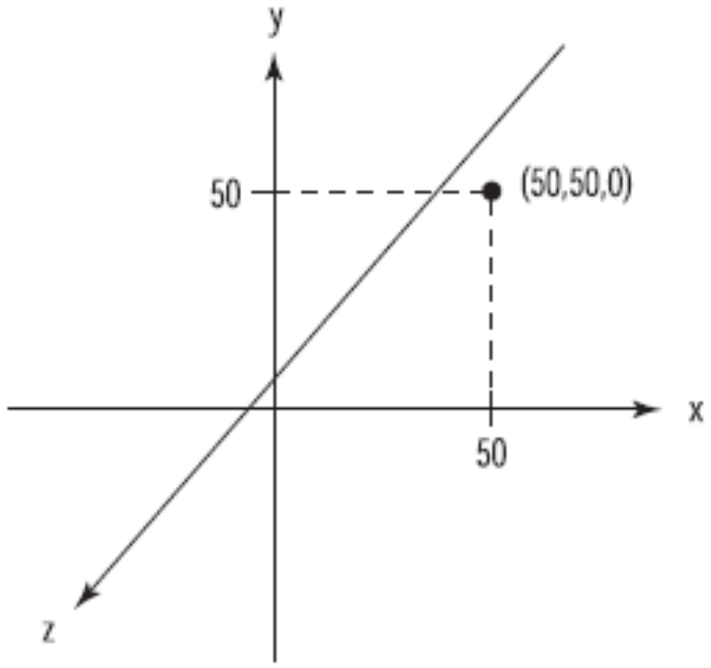
- You can use another function in the GLU library, gluErrorString, to get a string describing the error flag:
  - const GLubyte* gluErrorString(GLenum errorCode);

A simple viewing volume



- We established this volume with a call to glOrtho.

A 3D Point: The Vertex

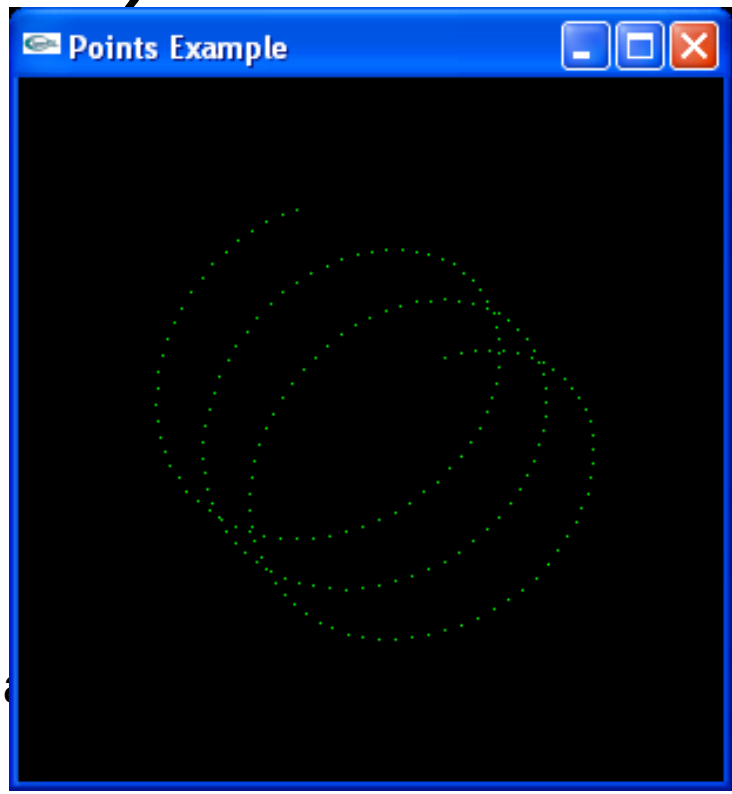


glVertex3f(50.0f, 50.0f, 0.0f);

OpenGL Geometric Drawing Primitives (Points)

```
glBegin(GL_POINTS); // Select points as the primitive
    glVertex3f(0.0f, 0.0f, 0.0f); // Specify a point
    glVertex3f(50.0f, 50.0f, 50.0f); // Specify another point
glEnd(); // Done drawing points
```

Points.cpp (ch3)




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

// Define a constant for the value of PI
#define GL_PI 3.1415f

// Rotation amounts
static GLfloat xRot = 0.0f;
static GLfloat yRot = 0.0f;

// Called to draw scene
void RenderScene(void)    {
    GLfloat x,y,z,angle; // Storeage for coordinates a

    // Clear the window with current clearing color
    glClear(GL_COLOR_BUFFER_BIT);

    // Save matrix state and do the rotation
    glPushMatrix();
    glRotatef(xRot, 1.0f, 0.0f, 0.0f);
    glRotatef(yRot, 0.0f, 1.0f, 0.0f);
```

```
// Call only once for all remaining points
glBegin(GL_POINTS);

z = -50.0f;
for(angle = 0.0f; angle <= (2.0f*GL_PI)*3.0f; angle += 0.1f)
        {

        //C runtime functions sin() and cos() accept angle values measured in radians
        x = 50.0f*sin(angle);
        y = 50.0f*cos(angle);

        // Specify the point and move the Z value up a little
        glVertex3f(x, y, z);
        z += 0.5f;
        }

// Done drawing points
glEnd();

// Restore transformations
glPopMatrix();

// Flush drawing commands
glutSwapBuffers();
}
```

```
// This function does any needed initialization on the rendering
// context.
void SetupRC()       {
      // Black background
      glClearColor(0.0f, 0.0f, 0.0f, 1.0f );

      // Set drawing color to green
      glColor3f(0.0f, 1.0f, 0.0f);
      }

void SpecialKeys(int key, int x, int y) {
      if(key == GLUT_KEY_UP) xRot-= 5.0f;
      if(key == GLUT_KEY_DOWN)  xRot += 5.0f;
      if(key == GLUT_KEY_LEFT) yRot -= 5.0f;
      if(key == GLUT_KEY_RIGHT) yRot += 5.0f;

      if(xRot > 356.0f) xRot = 0.0f;
      if(xRot < -1.0f) xRot = 355.0f;
      if(yRot > 356.0f) yRot = 0.0f;
      if(yRot < -1.0f) yRot = 355.0f;

      // Refresh the Window
      glutPostRedisplay();
      }
```

```
void ChangeSize(int w, int h)   {
      GLfloat nRange = 100.0f;

      // Prevent a divide by zero
      if(h == 0)          h = 1;

      // Set Viewport to window dimensions
    glViewport(0, 0, w, h);

      // Reset projection matrix stack
      glMatrixMode(GL_PROJECTION);
      glLoadIdentity();

      // Establish clipping volume (left, right, bottom, top, near, far)
    if (w <= h)
              glOrtho (-nRange, nRange, -nRange*h/w, nRange*h/w, -nRange, nRange);
    else
              glOrtho (-nRange*w/h, nRange*w/h, -nRange, nRange, -nRange, nRange);

      // Reset Model view matrix stack
      glMatrixMode(GL_MODELVIEW);
      glLoadIdentity();
      }
```

```
int main(int argc, char* argv[])
    {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutCreateWindow("Points Example");
    glutReshapeFunc(ChangeSize);
    glutSpecialFunc(SpecialKeys);
    glutDisplayFunc(RenderScene);
    SetupRC();
    glutMainLoop();

    return 0;
    }
```

Setting the Point Size

- To control the size of a rendered point, use **glPointSize()** and supply the desired size in pixels as the argument.

- void **glPointSize**(GLfloat *size*);
- Sets the width in pixels for rendered points; *size* must be greater than 0.0 and by default is 1.0.
- Thus, if the width is 1.0, the square is one pixel by one pixel; if the width is 2.0, the square is two pixels by two pixels, and so on.

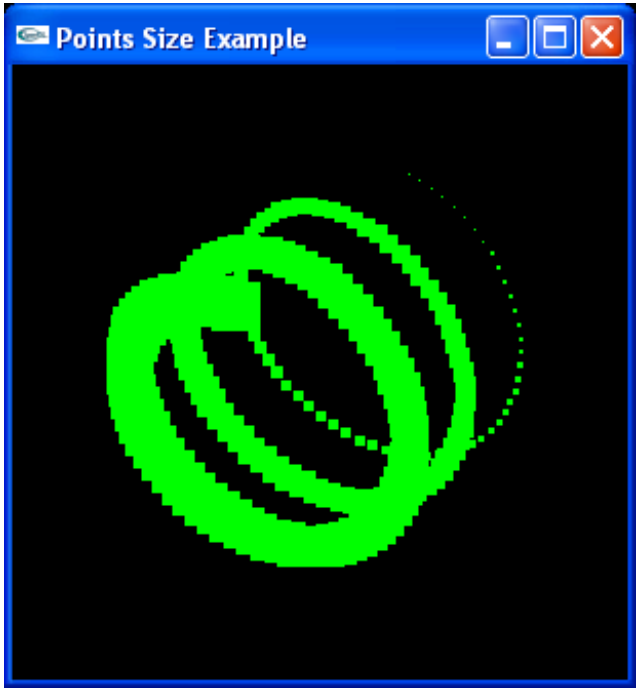
- Not all point sizes are supported, however, and you should make sure the point size you specify is available.

