

# Introduction to OpenGL and 3D Graphics

## Categories

- Introduction to graphics
- The History of 3D Graphics
- How 3D Graphics work
- Drawing Points, Lines and Polygons in OpenGL
- Transformations and Perspective

## Introduction to Graphics: Your screen

- Your screen is made of individual colored pixels. (640x480 during this presentation)
- We generally assign each of these pixels a color with red, green and blue (RGB) intensities.
- Today (most of the time) these are in "millions of colors" with 24 bits with 8 bits per channel.
- When we draw to the screen, we generally deposit these color values inside a memory buffer or "frame buffer" which your graphics card turns into signals for your display.

## Introduction to Graphics: 2D Graphics

- There are two basic methods of storing 2D graphics: bitmapped and vector.
- Bitmapped graphics take an image of a particular size and assign a color value to each pixel.
  - These cannot be enlarged without loss of quality, take lots of space.
  - ex: GIF
- Vector graphics use equations to represent lines and curves.
  - These can be enlarged as much as desired
  - ex: Adobe Illustrator

## Introduction to Graphics: Size of Bitmaps

- Suppose we have 640x480 image stored in "millions of colors".
  - There are  $640 \times 480 = 307,200$  pixels
  - A file would require are  $640 \times 480 \times 4 = 1.22$  MB (uncompressed)
- If we consider 1280x854 we get 4.37 MB per image
- With compression these sizes are okay with single images, but lets imagine we are trying to do the same in 3D.

## Introduction to Graphics: How we do 3D

- As you can tell from the previous slide, we cannot build an entire 3D world from bitmaps without using an enormous amount of space.
- For 3D, we use a vector graphics representation.
- We draw everything out of polygons. And wrap textures around them, color them and manipulate them. We can use their vertices as the basis of calculations. (for more take CS184)

## History of 3D Graphics-Early Days

- Not too long ago, there were almost no consumer 3D graphics applications.
- No Special 3D Hardware available to lay people
- Some early software only games: Doom, Wolf 3D.
- These early applications were severally restricted in what they could do by the power of the CPU.
  - 3D graphics take lots of processing power to do things like vector calculations, shading and texture mapping.

## History of Computer Graphics-Hardware Acceleration

- 3D Graphics uses very specialized and parallelizable calculations. These lend themselves to hardware implementations.
- Early hardware acceleration was only available for thousands of dollars.
- The first consumer solution was 3Dfx's Voodoo 1.
- The Voodoo 1 was very primitive, used a pass through cable through video card.
- Soon other manufacturers showed up in the consumer market (ex: Matrox, ATI, nVidia)

## History of Computers: Why OpenGL

- Many hardware manufacturers, each with its own hardware, drivers.
- Consumer software makers had to write their code to each individual card-very messy.
- OpenGL evolves from IRIS GL (SGI) which was originally designed for CAD.
- OpenGL can be software only or hardware accelerated and with proper drivers can support almost any 3D card.
- Microsoft did not like this idea, proposed and vigorously marketed its competing standard, DirectX.

## The Basic Idea (again)

- When we make 3D objects we will make them out of triangles (or other polygons). By placing the triangles side by side, we can create complex shapes.
- We can use textures and shading to make these objects look more real.
- As an introduction, we will clear the screen and then draw a 2D box.
- We will use a library called GLUT which is not actually part of OpenGL but is good for testing code and learning.

## Sample First Program: An Empty Window

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

void renderer(void);
void initialSetup(void);

int main (int argc, const char * argv[])
{
    glutInit(&argc, (char **)argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutCreateWindow("First Program");
    glutDisplayFunc(renderer);
    initialSetup();
    glutMainLoop();
    return 0;
}

void renderer(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glFlush();
}

void initialSetup(void)
{
    glClearColor(0.0, 0.0, .5, 1.0);
}
```

## What does this code mean? Part 1

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

Includes the GLUT and OpenGL libraries

```
void renderer(void);
void initialSetup(void);
```

Are the Standard Function Prototypes

- "renderer" is the function GLUT calls whenever it wants to redraw the window.
- "initialSetup" is a function that goes through the steps to set state machine parameters before we begin drawing.

## What does this code mean? Part 2

```
void initialSetup(void)
{
    glClearColor(0.0, 0.0, .5, 1.0);
}
```

- glClearColor sets the color for the window when it is cleared.

