

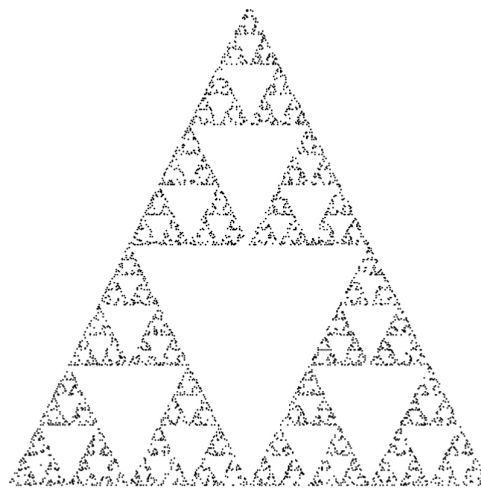
CS174A : Introduction to Computer Graphics

Royce 190
TT 4-6pm

Scott Friedman, Ph.D
UCLA Institute for Digital Research and Education

Let's get something on the screen

- Make that *thing* the example from Ch.2
 - Sierpinski Gasket
 - Fractal Geometry
 - More interesting than a triangle or square



Immediate / Retained Mode

- This example helps you see the difference between the two “modes”.
- They are really programming styles.
- These styles are *performance* related.
- The output is the same no matter which you use.

Immediate / Retained Mode

- Immediate Mode
 - Points generated every time.
 - No storage required.
 - Transfer to GPU every time.

```
main( )
{
    initialize_the_system( );
    p = find_initial_point( );
    for ( some_number_of_points )
    {
        q = generate_a_point( p );
        display_the_point( q );
        p = q;
    }
    cleanup( );
}
```

Immediate / Retained Mode

- Retained Mode #1
 - Points only generated *once*.
 - Storage required for all points.
 - Transfer to GPU every time.

```
main( )
{
    initialize_the_system( );
    p = find_initial_point( );
    for ( some_number_of_points )
    {
        q = generate_a_point( p );
        store_the_point( q );
        p = q;
    }
    display_all_points( );
    cleanup( );
}
```

Immediate / Retained Mode

- Retained Mode #2
 - Points only generated *once*.
 - Storage required for all points *once**
 - Transfer to GPU *once*.

```
main( )
{
    initialize_the_system( );
    p = find_initial_point( );
    for ( some_number_of_points )
    {
        q = generate_a_point( p );
        store_the_point( q );
        p = q;
    }
    send_all_points_to_GPU( );
    cleanup( );
    display_all_points_on_GPU( );
}
```

* for static data

Immediate / Retained Mode

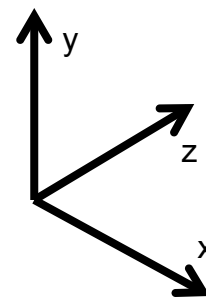
- Immediate Mode
 - Most work, but no storage.
- Retained Mode #1
 - Less work (after first time), but need storage and transfer to GPU every time.
- Retained Mode #2
 - Same work as #1, temporary storage on host, storage on GPU, single transfer time.

Baby Steps in 2D

- Sierpinski Gasket is a 2D object.
 - Lame, not what I signed up for! ☹
- Relax, 2D will be a restricted version of 3D for our exercise.
 - Specifically we will restrict ourselves to the x/y plane.

$$z = 0, p = (x, y, 0)$$

$$p = \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} x \\ y \\ 0 \end{bmatrix}$$



Points and vertices

- A point p is a thing in and of itself.
- A vertex v is understood to be part of the description of something more complex.
 - A point is a point.
 - A vertex defines part of a line, triangle, some other OpenGL primitive.

Sierpinski Gasket Code

- Code
 - Nothing fancy but will get us something to render on our display.

```
#include "vec.h"
typedef vec2 point2;
main( )
{
    const int NumPoints = 5000;
    // define a triangle in plane z=0
    point2 vertices[3]={point2(-1.0,-1.0),point2(0.0,1.0),point2(1.0,-1.0);
    // arbitrary point within triangle
    points[0] = point2(0.25,0.5);
    // compute and store NumPoints-1 new points
    for ( int k=1; k<NumPoints; k++ )
    {
        int j = rand( ) % 3; // pick a vertex at random
        // compute the point halfway between selected vertex and previous point
        points[k] = ( points[k-1] + vertices[j] ) / 2.0;
    }
}
```

Great, now what?