```
GLfloat sizes[2]; // Store supported point size range
GLfloat step; // Store supported point size increments
// Get supported point size range and step size
glGetFloatv(GL_POINT_SIZE_RANGE,sizes);
glGetFloatv(GL_POINT_SIZE_GRANULARITY,&step);
```

- The Microsoft software implementation of OpenGL, for example, allows for point sizes from 0.5 to 10.0, with 0.125 the smallest step size.

- By default, points, unlike other geometry, are not affected by the perspective division.
  - That is, they do not become smaller when they are further from the viewpoint, and they do not become larger as they move closer. Points are also always square pixels, even if you use glPointSize to increase the size of the points. You just get bigger squares! To get round points, you must draw them antialiased

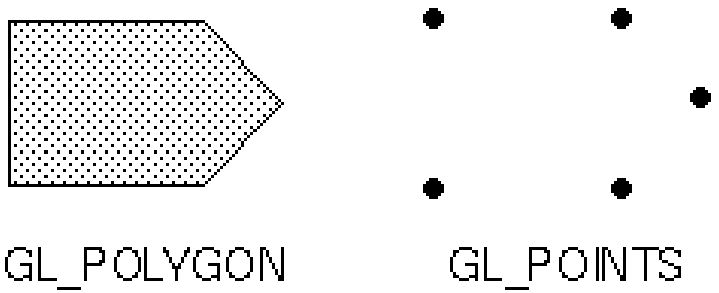
Pointsz.cpp



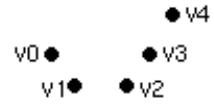

```
void RenderScene(void){ ….
    GLfloat sizes[2];  // Store supported point size range
    GLfloat step;      // Store supported point size increments
    GLfloat curSize;  // Store current size
….
    // Get supported point size range and step size
    glGetFloatv(GL_POINT_SIZE_RANGE,sizes);
    glGetFloatv(GL_POINT_SIZE_GRANULARITY,&step);

    curSize = sizes[0]; // Set the initial point size
     z = -50.0f;        // Set beginning z coordinate

    // Loop around in a circle three times
    for(angle = 0.0f; angle <= (2.0f*3.1415f)*3.0f; angle += 0.1f) {
            x = 50.0f*sin(angle);              y = 50.0f*cos(angle);

            // Specify the point size before the primative is specified
            glPointSize(curSize);

            glBegin(GL_POINTS);            glVertex3f(x, y, z); glEnd();

            // Bump up the z value and the point size
            z += 0.5f;
            curSize += step;
…..
```

OpenGL Geometric Drawing Primitives

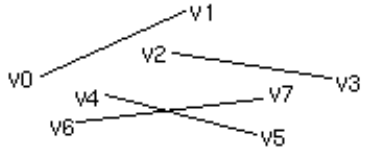
```
glBegin(GL_POLYGON);

    glVertex2f(0.0, 0.0);
    glVertex2f(0.0, 3.0);
    glVertex2f(3.0, 3.0);
    glVertex2f(4.0, 1.5);
    glVertex2f(3.0, 0.0);
glEnd();
```



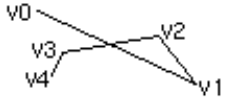
Geometric Primitive Types



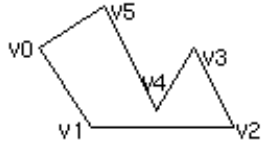
GL_POINTS



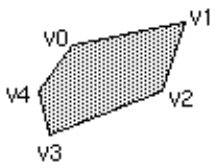
GL_LINES



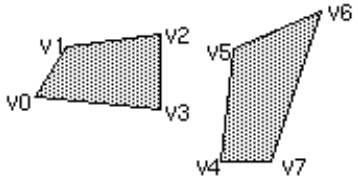
GL_LINE_STRIP



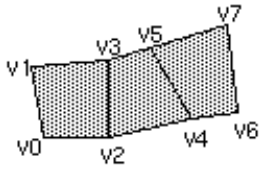
GL_LINE_LOOP



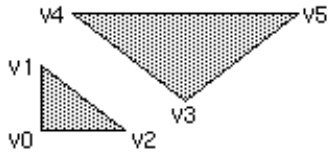
GL_POLYGON



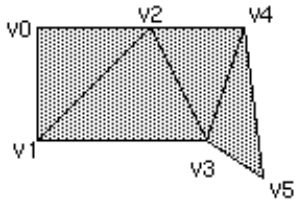
GL_QUADS



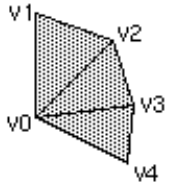
GL_QUAD_STRIP



GL_TRIANGLES



GL_TRIANGLE_STRIP



GL_TRIANGLE_FAN

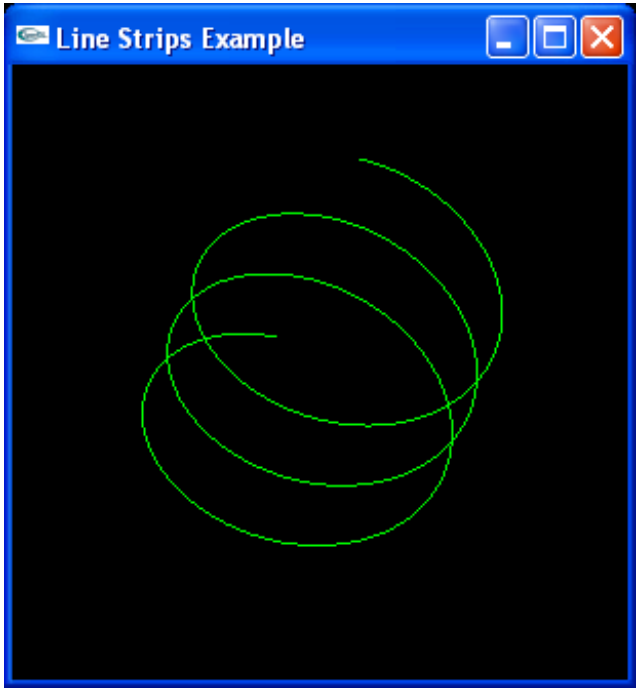
Restrictions on Using glBegin() and glEnd()

- You can also supply additional vertex-specific data for each vertex - a color, a normal vector, texture coordinates, or any combination of these - using special commands.
  - glColor*( ) set current color
  - glIndex*( ) set current color index
  - glNormal*( ) set normal vector coordinates
  - glEvalCoord*( ) generate coordinates
  - glCallList(), glCallLists( ) execute display list(s)
  - glTexCoord*( ) set texture coordinates
  - glEdgeFlag*( ) control drawing of edges
  - glMaterial*( ) set material properties

- No other OpenGL commands are valid between a **glBegin()** and **glEnd()** pair. Note, however, that only OpenGL commands are restricted;

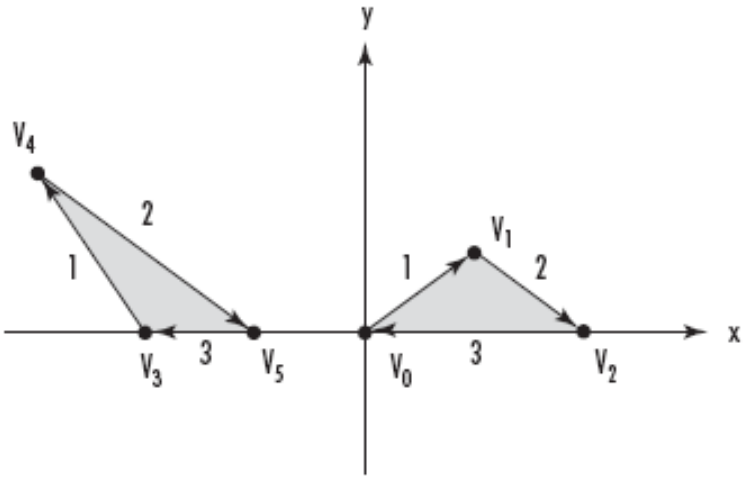
LStrips.cpp

```
// Call only once for all remaining points
glBegin(GL_LINE_STRIP);

z = -50.0f;
for(angle = 0.0f; angle <= (2.0f*GL_PI)*3.0f; angle += 0.1f){
      x = 50.0f*sin(angle);
      y = 50.0f*cos(angle);
      // Specify the point and
      //move the z value up a little
      glVertex3f(x, y, z);
      z += 0.5f;
}