```
void renderer(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glFlush();
}
```

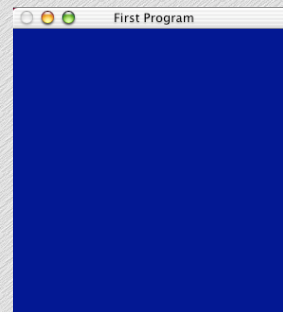
- "glClear" clears the entire window by purging the buffer.
- "glFlush" instructs OpenGL to carry out all unexecuted OpenGL commands (sequentially).

## What does this code mean? Part 3

```
int main (int argc, const char * argv[])
{
    glutInit(&argc, (char **)argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutCreateWindow("First Program");
    glutDisplayFunc(renderer);
    initialSetup();
    glutMainLoop();
    return 0;
}
```

- "glutCreateWindow" creates a new Window with the given name.
- "glutDisplayFunc" tells GLUT what function to call whenever it wants to redraw the window.
- "glutMainLoop" tells GLUT to begin running. It will automatically call "renderer" when it needs to redraw the screen.

## DEMO



## Drawing 2D Shapes

```
glRectf(GLfloat ulx, GLfloat uly, GLfloat lrx, GLfloat lry)
glColor3f(GLfloat r, GLfloat g, GLfloat b) //Colors in 0 to 1 range
```

### GL Datatypes

C datatypes aren't always the same size on different platforms. The OpenGL designers created platform specific datatypes (somewhere it's typedef char GLbyte)

- GLbyte: 8 bit Integer (GLubyte is unsigned)
- GLshort: 16 bit Integer (GLushort)
- GLint, GLsizei: 32 bit Integer (GLuint, GLenum, GLbitfield)
- GLfloat, GLclampf: 32 bit Floating Point
- GLdouble, GLclampd: 64 bit Floating Point
- GLus are unsigned versions of integers

## GL Function Naming Convention

- The designers of OpenGL choose a function naming convention to reflect it's library, function and arguments
- Convention: <library><command><# args><arg types>
- ex: <gl><Color><3><f>
  - Library is "gl" (vs. glut or glu)
  - Command is "Color"
  - Takes "3" GLfloat ("f") args
- Often we have other functions with the same name, different args. (ex: glColor4f)

## Modifications to the Code

```
void changeSize(GLsizei w, GLsizei h); //New Function Prototype

int main (int argc, const char * argv[]) {
    //...
    glutReshapeFunc(changeSize);
    //...
    void renderer(void) //Ch
    {
        glClear(GL_COLOR_BUFFER_BIT);
        glColor3f(1.0, 0.0, 0.0); //Set the color to red
        glRectf(100.0, 150.0, 150.0, 100.0); //Draw a rectangle from 100,150 to 150, 100
        glFlush();
    }
    void changeSize(GLsizei w, GLsizei h) //New Function for resizing
    {
        if(h==0)
            h = 1;

        glViewport(0,0,w,h);
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();

        if(w <= h)
            glOrtho(0, 250, 0, 250.0 * h/w, 1, -1);
        else
            glOrtho(0,250.0 * w/h, 0, 250, 1, -1);
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
    }
    //Initial Setup is the same as before
}
```

## What does this code mean?

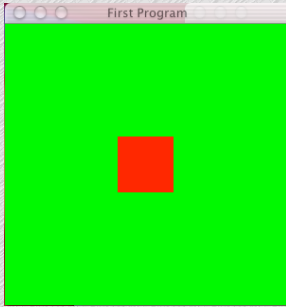
```
void changeSize(GLsizei w, GLsizei h) //New Function for resizing
{
    if(h==0)
        h = 1;