- Things we need to know (decide)
 - What color to use for rendering?
 - Where on display should rendering appear?
 - How large will rendering be?
 - How to we create a window to render into?
 - How much of the rendering plane will appear?
 - How long will the image appear on the display?

Use OpenGL API

- OpenGL provides various functionality
- Broken into various categories
 - Primitive functions
 - Attribute functions
 - Viewing functions
 - Transformation functions
 - Input functions
 - Control functions
 - Query functions

Primitive Functions

- Represent low level object we can draw.
- They are the “what” of the API
 - Points
 - Lines
 - Triangles
- Everything else made up from these.

Attribute Functions

- This is the “how” of the API
 - Color
 - Lines style
 - Point size
- Attributes can apply to different contexts
 - Globally
 - Per vertex
 - Per pixel (fragment)

Viewing Functions

- Camera model (from last time)
 - What can be seen (rendered) and how.
 - Position of eye and size of window
- There actually are not any functions specifically dedicated to the camera!
- We “define” the camera through a set of transformations.

Transformation Functions

- OpenGL it sometimes seems is mostly made up of these transformations.
- Transformation functions for
 - Translation, scaling, rotation of geometry.
 - Viewing uses transformation heavily.
- Transformations occur in the application itself or within a Shader.

Input Functions

- Required for any type of interaction.
- There are almost too many input devices to mention.
- We will stick to the basics
 - The trusty keyboard and mouse
- Feel free to explore others for your term project.

Control Functions

- Catch-all for handing the OS and GPU.
- Interfacing with the windowing system.
 - This is required for almost any graphics program today.
 - Can be very complex in and of itself.
 - We will use portable library to abstract things
 - HTML, CSS
 - GLUT: GL Utility Toolkit
 - Available for all three platforms.
 - We will return to this topic very shortly.

Query Functions

- Used to interrogate OpenGL state.
- Also can be used to find out about
 - Underlying host system (e.g. # cpus, memory)
 - Available GPUs (e.g. capabilities, driver rev.)
- WebGL which is based on OpenGL ES
 - Restricted version of the full OpenGL spec.

OpenGL is a State Machine

- We spoke briefly about this last time.
 - As a fixed function pipeline
- A kind of Black Box
- Accepts two types of input
 - Something that alters the internal state
 - Changes or returns state
 - Something that causes some output
 - Specify primitives that flow through the pipeline

OpenGL is a State Machine

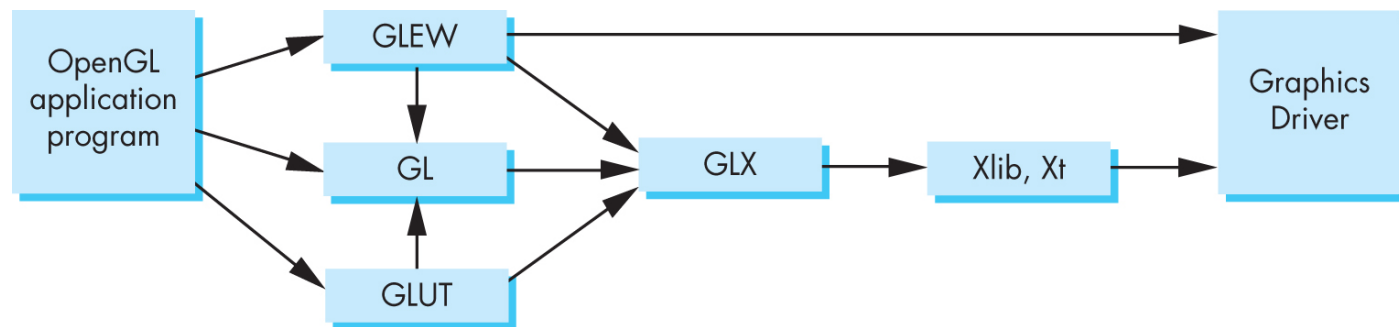
- State manipulation is primarily about
 - Enabling or disabling features
 - Parameter setting
- Older versions of OpenGL (<3.1)
 - Many many internal state variables
- Recent versions of OpenGL (≥ 3.1)
 - You define what you need and use via Shaders that you also define (program).
 - More work – but more power

OpenGL is a State Machine

- State is persistent.
 - Stays set until you change it.
 - Set color to red, it stays red until you change it.
 - This is the cause of a lot of heartache using OpenGL
- State is also not bound to primitives
 - This makes sense given the statement above
 - But we often want to think of things this way
 - Cube is yellow, grass is green, sun is yellow, etc.
 - You have to manage this yourself
 - Data structures to the rescue!!