// Done drawing points
glEnd();
```



Triangles: Your First Polygon

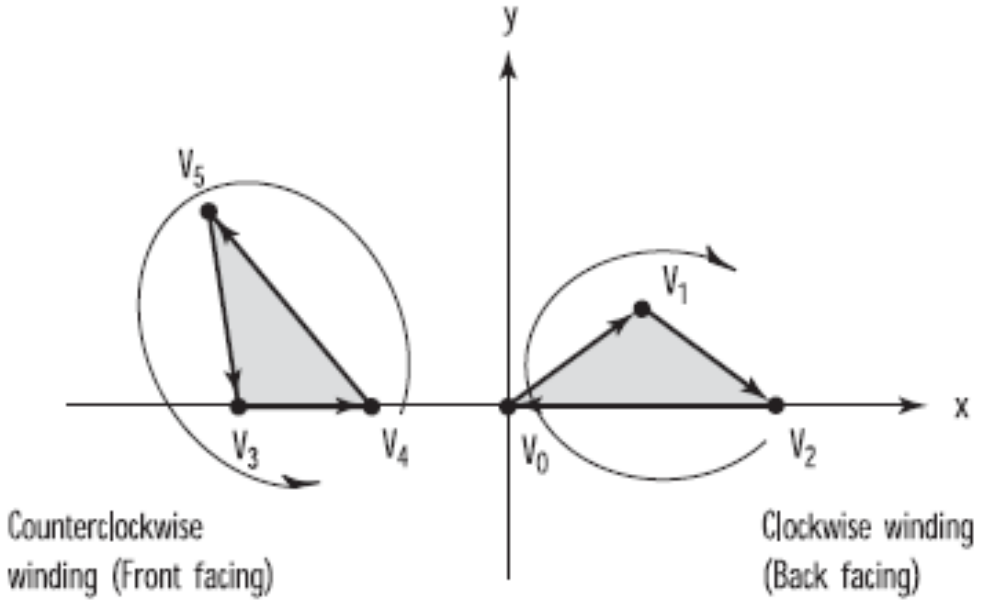
```
glBegin(GL_TRIANGLES);
    glVertex2f(0.0f, 0.0f); // V0
    glVertex2f(25.0f, 25.0f); // V1
    glVertex2f(50.0f, 0.0f); // V2
    glVertex2f(-50.0f, 0.0f); // V3
    glVertex2f(-75.0f, 50.0f); // V4
    glVertex2f(-25.0f, 0.0f); // V5
glEnd();
```



The triangles are said to have *clockwise winding* because they are literally wound in the clockwise direction.

- Triangles are the preferred primitive for object composition because most OpenGL hardware specifically accelerates triangles.

Two triangles with different windings



- By default, considers polygons that have counterclockwise winding to be front facing,
  - and the one on the right shows the back of the triangle.
- Why is this issue important?
  - You will often want to give the front and back of a polygon different physical characteristics.
  - You can hide the back of a polygon altogether or give it a different color and reflective property

- It’s important to keep the winding of all polygons in a scene consistent, using front-facing polygons to draw the outside surface of any solid objects.

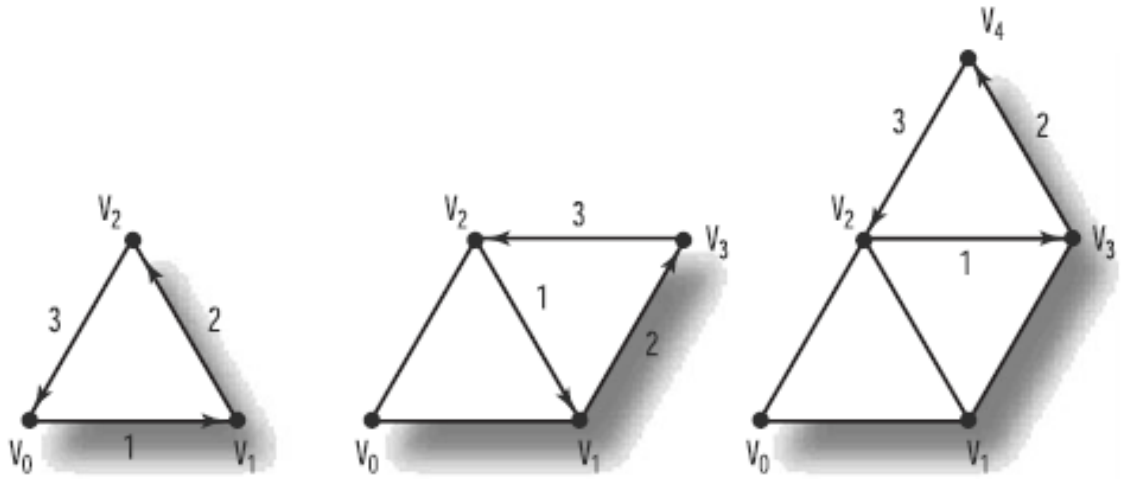
- If you need to reverse the default behavior of OpenGL, you can do so by calling the following function:

  glFrontFace(GL_CW);

- The GL_CW parameter tells OpenGL that clockwise-wound polygons are to be considered front facing.

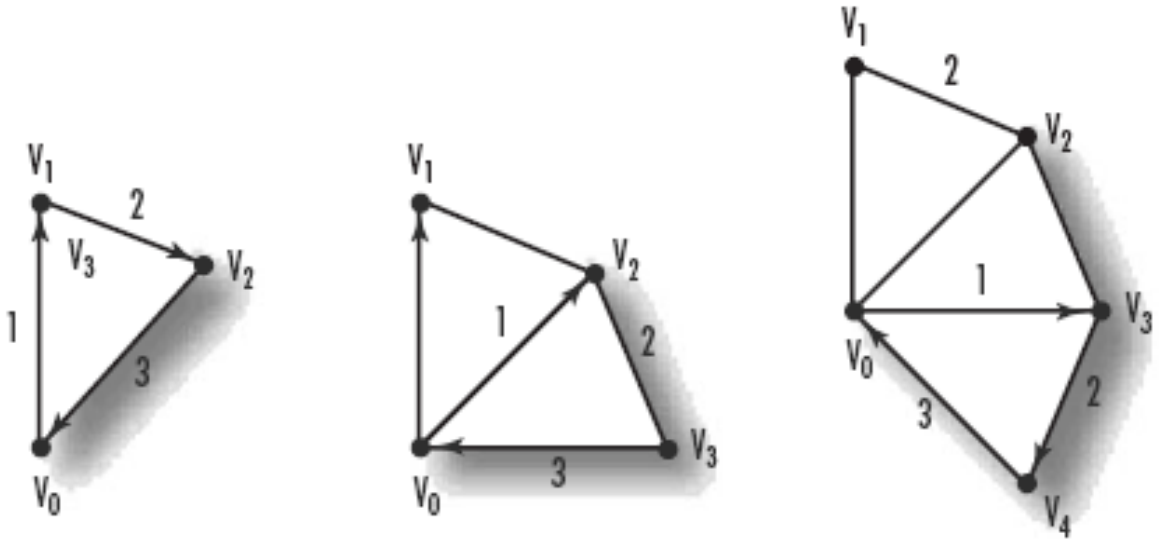
- To change back to counterclockwise winding for the front face, use GL_CCW.

GL_TRIANGLE_STRIP



- The figure shows the progression of a strip of three triangles specified by a set of five vertices numbered V0 through V4. Here, you see that the vertices are not necessarily traversed in the same order in which they were specified.
- The reason for this is to preserve the winding (counterclockwise) of each triangle.

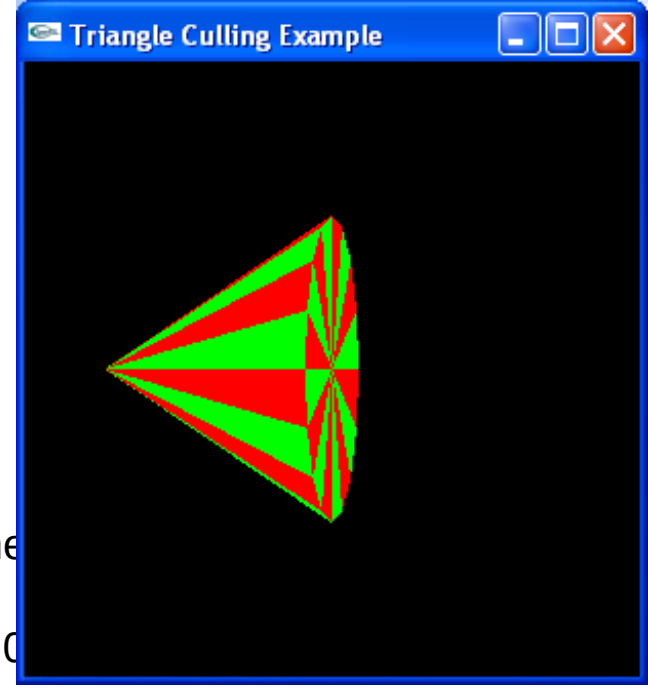
GL_TRIANGLE_FAN



- The figure shows a fan of three triangles produced by specifying four vertices

- // Clockwise-wound polygons are front facing; this is reversed
  // because we are using triangle fans
  glFrontFace(GL_CW);

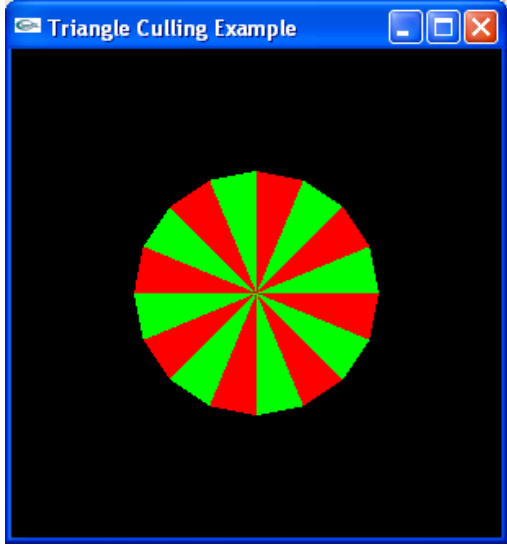
Triangle.cpp



- Draw cone

```
glBegin(GL_TRIANGLE_FAN);

// moved up Z axis
// to produce a cone instead of a circle
glVertex3f(0.0f, 0.0f, 75.0f);

// Loop around in a circle and specify even points along the
// as the vertices of the triangle fan
for(angle = 0.0f; angle < (2.0f*GL_PI); angle += (GL_PI/8.0
        // Calculate x and y position of the next vertex
        x = 50.0f*sin(angle);y = 50.0f*cos(angle);

        // Alternate color between red and green
        if((iPivot %2) == 0)   glColor3f(0.0f, 1.0f, 0.0f);
        else                   glColor3f(1.0f, 0.0f, 0.0f);

        iPivot++; // Increment pivot to change color next time

        glVertex2f(x, y); // Specify the next vertex for the triangle fan
        }

// Done drawing fan for cone
glEnd();
```

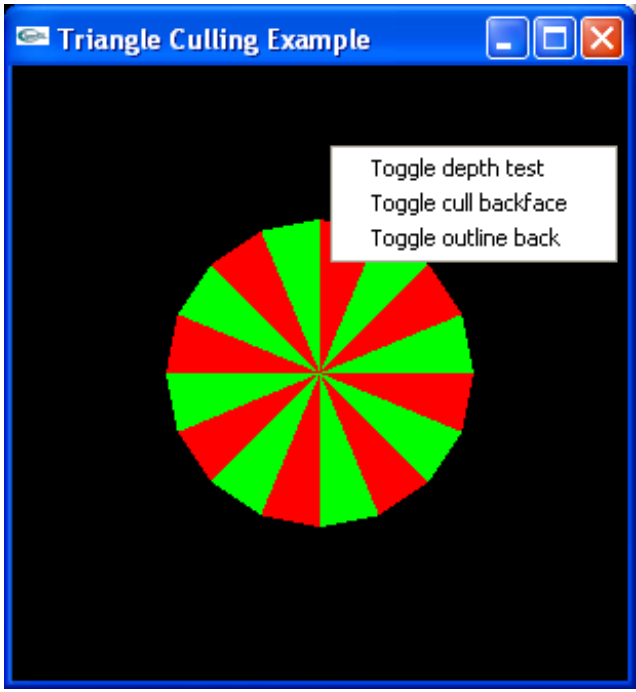
Triangle.cpp (ch3)



- Draw circle