    glViewport(0,0,w,h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    if(w <= h)
        glOrtho(0, 250, 0, 250.0 * h/w, 1, -1);
    else
        glOrtho(0,250.0 * w/h, 0, 250, 1, -1);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}
```

changeSize: Allows us to resize the window. Without it most OpenGL calls will not work on MacOS for some reason. We will discuss some of these calls later.

### Result of this code



### Theory: 3D Transformations

- When we create points or lines in 3D we will designate their location based on their Cartesian coordinates.
- We want to be able to move around in the 3D scene and create 3D objects relative to our present location.
- We also want to be able to move, rotate or scale a subset of all of the objects.
- This is done through linear algebra, multiplying points, represented by vectors, by transformation matrices. (No more math here)

### Time For 3D: Points

- We make everything of vertices. We create vertices with `glVertex3f(x,y,z)`. ( $x,y,z$  are `GLfloat`s)
- All calls to `glVertex` are between calls to `glBegin(what)` and `glEnd()`.
- We save the present state of the transformation matrix by pushing it onto the stack with `glPushMatrix()`
- We restore the old transformation matrix with `glPopMatrix()`

### Code for Drawing a 3D Spiral of Points

```
void render(void){
    GLfloat ang, x, y, z = -50;
    glClear(GL_COLOR_BUFFER_BIT);
    glPushMatrix();
    glRotatef(90,1,0,0);
    glColor3f(1,0,0);
    glBegin(GL_POINTS);
    for(ang = 0; ang < 10 * 3.14; ang += .1) {
        x = 70 * sin(ang);
        y = 70 * cos(ang);
        z += .75;
        glVertex3f(x,y,z);
    }
    glEnd();
    glPopMatrix();
    glFlush();
}

void changeSize(GLsizei w, GLsizei h) {
    GLfloat sideRange = 200;

    glViewport(0,0,w,h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(-sideRange, sideRange, -sideRange, sideRange, sideRange, -sideRange);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}
```

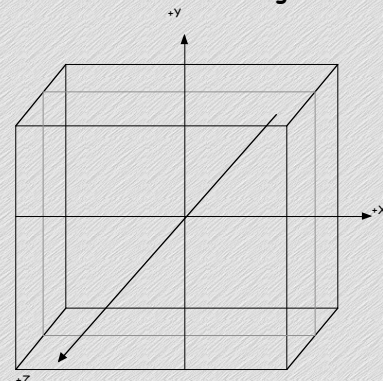
### What does this code mean? The Cartesian Volume

```
void changeSize(GLsizei w, GLsizei h) {
    GLfloat sideRange = 200;

    glViewport(0,0,w,h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(-sideRange, sideRange, -sideRange, sideRange, sideRange, -sideRange);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}
```

- In `changesize` we created a "Cartesian Viewing Volume", a  $200 \times 200 \times 200$  cube in which all subsequent drawing would occur.
- "Think of it as a 3D canvas"
- `glOrtho(left, right, bottom, top, near, far)` creates such a volume.

### The Cartesian Viewing Volume



## What does this code mean? Renderer

```
GLfloat ang, x, y, z = -50;
glClear(GL_COLOR_BUFFER_BIT);
glPushMatrix();
glRotatef(90, 1, 0, 0);
glColor3f(1, 0, 0);
glBegin(GL_POINTS);
for(ang = 0; ang < 10 * 3.14; ang += .1) {
    x = 70 * sin(ang);
    y = 70 * cos(ang);
    z += .75;
    glVertex3f(x, y, z);
}
glEnd();
glPopMatrix();
glFlush();
```

- Pretty much the code from your algebra class for drawing a 3D spiral of radius 70.
- We do a rotate because otherwise we would just see the top. We will revisit rotates later.

## Drawing Lines

Instead of using `glBegin(GL_POINTS)` we use `glBegin(GL_LINES)`

We can draw lines as follows

```
glBegin(GL_LINES);
glVertex3f(x1, y1, z1);
glVertex3f(x2, y2, z2);
glEnd();
```

But that would be very inefficient, we need to make lots of calls to `glBegin` and `glEnd`. So, we adopt a more convenient convention.

## Drawing Lines