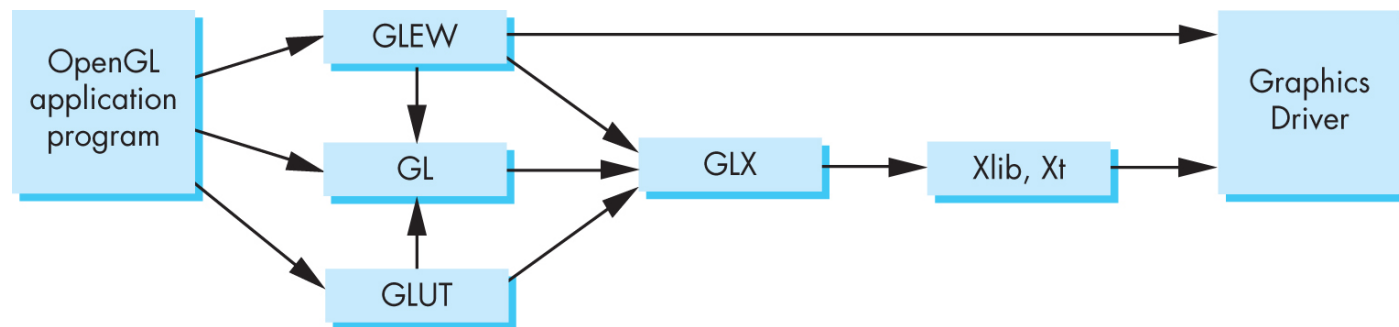
OpenGL is a Library

- Just like any other library you might use.
 - Usually libGL.{lib,so,ksym,dll,etc...}
 - All functions begin with “gl”
 - Shaders are written in a C-like language, GLSL



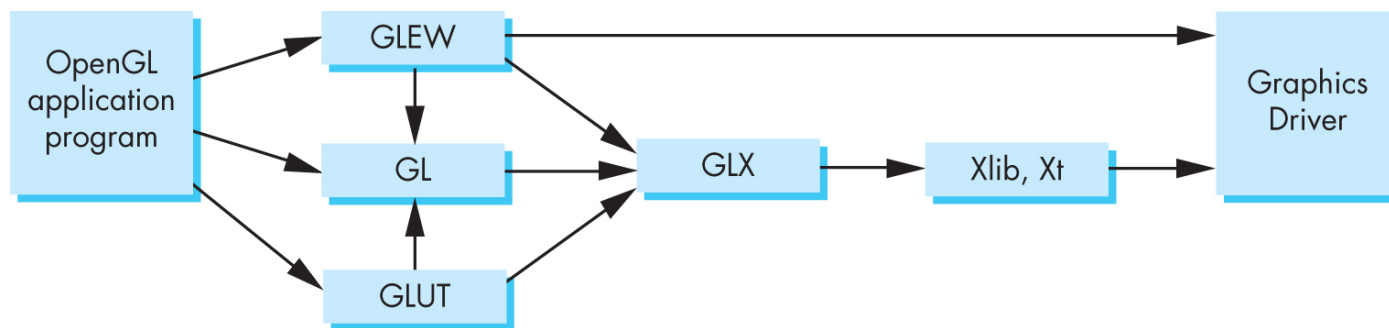
OpenGL is a Library

- Uses a “glue” library to interface with the hosts’ underlying window system.
 - For linux it is called GLX
 - OSX it used to be called agl
 - Windows it is called wgl
 - Web, HTML



OpenGL and portability

- GLUT
 - Simple system independent windowing and input processing.
- GLEW
 - Removes OS specific dependencies w.r.t. OpenGL extensions.