```
glBegin(GL_TRIANGLE_FAN);

// Center of fan is at the origin
glVertex2f(0.0f, 0.0f);
for(angle = 0.0f; angle < (2.0f*GL_PI); angle += (GL_PI/8.0f)) {
        // Calculate x and y position of the next vertex
        x = 50.0f*sin(angle);          y = 50.0f*cos(angle);

        // Alternate color between red and green
        if((iPivot %2) == 0)           glColor3f(0.0f, 1.0f, 0.0f);
        else                   glColor3f(1.0f, 0.0f, 0.0f);

        iPivot++;              // Increment pivot to change color next time

        glVertex2f(x, y); // Specify the next vertex for the triangle fan
        }
// Done drawing the Circle
glEnd();
```

Add Menu

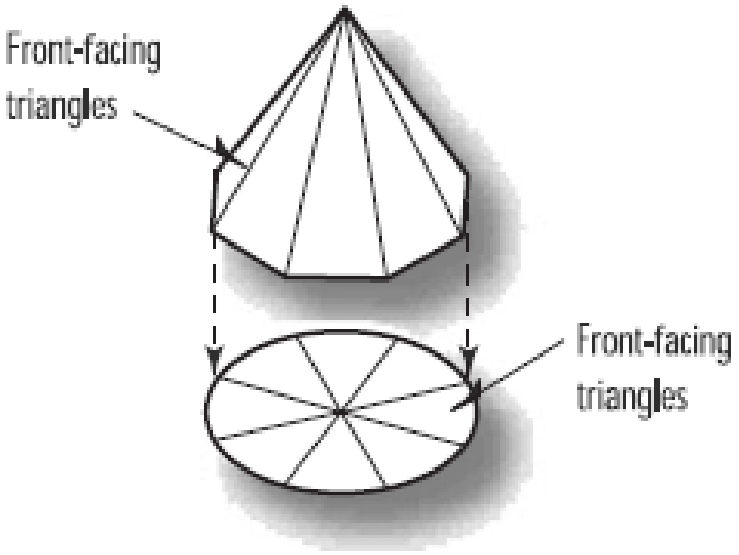


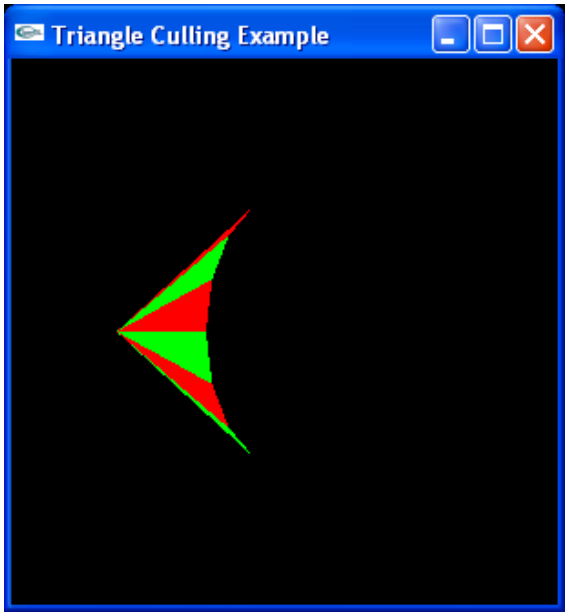

```
void ProcessMenu(int value) {
    switch(value)    {
        case 1: iDepth = !iDepth; break;

        case 2: iCull = !iCull; break;

        case 3: iOutline = !iOutline;

        default: break;
        }
    glutPostRedisplay();
    }
int main(int argc, char* argv[]) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutCreateWindow("Triangle Culling Example");

    // Create the Menu
    glutCreateMenu(ProcessMenu);
    glutAddMenuEntry("Toggle depth test",1);
    glutAddMenuEntry("Toggle cull backface",2);
    glutAddMenuEntry("Toggle outline back",3);
    glutAttachMenu(GLUT_RIGHT_BUTTON);
```

Culling: Hiding Surfaces for Performance

- *Culling* is the term used to describe the technique of eliminating geometry that we know will never be seen.

```
// Turn culling on if flag is set
if(iCull) glEnable(GL_CULL_FACE);
else      glDisable(GL_CULL_FACE);
```



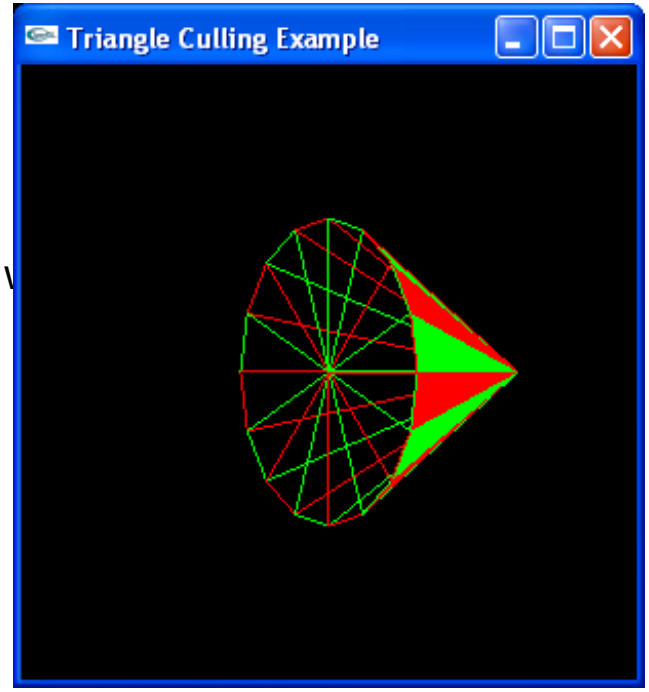


Polygon Modes

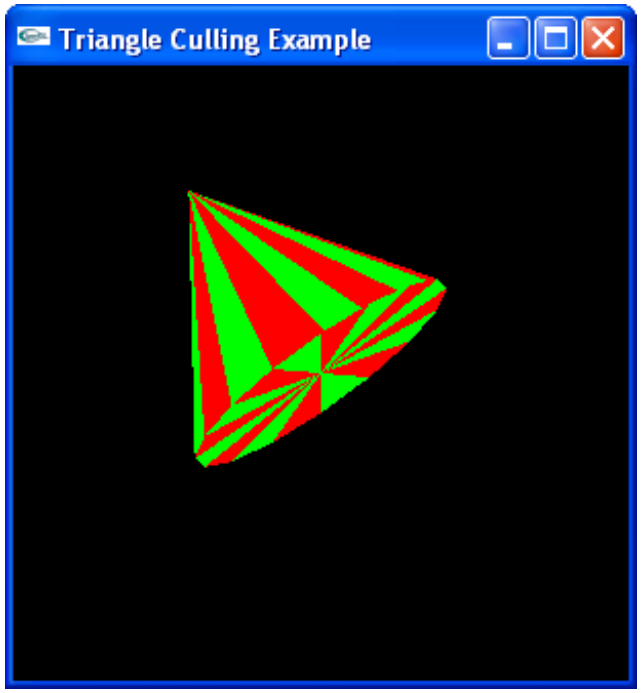
- By default, polygons are drawn solid, but you can change this behavior.
- The function glPolygonMode allows polygons to be rendered as filled solids, as outlines, or as points only.
- In addition, you can apply this rendering mode to both sides of the polygons or only to the front or back.
- // Draw back side as a polygon only, if flag is set

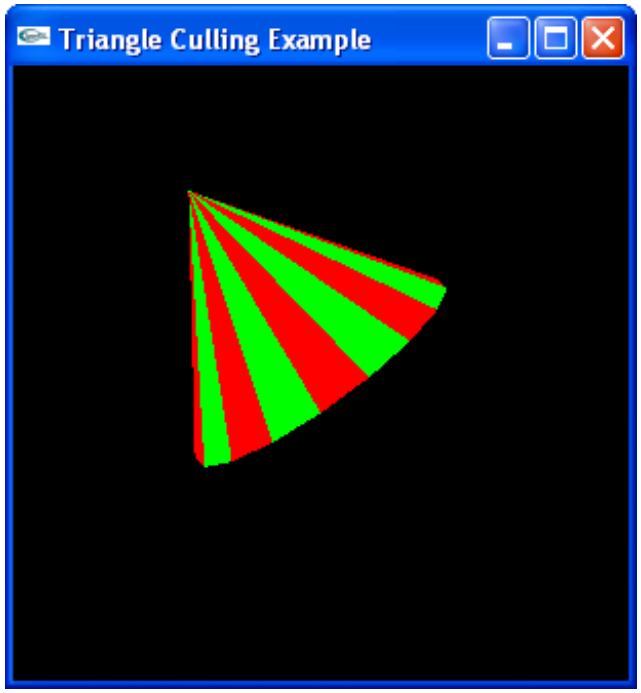
```
if(bOutline)
    glPolygonMode(GL_BACK,GL_LINE);
else
    glPolygonMode(GL_BACK,GL_FILL);
```

- We had to disable culling to produce this image;
  - otherwise, the inside would be eliminated and you v



Hidden Surface Removal





- You should request a depth buffer when you set up your OpenGL window with GLUT. For example, you can request a color and a depth buffer like this:
- glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH);

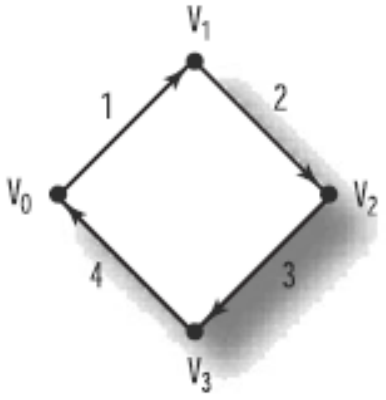
```
if(bDepth) glEnable(GL_DEPTH_TEST);
else       glDisable(GL_DEPTH_TEST);
```

- // Clear the window and the depth buffer

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

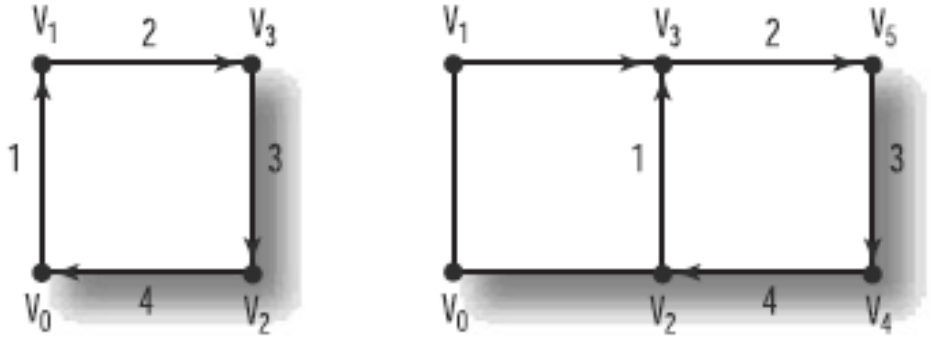
Four-Sided Polygons: Quads

- OpenGL's GL_QUADS primitive draws a four-sided polygon.
- One important rule to bear in mind when you use quads is that all four corners of the quadrilateral must lie in a plane (no bent quads!).



Quad Strips (GL_QUAD_STRIP)

- The figure shows the progression of a quad strip specified by six vertices. Note that these quad strips maintain a clockwise winding.



Line Details

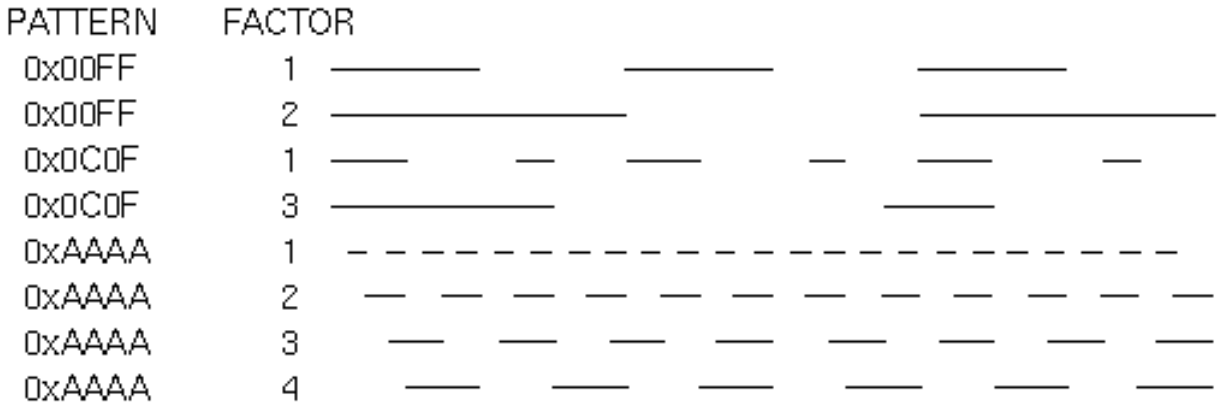
- you can specify lines with different widths and lines that are *stippled* in various ways - dotted, dashed, drawn with alternating dots and dashes, and so on.
- void **glLineWidth**(GLfloat *width*); Sets the width in pixels for rendered lines; *width* must be greater than 0.0 and by default is 1.0.