```
glBegin(GL_LINES)
glVertex3f(x1, y1, z1);
glVertex3f(x2, y2, z2);
glVertex3f(x3, y3, z3);
glVertex3f(x4, y4, z4);
//etc
glEnd();
```

A Line is drawn from 1 to 2 and a second line drawn from 3 to four.

## Line Strips

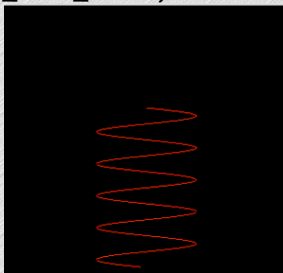
Often we don't want to draw individual lines but want to draw a line strip where each point connects to the next. For example:

```
glBegin(GL_LINE_STRIP);
glVertex3f(x1, y1, z1);
glVertex3f(x2, y2, z2);
glVertex3f(x3, y3, z3);
glVertex3f(x4, y4, z4);
glEnd();
```

Draws a line from 1 to 2, a line from 2 to 3 and a line from 3 to 4.

## Changing the spiral to lines

- We can change our spiral from points to lines simply by changing: `glBegin(GL_POINTS)`;
- to `glBegin(GL_LINE_STRIP)`;



## Doing Animation

- So far we have only drawn still images. Now we will discuss animation.
- We need two buffers. One for the present frame which is being displayed and one for the next frame (which we are working on).
  - change to `glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGB)`;
- Instead of flushing buffers we will swap them.
  - use `glutSwapBuffers()` instead of `glFlush()`
- We need the ability to order GLUT to update the display after a period of time.
  - Use `glutTimerFunc(time_ms, func_ptr, arg)`;

## Simple Explosion Simulator Code Part 1

```
int main (int argc, const char * argv[]) {
    //...
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
    //...
    glutTimerFunc(25, tf, 1);
    glutMainLoop();
    //...
}

void tf(int blah){
    glutPostRedisplay(); //We do not call renderer directly
    glutTimerFunc(25, tf, 1); //Call our self again in 25 ms
}

void renderer(){
    int i;
    glClear(GL_COLOR_BUFFER_BIT);
    glPushMatrix();
    glColor3f(1, 0, 0);
    glBegin(GL_POINTS);
    for(i=0; i < 256; i++){
        glVertex3f(xs[i], ys[i], zs[i]);
        xs[i] += dxs[i];
        ys[i] += dys[i];
        zs[i] += dzs[i];
    }
    glEnd();
    glPopMatrix();
    glutSwapBuffers(); //Swap Buffers, not flush
}
```

## Explosion Demo Part 2

```
void initialSetup(void)
{
    int i;
    glClearColor(0.0, 0.0, 0.0, 1.0);
    srand(SEED);
    for(i=0; i < 256; i++)
    {
        xs[i] = ys[i] = zs[i] = 0;
        dxs[i] = (rand() % MAX_V) / DIV_V - SUB_V;
        dys[i] = (rand() % MAX_V) / DIV_V - SUB_V;
        dzs[i] = (rand() % MAX_V) / DIV_V - SUB_V;
    }
}
```

DEMO

## Drawing Triangles & Winding Part 1

- We draw triangles the same we did lines and points.
  - We use `glBegin(GL_TRIANGLES)`
- Most shapes made out of polygons are closed, so we don't want to render triangles both the inside and outside of triangles in these shapes.
- We use winding to specify the inside and outside of a triangle. The order in which the vertices are entered expresses a front side and a back side.
  - Clockwise winding: The front is the side in which the vertices are in clockwise order.

## Drawing Triangles & Winding Part 2

- We can use either clockwise or counterclockwise winding but we want to be consistent.
  - specify using: `glFrontFace(GL_CW)` [or `GL_CCW`]
- The process of eliminating the back face is called backface culling. By default this is off.
- Winding while not completely necessary is a good thing to keep in mind while writing code.
  - It is hard to fix if you realize you need it after you have drawn a bunch of triangles without following either convention.

## Changes to main, initialSetup

In main: We change `glutInitDisplayMode` to include `GLUT_DEPTH`.

### In Initial Setup

```
glEnable(GL_DEPTH_TEST);
glEnable(GL_CULL_FACE);
glPolygonMode(GL_FRONT, GL_LINE);
```

- `glEnable()` and `glDisable` are used to turn on and off various OpenGL features. Here we turn on depth testing and hidden surface elimination (culling).
- `glPolygonMode(GL_FRONT, GL_LINE)` tells OpenGL to draw the front of the polygons as wire frames only. `GL_FILL` would tell it to draw filled polygons. `GL_BACK` would be used to give different settings for the backs. (NOTE: `glPolygon Mode` doesn't cull, it just tells OpenGL how to draw stuff.

## Rewriting Renderer