OpenGL and portability

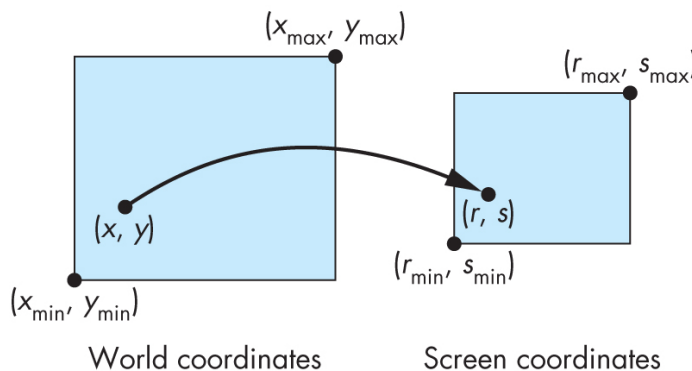
- OpenGL defines many types
 - Almost always as a `typedef`
 - Hides differences between systems representation of things like `int` and `float`
 - The type `GLfloat` will always have the same physical representation.
- Function names follow a pattern (`nt`, `ntv`)
 - `n`=(1,2,3,4,matrix) and specifies dimensions
 - `t`=(I,f,d) and specifies data type
 - `v`=pointer to an array of specified type.

Coordinate systems

- Device independent graphics
 - We do not specify pixels on the display.
 - There are no units in OpenGL
- World coordinate system
 - This is where our geometry is defined
 - Represented with floats
- Window coordinate system
 - This is the physical display (or window)
 - It has a width and height in pixels

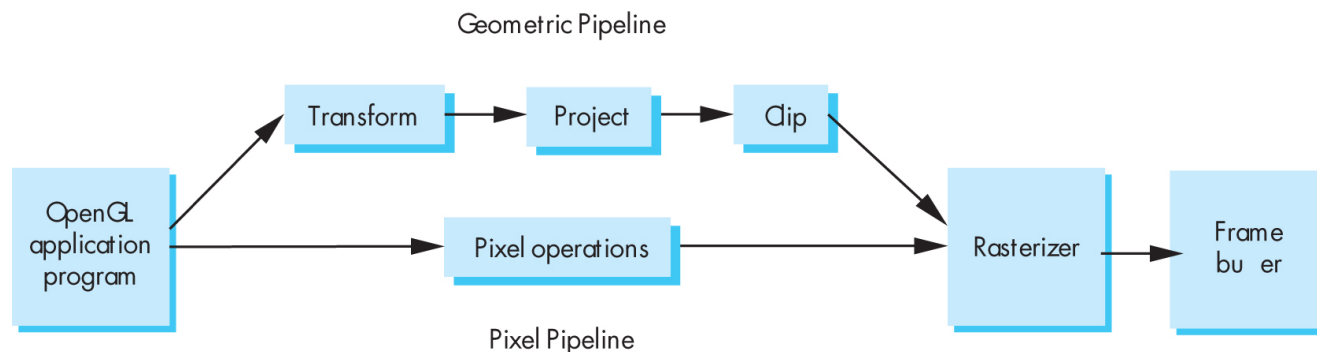
Coordinate systems

- They can be (are) others...but for now
- Transformations FTW!
 - Gets us from the world coordinate system to the window (screen) coordinate system.
 - Gets *what* exactly from world to window?



Primitives of course!

- Primitives go down the OpenGL pipeline and onto our display.
 - Two types: geometry and raster primitives
 - Geometry, in world space, has to be transformed, projected and clipped.
 - Raster data is already in screen space (later)



Primitives of course!

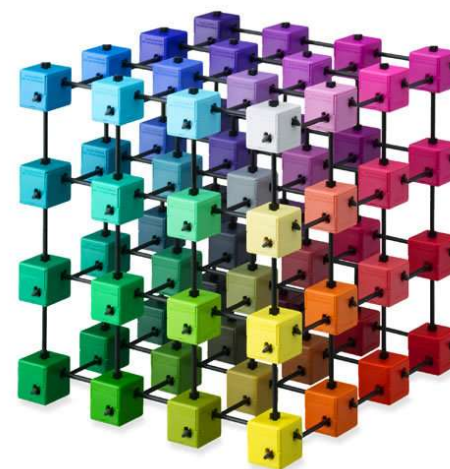
- Recall the types available
 - Points, lines and triangles + strip variations
 - Hardware can render these *very quickly*
 - More complex geometry made from these
 - Specifically, approximated using these types
 - Curves, surfaces and four or more sided polygons
 - All primitives are made up of vertices
 - Attributes assigned define how vertices are processed by GPU

Primitives of course!