```
GLfloat sizes[2]; // Store supported line width range
GLfloat step; // Store supported line width increments

// Get supported line width range and step size
glGetFloatv(GL_LINE_WIDTH_RANGE,sizes);
glGetFloatv(GL_LINE_WIDTH_GRANULARITY,&step);
```

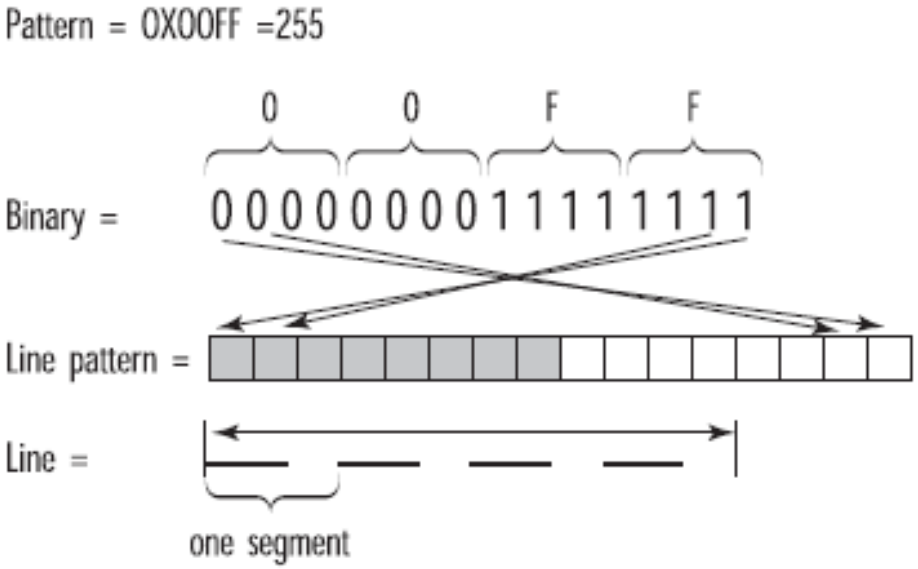
- The Microsoft implementation of OpenGL allows for line widths from 0.5 to 10.0, with 0.125 the smallest step size.

- **Stippled Lines (dotted or dashed)**
- glLineStipple(1, 0x3F07); glEnable(GL_LINE_STIPPLE);
- void **glLineStipple**(GLint *factor*, GLushort *pattern*);
- The *pattern* argument is a 16-bit series of 0s and 1s, and it's repeated as necessary to stipple a given line. A 1 indicates that drawing occurs, and 0 that it does not, on a pixel-by-pixel basis, beginning with the low-order bits of the pattern. The pattern can be stretched out by using *factor.*

- With the preceding example and the pattern 0x3F07 (which translates to 0011111100000111 in binary), a line would be drawn with 3 pixels on, then 5 off, 6 on, and 2 off.
- If *factor* had been 2, the pattern would have been elongated: 6 pixels on, 10 off, 12 on, and 4 off.



A stipple pattern is used to construct a line segment



- You might wonder why the bit pattern for stippling is used in reverse when the line is drawn. Internally, it’s much faster for OpenGL

Using Line Stipple Patterns
lines.c

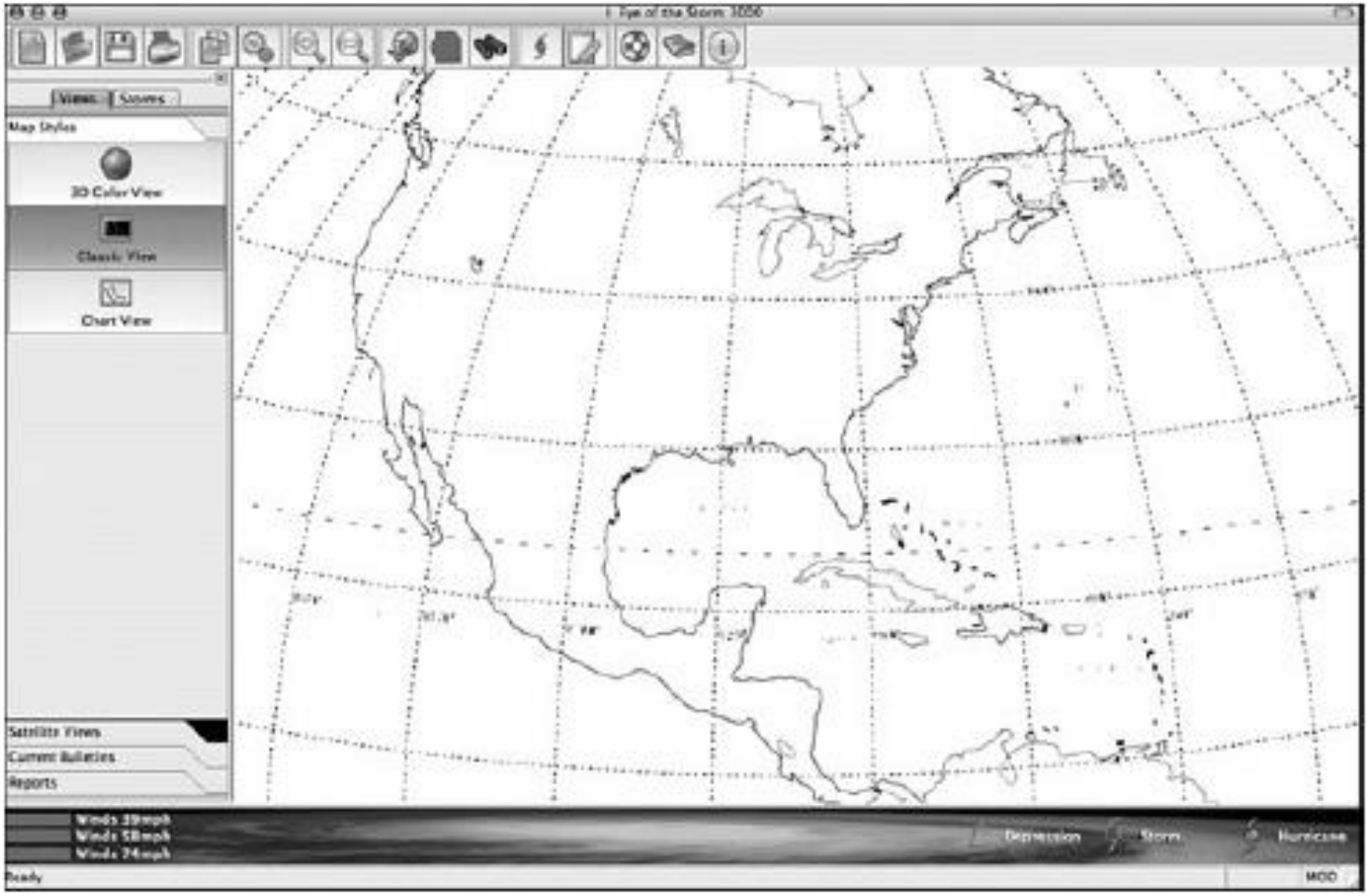
```
#include <GL/glut.h>