```
void renderer() {
    int lastang;
    float ang;

    lastseed = SEED; //For random number generator
    lastang = ANG_ROT; //For rotation
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glPushMatrix();
    glRotatef(lastang, 1, 1, 0);
    glBegin(GL_TRIANGLES);
    for(ang = 0; ang < 2 * MY_PI; ang += ANG_INCR) {
        glColor3f((rand() % 1000) / 1000.0, (rand() % 1000) / 1000.0, (rand() % 1000) / 1000.0);
        glVertex3f(0, 0, 0); //top
        glVertex3f(70 * sin(ang - ANG_INCR), 70 * cos(ang - ANG_INCR), 70); //left
        glVertex3f(70 * sin(ang), 70 * cos(ang), 70); //right
    }

    for(ang = 0; ang < 2 * MY_PI; ang += ANG_INCR) {
        glColor3f(last, last, 0);
        glVertex3f(0, 0, 70); //Middle
        glVertex3f(70 * sin(ang), 70 * cos(ang), 70); //right
        glVertex3f(70 * sin(ang - ANG_INCR), 70 * cos(ang - ANG_INCR), 70); //left
        last = lastang;
    }
    glEnd();
    glPopMatrix();
    glutSwapBuffers();
}
```

## What does this code mean? Renderer

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

Clears both the color buffer and the depth buffer

```
for(ang = 0; ang < 2 * MY_PI; ang += ANG_INCR) {
    glColor3f((rand2() % 1000) / 1000.0, (rand2() % 1000) / 1000.0, (rand2() % 1000) / 1000.0);
    glVertex3f(0, 0, 0); //top
    glVertex3f(70*sin(ang-ANG_INCR), 70*cos(ang-ANG_INCR), 70); //left
    glVertex3f(70*sin(ang), 70*cos(ang), 70); //right
}
```

Sets a random color, draws a vertex for the top of a cone and vertices for the bottom.

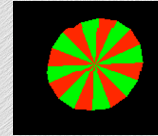


## What does this code mean? Renderer

```
for(ang = 0; ang < 2*MY_PI; ang+= ANG_INCR) {
    glColor3f(last, !last, 0);
    glVertex3f(0, 0, 70); //Middle
    glVertex3f(70*sin(ang), 70*cos(ang), 70); //right
    glVertex3f(70*sin(ang-ANG_INCR), 70*cos(ang-ANG_INCR), 70); //left
    last = !last;
}
```

Sets the color of the bottom to alternate between green and red.

Draws the middle of the bottom, and the two triangles on the edges.



## Some Nuisances

- We cannot put state command between `glBegin()` and `glEnd()`
  - This means we have to call `glEnd` to change states
- `glPolygonMode()` changes the way polygon front or backs are drawn but not if they are drawn
- We cannot forget to add `GLUT_DEPTH` in `glutInitDisplayMode()`
- We cannot forget to clear the depth buffer using `glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)`

## Aside Pseudorandom Numbers in OS X

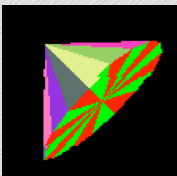
- In OS X, the built-in C function `rand()` is broken
- It's supposed to work like this:
  - $\text{rand\_val} = (\text{previous\_val} * 16807) \% (2^{31}-1)$
- My guess is that Apple Implemented it something like this

```
unsigned int glb_seed;
unsigned int rand2()
{
    //Below Causes overflow in the multiply! Implicit mod by 2^32.
    glb_seed = (glb_seed * 16807) % 2147483647;
    return glb_seed;
}
```

- We can fix the whole problem by using changing `glb_seed`'s type to unsigned long long
- So I implemented my own version for the examples

## DEMO

### • DEMO



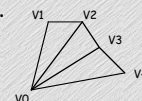
Output from Builtin `rand` and my `rand2`