- Given an OpenGL primitive type
 - `GL_POINT`, `GL_LINE`, `GL_TRIANGLE`, etc.
- A primitive can be drawn with command
 - `glDrawArrays(GL_POINT, 0, numPoints)`
 - OMG – first line of OpenGL at slide 31!
- Actually quite a bit required before this will work, unfortunately.

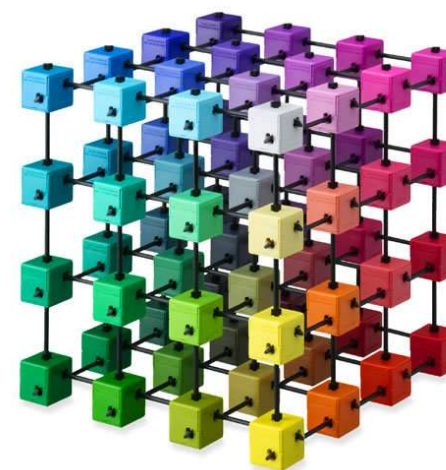
For starters...

- Need at least one attribute, like color.
- CG uses additive color
 - Add color to black (nothing)
- Finger painting is subtractive (and messy)
 - Subtract from white (absorb)



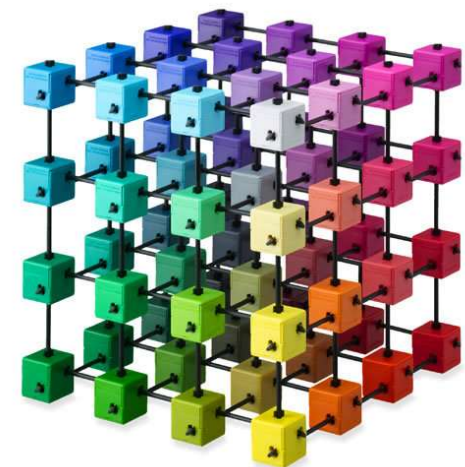
Super Basic Color

- The three primaries are
 - Red, Green and Blue – lets make a cube!
 - Intensities along each axis give all colors
 - Including white and black
 - White=all colors max intensity
 - black=seriously?
 - Hardware stores as bits
 - Typically 8-bits for each color
 - Plus one more for “alpha”
 - You will see RGB and RGBA
 - We will revisit the A later on



Super Basic Color

- OpenGL doesn't know how many 'bits'
- It does not want to...
 - So it abstracts the color to floats
 - RGB and A are represented by a range from (0.0-1.0)
 - White=(1.0, 1.0, 1.0, 1.0)
 - Black=(0.0, 0.0, 0.0, 1.0)
 - Wait, what's that last 1.0?
 - A = Alpha!
 - 0.0=transparent, 1.0=opaque