#define   drawOneLine(x1,y1,x2,y2) glBegin(GL_LINES);
   glVertex2f ((x1),(y1)); glVertex2f ((x2),(y2)); glEnd();

void myinit (void) {
    /*  background to be cleared to black   */
    glClearColor (0.0, 0.0, 0.0, 0.0);
    glShadeModel (GL_FLAT);
}

void display(void)
{
    int i;

    glClear (GL_COLOR_BUFFER_BIT);
/*  draw all lines in white   */
    glColor3f (1.0, 1.0, 1.0);
```

```
/*  in 1st row, 3 lines, each with a different stipple   */
    glEnable (GL_LINE_STIPPLE);
    glLineStipple (1, 0x0101);   /*  dotted   */
    drawOneLine (50.0, 125.0, 150.0, 125.0);
    glLineStipple (1, 0x00FF);   /*  dashed   */
    drawOneLine (150.0, 125.0, 250.0, 125.0);
    glLineStipple (1, 0x1C47);   /*  dash/dot/dash   */
    drawOneLine (250.0, 125.0, 350.0, 125.0);

/*  in 2nd row, 3 wide lines, each with different stipple */
    glLineWidth (5.0);
    glLineStipple (1, 0x0101);
    drawOneLine (50.0, 100.0, 150.0, 100.0);
    glLineStipple (1, 0x00FF);
    drawOneLine (150.0, 100.0, 250.0, 100.0);
    glLineStipple (1, 0x1C47);
    drawOneLine (250.0, 100.0, 350.0, 100.0);
    glLineWidth (1.0);
```

```
/*  in 3rd row, 6 lines, with dash/dot/dash stipple,   */
/*  as part of a single connected line strip         */
    glLineStipple (1, 0x1C47);
    glBegin (GL_LINE_STRIP);
    for (i = 0; i < 7; i++)
   glVertex2f (50.0 + ((GLfloat) i * 50.0), 75.0);
    glEnd ();

/*  in 4th row, 6 independent lines,   */
/*  with dash/dot/dash stipple         */
    for (i = 0; i < 6; i++) {
   drawOneLine (50.0 + ((GLfloat) i * 50.0),
      50.0, 50.0 + ((GLfloat)(i+1) * 50.0), 50.0);
    }

/*  in 5th row, 1 line, with dash/dot/dash stipple   */
/*  and repeat factor of 5         */
    glLineStipple (5, 0x1C47);
    drawOneLine (50.0, 25.0, 350.0, 25.0);
    glFlush ();
}
```

```
int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGBA);
    auxInitPosition (0, 0, 400, 150);
    auxInitWindow (argv[0]);
    myinit ();
    auxMainLoop(display);
}
```

A 3D map rendered with solid and stippled lines.



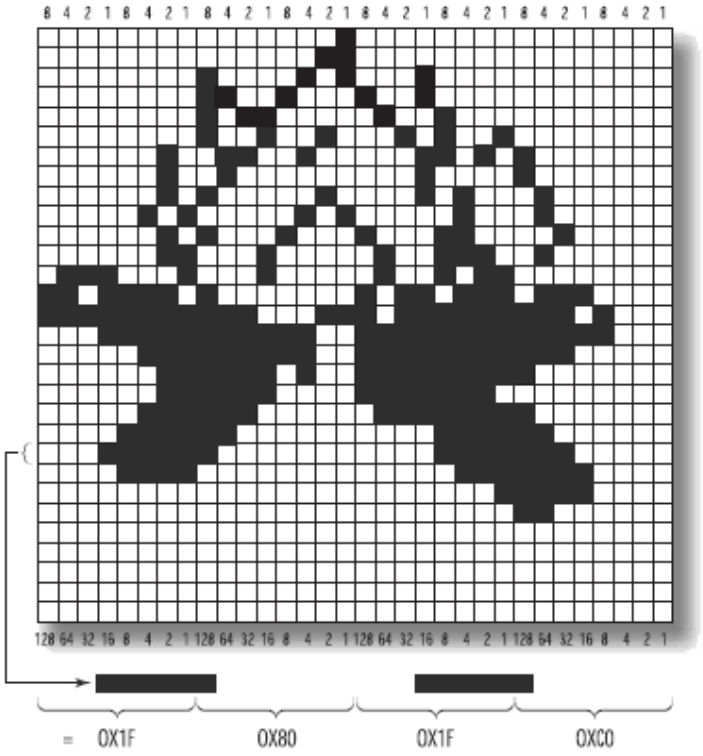
Filling Polygons,

- There are two methods for applying a pattern to solid polygons.
  - The customary method is texture mapping
  - Another way is to specify a stippling pattern, as we did for lines. A polygon stipple pattern is nothing more than a 32×32 monochrome bitmap that is used for the fill pattern.

- To enable polygon stippling, call
  - glEnable(GL_POLYGON_STIPPLE);

  and then call
  - glPolygonStipple(pBitmap);

- pBitmap is a pointer to a data area containing the stipple pattern. Hereafter, all polygons are filled using the pattern specified by pBitmap (GLubyte *).
- This pattern is similar to that used by line stippling, except the buffer is large enough to hold a 32-by-32-bit pattern.
- Also, the bits are read with the most significant bit (MSB) first, which is just the opposite of line stipple patterns.



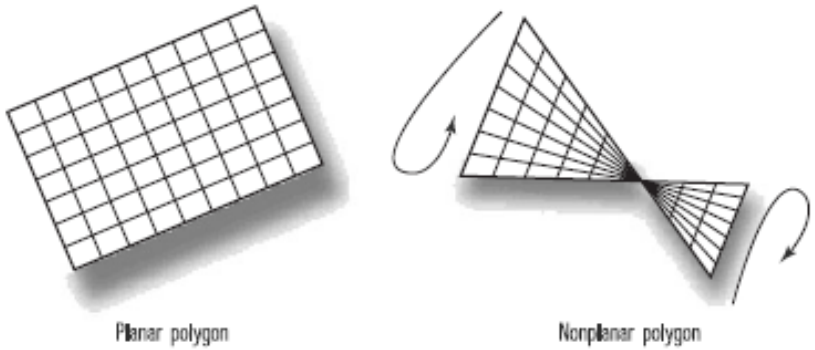

```
// Bitmap of campfire
GLubyte fire[] = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                   0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                   0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                   0x00, 0x00, 0x00, 0xc0, 0x00, 0x00, 0x01, 0xf0,
                   0x00, 0x00, 0x07, 0xf0, 0x0f, 0x00, 0x1f, 0xe0,
                   0x1f, 0x80, 0x1f, 0xc0, 0x0f, 0xc0, 0x3f, 0x80,
                   0x07, 0xe0, 0x7e, 0x00, 0x03, 0xf0, 0xff, 0x80,
                   0x03, 0xf5, 0xff, 0xe0, 0x07, 0xfd, 0xff, 0xf8,
                   0x1f, 0xfc, 0xff, 0xe8, 0xff, 0xe3, 0xbf, 0x70,
                   0xde, 0x80, 0xb7, 0x00, 0x71, 0x10, 0x4a, 0x80,
                   0x03, 0x10, 0x4e, 0x40, 0x02, 0x88, 0x8c, 0x20,
                   0x05, 0x05, 0x04, 0x40, 0x02, 0x82, 0x14, 0x40,
                   0x02, 0x40, 0x10, 0x80, 0x02, 0x64, 0x1a, 0x80,
                   0x00, 0x92, 0x29, 0x00, 0x00, 0xb0, 0x48, 0x00,
                   0x00, 0xc8, 0x90, 0x00, 0x00, 0x85, 0x10, 0x00,
                   0x00, 0x03, 0x00, 0x00, 0x00, 0x00, 0x10, 0x00 };
```

PStipple.cpp (ch3)

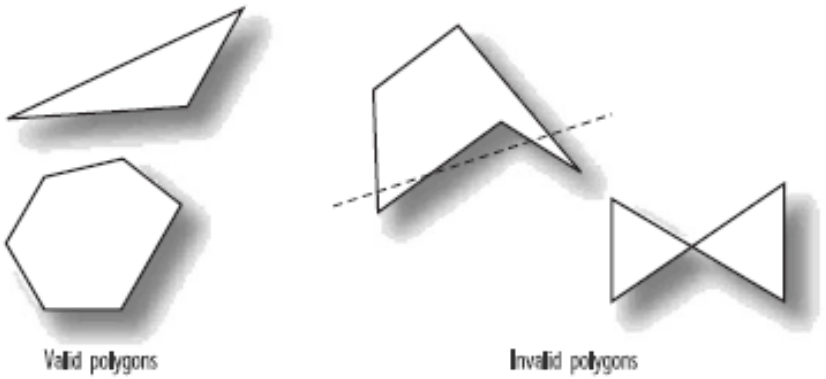
```
void SetupRC() {
…
// Enable polygon stippling
glEnable(GL_POLYGON_STIPPLE);