```
Four random numbers between 0 & 4 with rand: 3, 0, 1, 2
Four random numbers between 0 & 4 with rand: 3, 0, 1, 2
Four random numbers between 0 & 4 with rand: 3, 0, 1, 2
Four random numbers between 0 & 4 with rand: 3, 0, 1, 2
Four random numbers between 0 & 4 with rand: 3, 0, 1, 2
Four random numbers between 0 & 4 with rand: 3, 0, 1, 2
Four random numbers between 0 & 4 with rand2: 3, 3, 2, 2
Four random numbers between 0 & 4 with rand2: 0, 1, 2, 3
Four random numbers between 0 & 4 with rand2: 3, 0, 0, 0
Four random numbers between 0 & 4 with rand2: 1, 2, 0, 0
Four random numbers between 0 & 4 with rand2: 1, 3, 1, 2
Four random numbers between 0 & 4 with rand2: 2, 2, 0, 2
```

## Triangle Fans

- We entered lots of vertices multiple times in the previous file, that's bad.
- For example this entered the top vertex every time:
 

```
for(ang = 0; ang < 2 * MY_PI; ang += ANG_INCR) {
    glColor3f((rand2() % 1000) / 1000.0, (rand2() % 1000) / 1000.0, (rand2() % 1000) / 1000.0);
    glVertex3f(0, 0, 0); //top
    glVertex3f(70*sin(ang-ANG_INCR), 70*cos(ang-ANG_INCR), 70); //left
    glVertex3f(70*sin(ang), 70*cos(ang), 70); //right
}
```
- We don't want to do this, it slows down the program and requires more code. We would like to make fans instead.



## New code using Triangle Fans

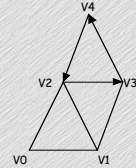
```
glFrontFace(GL_CCW);
glBegin(GL_TRIANGLE_FAN);
glVertex3f(0,0,0); //top
for(ang = 0; ang < 2 * MY_PI + ANG_INCR; ang += ANG_INCR) {
    glColor3f((rand2C) % 1000) / 1000.0, (rand2C) % 1000) / 1000.0, (rand2C) % 1000) / 1000.0;
    glVertex3f(70*sin(ang), 70*cos(ang), 70); //right
}
glEnd();

glFrontFace(GL_CCW);
glBegin(GL_TRIANGLE_FAN);
glVertex3f(0,0,70); //Middle
for(ang = 0; ang < 2*MY_PI+ANG_INCR; ang+= ANG_INCR) {
    glColor3f(last, 1last, 0);
    glVertex3f(70*sin(ang), 70*cos(ang), 70); //right
    last = 1last;
}
}
```

We now only enter one vertex per iteration of the for loops instead of the previous three. (Code is in Demo 5)

## Triangle Strips

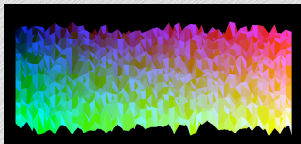
- Often we have code where we want to form a surface by placing many triangles side by side.
- It would be very redundant to draw all three vertices every time. We only need one additional point for each triangle.
- We need to draw in an order which preserves winding.
- Each vertex  $V_n$  connects to  $V_{n-1}$  and  $V_{n-2}$ .



## Demo

```
//In Renderer
for(i=0; i < 49; i++) {
    glBegin(GL_TRIANGLE_STRIP);
    for(j=0; j < 50; j++) {
        glColor3f(i/50.0, j/50.0, heightfield[i][j] / 10.0);
        glVertex3f(i, j, heightfield[i][j]);
        glVertex3f(i+1, j, heightfield[i+1][j]);
    }
    glEnd();
}