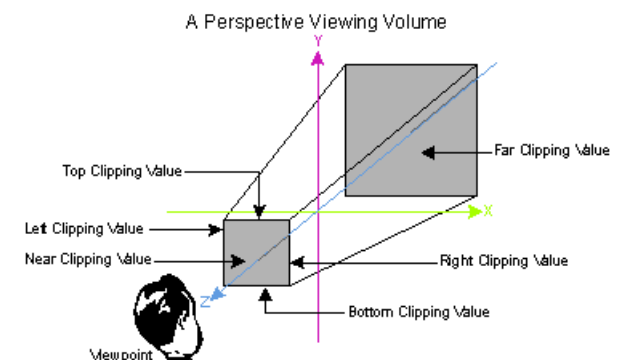
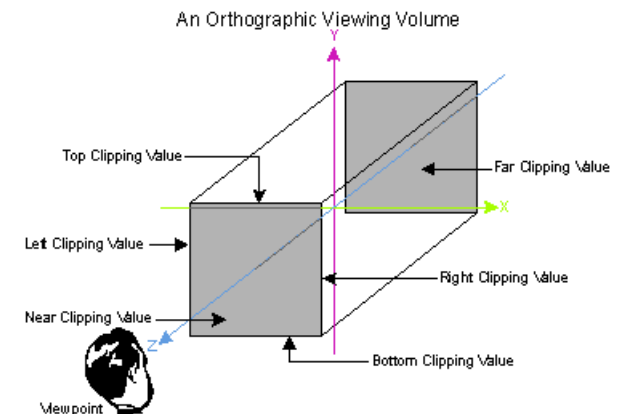
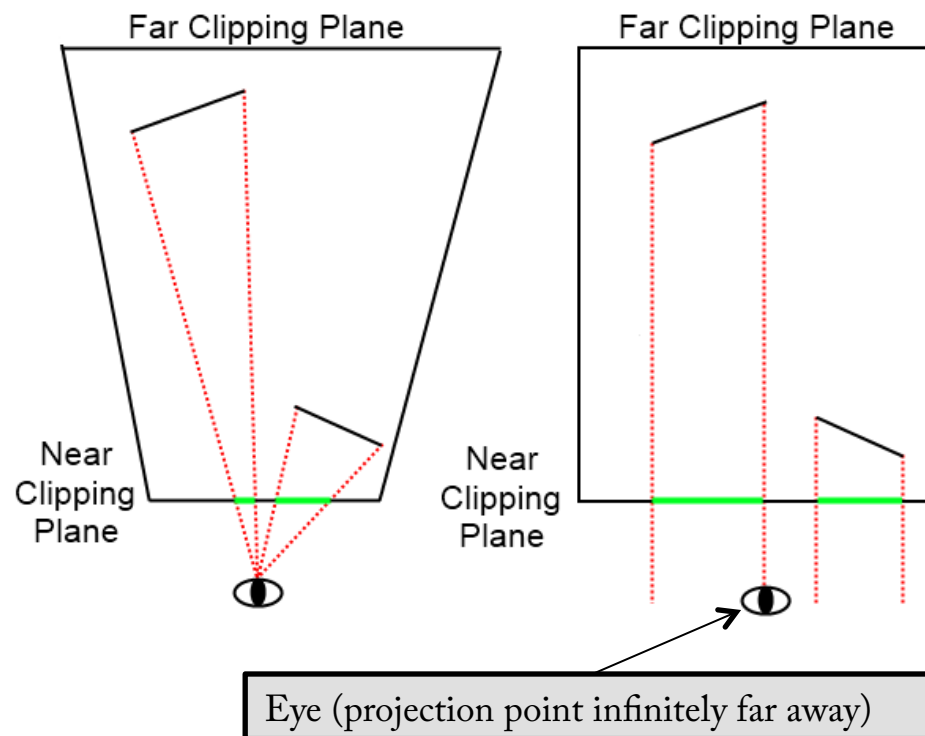


Getting there - Viewing

- Two types, generally, good for viewing
 - Perspective
 - What we talked about last time
 - Single projection point
 - Appearance of depth (like our own vision)
 - Orthographic
 - Move projection point infinitely far away
 - Projection lines parallel to image plane
 - As if projection point is always coincident with image plane

Getting there - Viewing

- Two views to illustrate the difference.

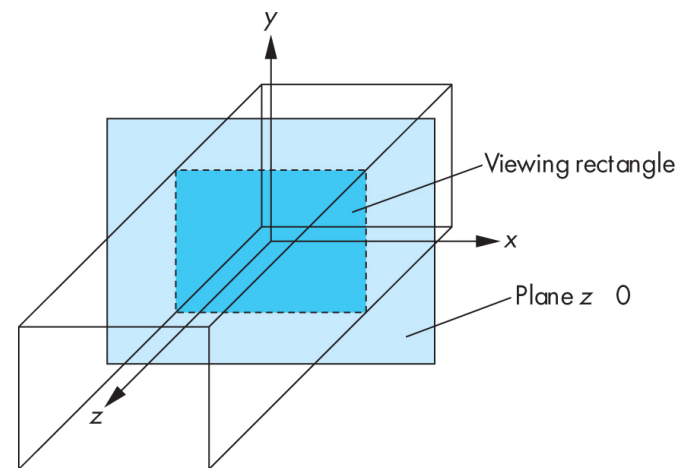
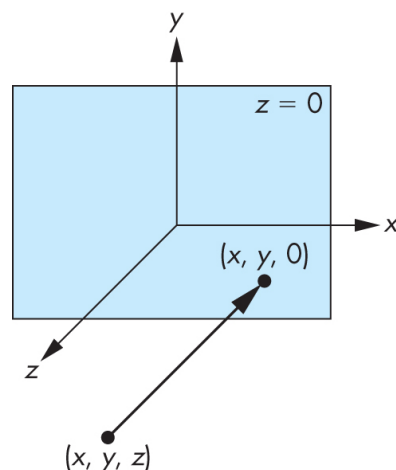