// Specify a specific stipple pattern
glPolygonStipple(fire);
}

void RenderScene(void) {
...
// Begin the stop sign shape,
// use a standard polygon for simplicity
glBegin(GL_POLYGON);
        glVertex2f(-20.0f, 50.0f);
        glVertex2f(20.0f, 50.0f);
        glVertex2f(50.0f, 20.0f);
        glVertex2f(50.0f, -20.0f);
        glVertex2f(20.0f, -50.0f);
        glVertex2f(-20.0f, -50.0f);
        glVertex2f(-50.0f, -20.0f);
        glVertex2f(-50.0f, 20.0f);
glEnd();
...
glutSwapBuffers ();
}
```

Polygon Construction Rules

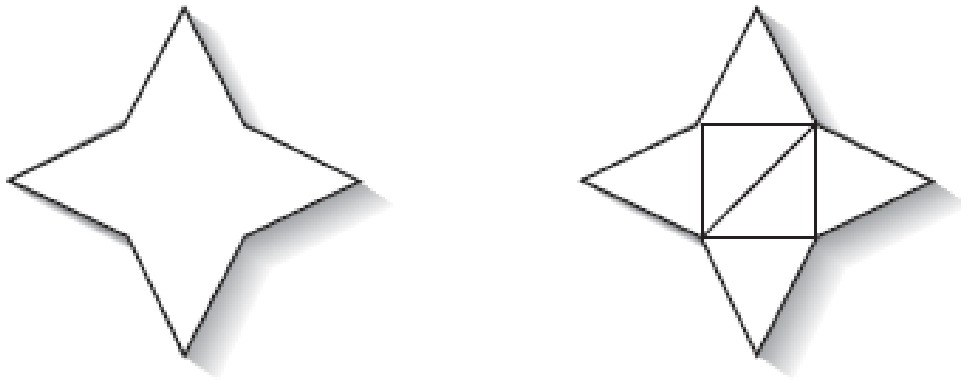
- When you are using many polygons to construct a complex surface, you need to remember two important rules:
  - The first rule is that all polygons must be planar. That is, all the vertices of the polygon must lie in a single plane
    - Here is yet another good reason to use triangles.



- The second rule of polygon construction is that the polygon's edges must not intersect, and the polygon must be convex.
  - If any given line enters and leaves the polygon more than once, the polygon is not convex.



- Even though OpenGL can draw only convex polygons, there’s still a way to create a nonconvex polygon: by arranging two or more convex polygons together.



Polygons as Points, Outlines, or Solids

void **glPolygonMode**(GLenum **face**, GLenum **mode**);

The parameter *face* can be GL_FRONT_AND_BACK, GL_FRONT, or GL_BACK;

mode can be GL_POINT, GL_LINE, or GL_FILL

For example, you can have the front faces filled and the back faces outlined with two calls to this routine:

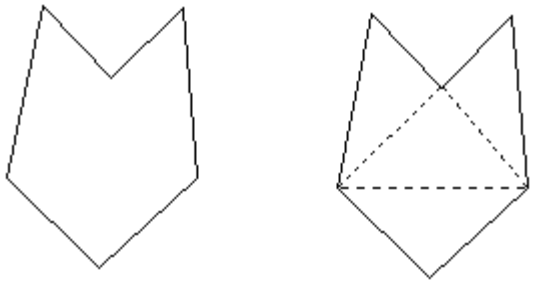
```
glPolygonMode(GL_FRONT, GL_FILL);
glPolygonMode(GL_BACK, GL_LINE);
```

Culling Polygon Faces

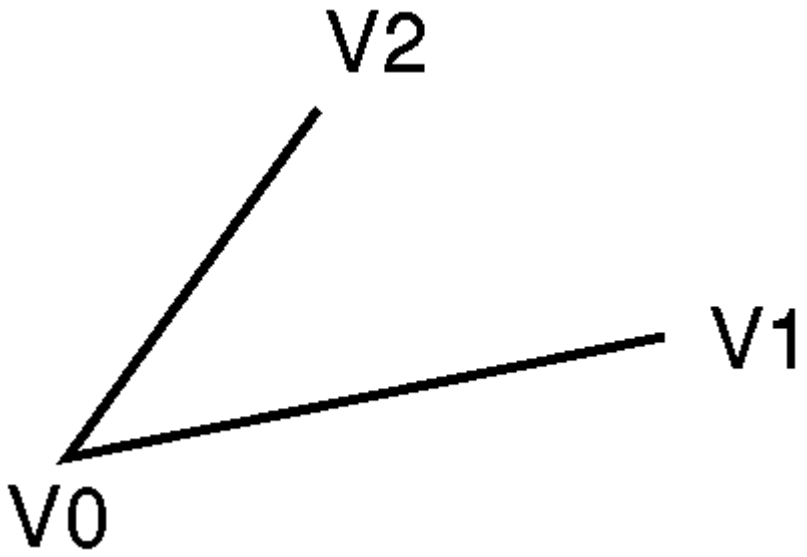
- By convention, polygons whose vertices appear in counterclockwise (CCW) order on the screen are called front-facing.
- You can swap what OpenGL considers the back face by using the function void **glFrontFace**(GLenum *mode*);
- By default, *mode* is GL_CCW. If *mode* is GL_CW, faces with a clockwise orientation are considered front-facing.
- In a completely enclosed surface constructed from polygons with a consistent orientation, none of the back-facing polygons are ever visible. You can maximize drawing speed by having OpenGL discard polygons as soon as it determines that they're back-facing.
- Similarly, if you are inside the object, only back-facing polygons are visible.

- To instruct OpenGL to discard front- or back-facing polygons, use the command **glCullFace**(GLenum *mode*); and enable culling with **glEnable()**.
- Indicates which polygons should be discarded (culled) before they're converted to screen coordinates.
- The mode is either GL_FRONT, GL_BACK, or GL_FRONT_AND_BACK
- Culling must be enabled using **glEnable()** with GL_CULL_FACE; it can be disabled with **glDisable()** and the same argument.

Marking Polygon Boundary Edges



- OpenGL can render only convex polygons, but many nonconvex polygons arise in practice. To draw these nonconvex polygons, you typically subdivide them into convex polygons - usually triangles
- Unfortunately, if you decompose a general polygon into triangles and draw the triangles, you can't really use **glPolygonMode()** to draw the polygon's outline.
- OpenGL keeps track of this information by passing along with each vertex a bit indicating whether that vertex is followed by a boundary edge.
- You can manually control the setting of the edge flag with the command **glEdgeFlag**(GLboolean *flag*);




```
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
glBegin(GL_POLYGON);
    glEdgeFlag(GL_TRUE);
    glVertex3fv(V0);
    glEdgeFlag(GL_FALSE);
    glVertex3fv(V1);
    glEdgeFlag(GL_TRUE);
    glVertex3fv(V2);
glEnd();
```

Normal Vectors

- An object's normal vectors define the orientation of its surface in space- in particular, its orientation relative to light sources.

```
 glBegin (GL_POLYGON);
   glNormal3fv(n0);
   glVertex3fv(v0);
   glNormal3fv(n1);
   glVertex3fv(v1);
   glNormal3fv(n2);
   glVertex3fv(v2);
   glNormal3fv(n3);
   glVertex3fv(v3);
glEnd();
```

Calculating Normal Vectors

1. **Finding Normal for Analytic Surfaces**

2. **Finding Normal from Polygonal Data**

1-Finding Normals for Analytic Surfaces

- **V**(**s,t**) = [**X**(**s,t**) **Y**(**s,t**) **Z**(**s,t**)]
- To calculate the normal, find $\frac{\partial V}{\partial s}, \frac{\partial V}{\partial t}$
- which are vectors tangent to the surface in the *s* and *t* directions. The cross product $\frac{\partial V}{\partial s} \times \frac{\partial V}{\partial t}$

$$\begin{bmatrix} v_x & v_y & v_z \end{bmatrix} \times \begin{bmatrix} w_x & w_y & w_z \end{bmatrix} = \begin{bmatrix} (v_y w_z - w_y v_z) & (w_x v_z - v_x w_z) & (v_x w_y - w_x v_y) \end{bmatrix}$$

- As an example of these calculations, consider the analytic surface
- **V**(**s**,**t**) = [$\mathbf{s}^2$ $\mathbf{t}^3$ 3-**st**]
- From this we have

$$\frac{\partial V}{\partial s} = \left[2s \;\; 0 \;\; -t\right], \frac{\partial V}{\partial t} = \left[0 \;\; 3t^2 \;\; -s\right], \text{and } \frac{\partial V}{\partial s} \times \frac{\partial V}{\partial t} = \left[-3t^3 \;\; 2s^2 \;\; 6st^2\right]$$

- when **s**=1 and **t**=2, the corresponding point on the surface is (1, 8, 1), and the vector (-24, 2, 24) is perpendicular to the surface at that point. The length of this vector is 34, so the unit normal vector is (-24/34, 2/34, 24/34) = (-0.70588, 0.058823, 0.70588).

- For analytic surfaces that are described implicitly, as **F**(**x, y, z**) = 0, the problem is harder.
- In some cases, you can solve for one of the variables, say **z** = **G**(**x, y**), and put it in the explicit form given previously:
  - **V**(**s,t**) = [**s t G(s,t)**]
- If you can't get the surface equation in an explicit form, you might be able to make use of the fact that the normal vector is given by the gradient.

$$\nabla F = \left[\frac{\partial F}{\partial x}\ \frac{\partial F}{\partial y}\ \frac{\partial F}{\partial z}\right]$$

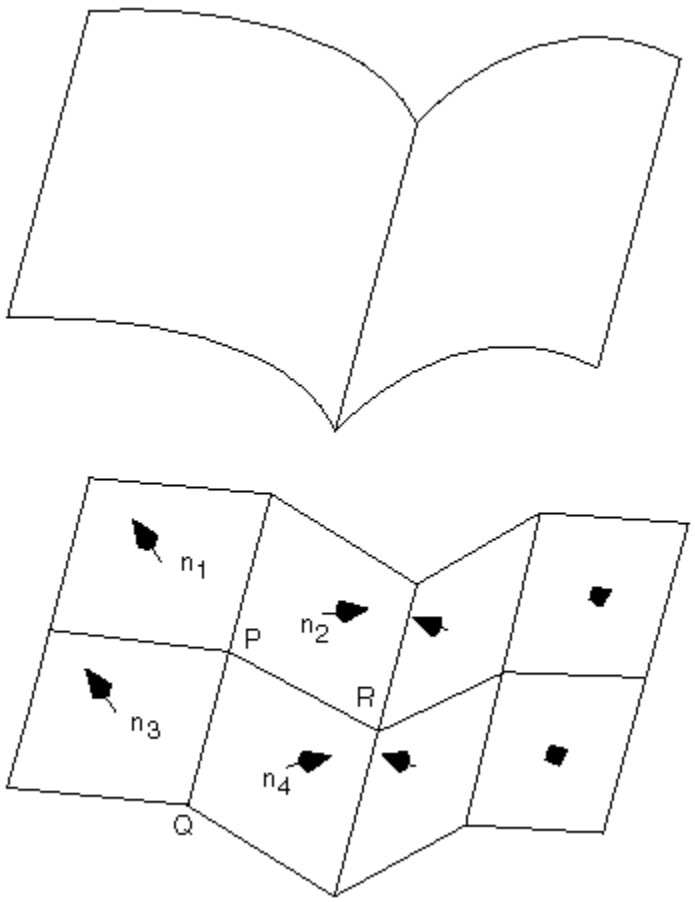
- Calculating the gradient might be easy, but finding a point that lies on the surface can be difficult.
- As an example of an implicitly defined analytic function, consider the equation of a sphere of radius 1 centered at the origin:
- $X^2 + y^2 + z^2 - 1 = 0$
- This means that
- $F(x, y, z) = x^2 + y^2 + z^2 - 1$
- which can be solved for z to yield $z=\pm\text{sqrt}(1- x^2 - y^2)$

- If you could not solve for *z*, you could have used the gradient. $\nabla F = \begin{bmatrix} 2x & 2y & 2z \end{bmatrix}$
- as long as you could find a point on the surface. In this case, it's not so hard to find a point - for example, (2/3, 1/3, 2/3) lies on the surface. Using the gradient, the normal at this point is (4/3, 2/3, 4/3). The unit-length normal is (2/3, 1/3, 2/3), which is the same as the point on the surface, as expected.

Finding Normals from Polygonal Data

- To find the normal for a flat polygon, take any three vertices **v**1, **v**2, and **v**3 of the polygon that do not lie in a straight line. The cross product [*v*1 - *v*2] × [*v*2 - *v*3] is perpendicular to the polygon.

- You need to average the normals for adjoining facets.
- If **n**1, **n**2, **n**3, and **n**4 are the normals for the four polygons meeting at point P, calculate **n**1+**n**2+**n**3+**n**4 and then normalize it. The resulting vector can be used as the normal for point P.
- If you're drawing a cube or a cut diamond, for example - don't do the averaging



An Example: Building an Icosahedron

- An icosahedron can be considered a rough approximation for a sphere.