//In Initial Setup
//Generate Height Field
for(i=0; i < 50; i++)
    for(j=0; j < 50; j++)
        heightfield[i][j] = rand2C) % 100 / 10.0;
```



## Transformation: Moving, Rotating and Scaling

- In several of my most recent demos, I have demonstrated that stuff is actually 3D by spinning (rotating) it. In addition to rotating, we can engage in scaling, and moving (translating). This section deals with how to do this.
- We often want to move, scale or rotate a specific object or draw one thing from a specific point.
- We then would like to back up to where we were before. We use a matrix stack to assist us.
- We use `glPushMatrix()` to save current state
- We use `glPopMatrix()` to restore the last saved state (warning: there is a maximum depth)

## Definition of Terms

- Eye coordinates are the coordinates where the observer watching the scene on the screen. These can be thought of as absolute coordinates.
- Viewing Transformation: Determines the point of observation and the direction in which the observer is looking.
  - Think of it as placing and pointing the camera
  - Default: (0,0,0) looking down -z axis
- Modeling Transformations: Used to manipulate objects through placing (translating), rotating and scaling
  - Order matters of transformations matters
- Model and viewing matrices are actually the same (ex: move camera forward vs. moving objects all backward), these are called Modelview transformations
- Perspective Transformations: "How finished scene is translated into an image on the screen."
  - Defines clipping planes, viewing volume
  - Orthographic or Perspective

## Translations

- We have a function that comes with glut called `glutWireCube(size)`.
- We can think of ourselves as moving around a 3D canvas, drawing stuff relative to where we are.
- `glutWireCube(size)` always draws at the origin
- In order to draw the cube where we are, we need to translate to the place we want to place we want to draw it and then call `glutWireCube()`
- `glTranslate3f(GLfloat x, GLfloat y, GLfloat z)`



## Translations Example

```
void render()
{
    int last=0;
    float ang;

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

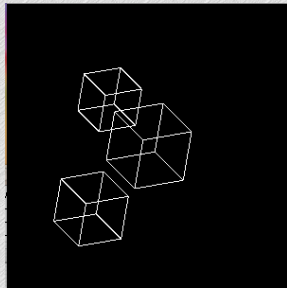
    glRotatef(45,1,1,0);
    glPushMatrix();

    //Draw a Cube Centered at origin
    glutWireCube(80);

    //Draw a Cube Centered at -80, -90, 0
    glPushMatrix();
    glTranslatef(-80, -90, 0);
    glutWireCube(70);
    glPopMatrix();

    //Draw a Cube Centered at -80, 90, 0
    glPushMatrix();
    glTranslatef(-80, 90, 0);
    glutWireCube(60);
    glPopMatrix();

    glPopMatrix();
    glutSwapBuffers();
}
```



## Rotation

Rotation has several uses

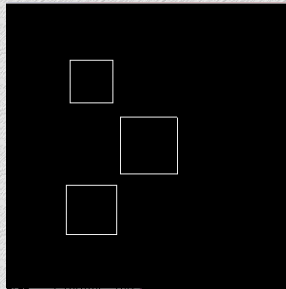
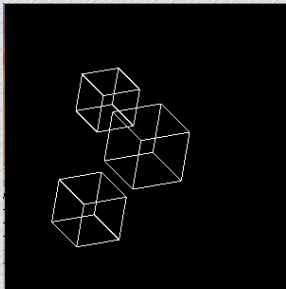
- We can make the entire scene rotate to demonstrate that an object is actually 3D.
- We could be drawing a smaller piece of scene and want that to rotate
  - For instance, we might want a door to swing open or a character's head to turn toward us
- `glRotatef(GLfloat angle_degrees, GLfloat x, GLfloat y, GLfloat z)`
  - Rotates all objects by `angle_degrees` about the given axis

## Rotations Example

In the three cubes example, we used `glRotatef(45,1,1,0)`;

With Rotation

Without Rotation



## Scales

- We can scale objects uniformly or non-uniformly (differently in different directions).
- We might want to scale things for a variety of reasons
  - For example: We made our model too big or too small.
- `glScalef(GLfloat x, GLfloat y, GLfloat z)`
  - Scales by `x`, `y` and `z` in their respective directions

## Scales Example

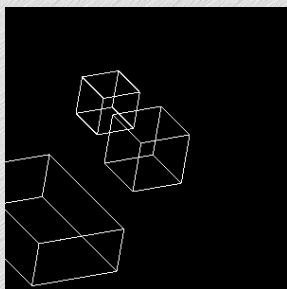
We slightly modify the Cubes code