Getting there - Viewing

- What's orthographic projection good for in a 3D environment?
 - 2D data
 - Text “on the glass”, e.g. high-score
 - Head-up Displays (HUD, think cockpit display)

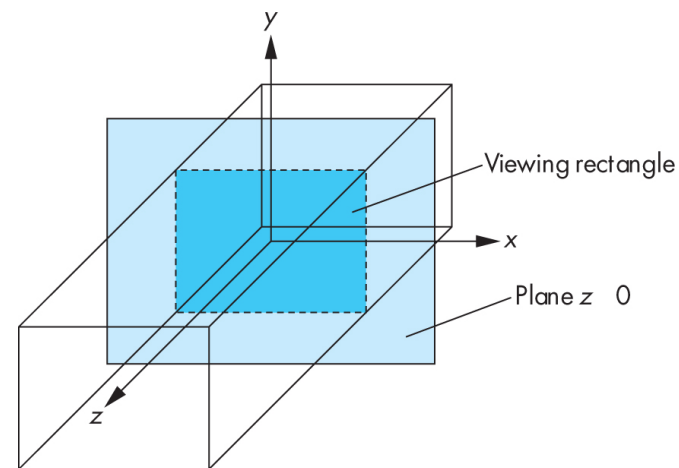
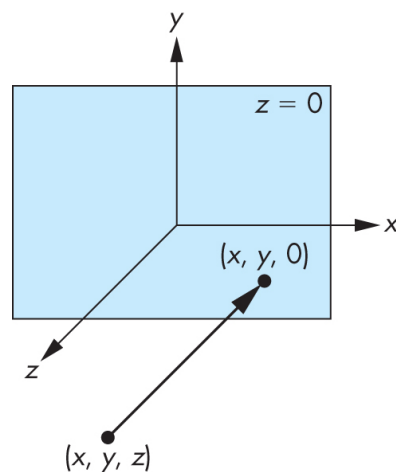
Getting there - Viewing

- 2D image plane will be defined by $z=0$
 - OpenGL defines a canonical view volume and projection point. *a.k.a. the default.*
 - We can use it without any transformations.
 - That default “looks” down the $-z$ axis.



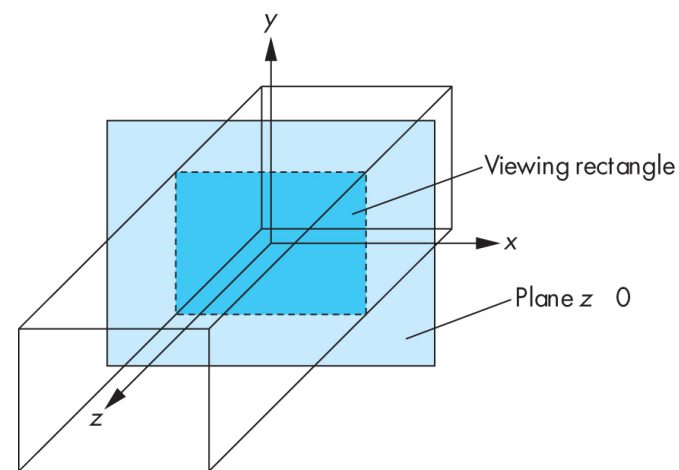
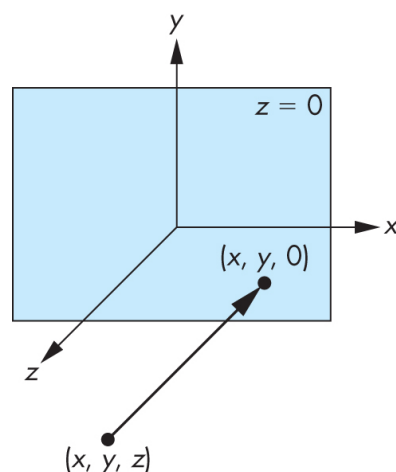
Getting there - Viewing