```
#define X .525731112119133606
#define Z .850650808352039932

static GLfloat vdata[12][3] = {
   {-X, 0.0, Z}, {X, 0.0, Z}, {-X, 0.0, -Z}, {X, 0.0, -Z},
   {0.0, Z, X}, {0.0, Z, -X}, {0.0, -Z, X}, {0.0, -Z, -X},
   {Z, X, 0.0}, {-Z, X, 0.0}, {Z, -X, 0.0}, {-Z, -X, 0.0}
};

static GLint tindices[20][3] = {
   {0,4,1}, {0,9,4}, {9,5,4}, {4,5,8}, {4,8,1},
   {8,10,1}, {8,3,10}, {5,3,8}, {5,2,3}, {2,7,3},
   {7,10,3}, {7,6,10}, {7,11,6}, {11,0,6}, {0,1,6},
   {6,1,10}, {9,0,11}, {9,11,2}, {9,2,5}, {7,2,11} };

for (i = 0; i < 20; i++) {
   /* color information here */
   glBegin(GL_TRIANGLES);
      glVertex3fv(&vdata[tindices[i][0]][0]);
      glVertex3fv(&vdata[tindices[i][1]][0]);
      glVertex3fv(&vdata[tindices[i][2]][0]);
   glEnd();
}
```

- The line that mentions color information should be replaced by a command that sets the color of the **i**th face. If no code appears here, all faces are drawn in the same color, and it'll be impossible to discern the three-dimensional quality of the object.

- An alternative to explicitly specifying colors is to define surface normals and use lighting.

Supplying Normals for an Icosahedron

```
GLfloat d1[3], d2[3], norm[3];
for (j = 0; j < 3; j++) {
   d1[j] = vdata[tindices[i][0]][j] - vdata[tindices[i][1]][j];
   d2[j] = vdata[tindices[i][1]][j] - vdata[tindices[i][2]][j];
}
normcrossprod(d1, d2, norm);
glNormal3fv(norm);
```

Calculating the Normalized Cross Product of Two Vectors

```
void normalize(float v[3]) {
   GLfloat d = sqrt(v[0]*v[0]+v[1]*v[1]+v[2]*v[2]);
   if (d == 0.0) {
      error("zero length vector");
      return;
   }
   v[0] /= d; v[1] /= d; v[2] /= d;
}

void normcrossprod(float v1[3], float v2[3], float out[3])
{
   GLint i, j;
   GLfloat length;

   out[0] = v1[1]*v2[2] - v1[2]*v2[1];
   out[1] = v1[2]*v2[0] - v1[0]*v2[2];
   out[2] = v1[0]*v2[1] - v1[1]*v2[0];
   normalize(out);
}
```

- If you're using an icosahedron as an approximation for a shaded sphere, you'll want to use normal vectors that are perpendicular to the true surface of the sphere

```
for (i = 0; i < 20; i++) {
   glBegin(GL_POLYGON);
      glNormal3fv(&vdata[tindices[i][0]][0]);
      glVertex3fv(&vdata[tindices[i][0]][0]);
      glNormal3fv(&vdata[tindices[i][1]][0]);
      glVertex3fv(&vdata[tindices[i][1]][0]);
      glNormal3fv(&vdata[tindices[i][2]][0]);
      glVertex3fv(&vdata[tindices[i][2]][0]);
   glEnd();
}
```

Cube1.c

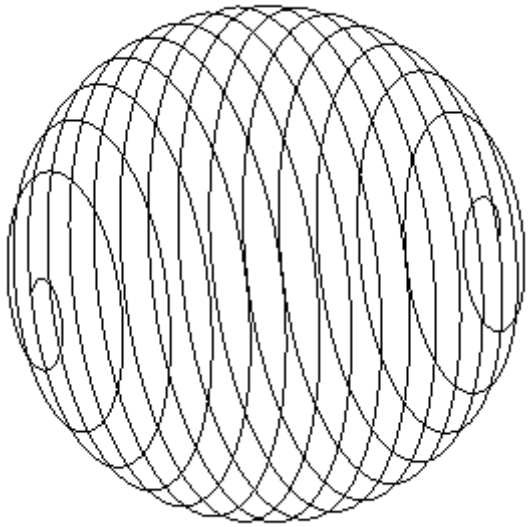
```
#include <GL/glut.h>

    int a[3]={10,10,10}, b[3]={10,-10,10},
    c[3]={-10,-10,10}, d[3]={-10,10,10},
    e[3]={10,10,-10}, f[3]={10,-10,-10},
    g[3]={-10,-10,-10}, h[3]={-10,10,-10};

    float angle=1.0;

    void drawcube(void)
    {
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(1.0, 1.0, 1.0);

    glMatrixMode(GL_MODELVIEW);
    glRotatef(angle, 0.0, 1.0, 0.0);
    glBegin(GL_LINE_LOOP);
            glVertex3iv(a);
            glVertex3iv(b);
            glVertex3iv(c);
            glVertex3iv(d);
    glEnd();
```

```
glBegin(GL_LINE_LOOP);
    glVertex3iv(a);
    glVertex3iv(e);
    glVertex3iv(f);
    glVertex3iv(b);
glEnd();
glBegin(GL_LINE_LOOP);
    glVertex3iv(d);
    glVertex3iv(h);
    glVertex3iv(g);
    glVertex3iv(c);
glEnd();
glBegin(GL_LINE_LOOP);
    glVertex3iv(e);
    glVertex3iv(f);
    glVertex3iv(g);
    glVertex3iv(h);
glEnd();
glFlush();
glutSwapBuffers();}
```

- ```
  void keyboard(unsigned char key, int x, int y)
  {
  switch (key){
  case 'q':
  case 'Q':
          exit(0);
          break;}
  }

  void mouse(int btn, int state, int x, int y)
  {
  if (state == GLUT_DOWN)
  {
  if (btn == GLUT_LEFT_BUTTON)
          angle = angle + 1.0f;
  else if (btn == GLUT_RIGHT_BUTTON)
          angle = angle - 1.0f;
  else
          angle = 0.0f;
  }
  }
  ```
  ```

- ```
  int main(int argc, char **argv)
  {
  glutInit(&argc, argv);
  glutInitWindowSize(500, 500);
  glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE);
  glutCreateWindow("Glut rotate");
  glutMouseFunc(mouse);
  glutKeyboardFunc(keyboard);
  glutDisplayFunc(drawcube);
  glutIdleFunc(drawcube);

  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glOrtho(-30.0, 30.0, -30.0, 30.0, -30.0, 30.0);
  glRotatef(30.0, 1.0, 0.0, 0.0);
  glMatrixMode(GL_MODELVIEW);
  glClearColor(0.0, 0.0, 0.0, 0.0);

  glutMainLoop();
  return(0);
  }
  ```
  ```



```
glBegin(GL_LINE_STRIP);
r=50.0f;
angle = 0.0f;
for (z = -r; z <r ; z += .1f) {
//x = r cos(theta) cos(phi)
//y = r cos(theta) sin(phi)
//z = r sin(theta)

sin_theta = z/r;
cos_theta = sqrt(1-sin_theta*sin_theta);
//C runtime functions sin() and cos() accept angle values measured in radians
x = r*sin(angle)*cos_theta;
y = r*cos(angle)*cos_theta;
angle += 0.1f;
// Specify the point and move the Z value up a little
glVertex3f(x, y, z);
}
// Done drawing points
glEnd();
```