```
glRotatef(45,1,1,0);
glPushMatrix();
glutWireCube(80);

glPushMatrix();
glScalef(2, 1, 3); //Double x, triple depth
glTranslatef(-80, -90, 0);
glutWireCube(70);
glPopMatrix();

glPushMatrix();
glTranslatef(-80, 90, 0);
glutWireCube(60);
glPopMatrix();

glPopMatrix();
```



## How we Made Stuff Spin

In Previous Examples We

- Used a timer function that had a 66.6 FPS frame rate
- We used a global float called "lastang" to keep track of the previous angle
- Each time we refreshed the frame, we incremented "lastang" by a tiny bit, causing a slightly larger rotation

```
lastang += ANG_ROT;
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

glPushMatrix();
glRotatef(lastang,1,1,0);
```

## glLoadIdentity()

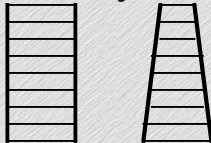
- glLoadIdentity() simply restores everything to the "normal" coordinate system. All transformations: rotations, scales and translates are undone. This resets the transformation (the thing that does all the scales, rotates and translates) to the identity matrix.
  - It does not effect the stack in anyway. Popping a matrix off the stack will overwrite the identity with whatever was on the stack.

## Two Modes, Two Kinds of Transforms

- There are actually two different matrices used in OpenGL
- There is a `GL_MODELVIEW`, the matrix we use and manipulate to draw pieces of our scene
    - This is what we have used so far in these examples
    - "Used to place and orient objects" and "move objects around"
  - `GL_PROJECTION`, the
    - "Used to define clipping volume"
    - All Points are multiplied by this matrix
    - Allows for perspective projections, making stuff in the distance appear smaller
    - We can also manipulate this matrix to apply a scene wide transformation
    - Calls to `GLOrtho` and `GLFrustum` set up this matrix

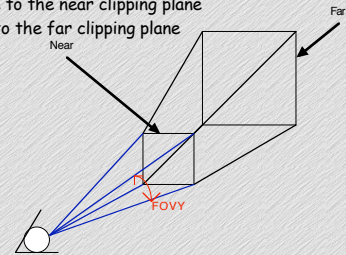
## Perspective vs. Orthographic Projection

- In orthographic projection, all polygons are drawn at the same size and shape regardless of distance from the eye.
  - Used in CAD (ex: drawing the plans for assembling products)
  - Can also be used when the relative distances of items are small compared to total distance to eye
- Perspective makes object that are far away look smaller (foreshortening).
  - Parallel lines converge at a "vanishing point" as their distance from the viewer increases.
  - Used in most realistic looking scenes



## Setting up Perspective

- Hard way: Using `glFrustum`
- Easier way: using `gluPerspective`
  - `gluPerspective(GLdouble fovy, GLdouble aspect, GLdouble zNear, GLdouble zFar)`
  - `fovy`: Field of view in y (vertical) direction
  - `aspect`: Width/Height
  - `zNear`: Distance to the near clipping plane
  - `zFar`: Distance to the far clipping plane



## Code for Perspective

Replace the call to `glOrtho` with

```
gluPerspective(30, (GLfloat) w/(GLfloat) h, 1, 400);
```

Example drawing code

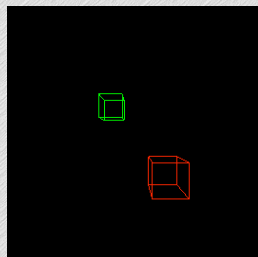
```
glTranslatef(0, 0, -200);
glRotatef(30, 1, 0, 0);
glPushMatrix();
glColor3f(1, 0, 0);

glPushMatrix();
glTranslatef(40, 40, 50);
glutWireCube(40);
glPopMatrix();

glColor3f(0, 1, 0);

glPushMatrix();
glTranslatef(-40, -40, -50);
glutWireCube(40);
glPopMatrix();

glPopMatrix();
```



## The End

In summary we have done:

- A Little History
- Setup of OpenGL and Some Conventions
- Drawing Points, Lines and Triangles
- Animation
- Culling and Winding
- Triangle Strip and Fans
- Scene Graphs: Translation, Rotation and Scaling
- Orthographic vs. Perspective

When I have some free time, I'll try to post some slides on the web about doing shading and textures.