- The canonical view volume
 - Defines what is projected onto the window and what is clipped.
 - Defined by default as the cube, $(-1, -1, -1)$ to $(1, 1, 1)$



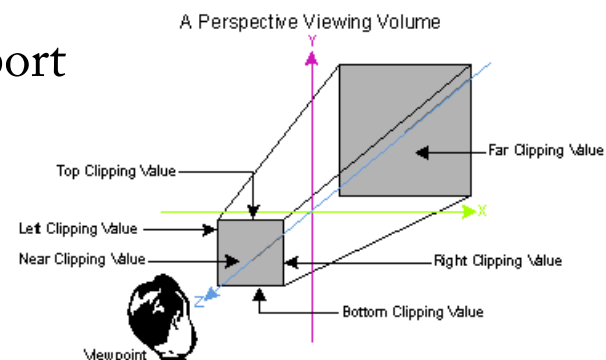
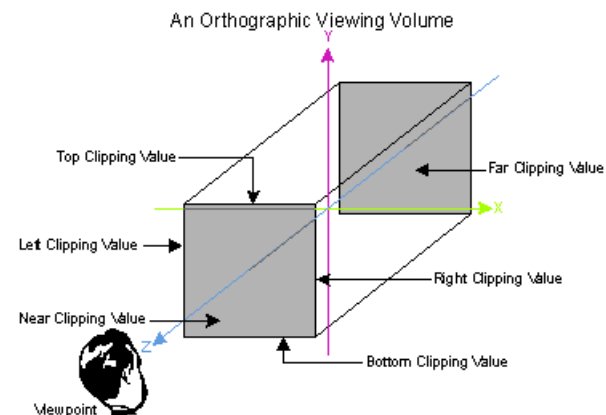
Getting there - Viewing

- Normally, a transformation would need to be defined to convert our desired projection view volume into the canonical view volume.



Getting there - Viewing

- Why do we even have a volume?
 - You said this is 2D!
 - Yes, but internally it's all 3D
 - So we need
 - Near clip
 - Far clip
 - As well as the “sides”
which is what we call the viewport



Getting there - Viewing

- In any case...
 - Using the canonical view volume, for now, lets us avoid explaining transformation matrices for now.
 - As long as we use vertex coordinates that are within the range of $(-1,-1)$ and $(1,1)$ we will stay within the bounds of the view volume and avoid having anything clipped.
- Which is a Good ThingTM, for now

Boilerplate

- Let's write some code already...
- We are going to use GLUT
 - So let's see what is at the core of every GLUT based program so we can put this all together.
 - We will distribute basic GLUT code online that will help you get the linking right for your platform. (we hope)

Boilerplate GLUT

- Here is the basic code
 - Include the header which defines GLUT
 - A little different on OSX
 - It includes `#include <GL/gl.h>`

```
#include <GL/glut.h>
int main( int argc, char **argv )
{
    glutInit( &argc, argv );
    glutInitDisplayMode( GLUT_SINGLE, GLUT_RGB );
    glutInitWindowSize( 500, 500 );
    glutInitWindowPosition( 0, 0 );
    glutCreateWindow( "my window" );
    glutDisplayFunc( display );
    init( );
    glutMainLoop( );
}
```

Boilerplate GLUT

- Let's step through each line
 - Call `glutInit()`
 - Pass command line parameters through.
 - Should be called very near the start of your code.

```
#include <GL/glut.h>
int main( int argc, char **argv )
{
    glutInit( &argc, argv );
    glutInitDisplayMode( GLUT_SINGLE, GLUT_RGB );
    glutInitWindowSize( 500, 500 );
    glutInitWindowPosition( 0, 0 );
    glutCreateWindow( "my window" );
    glutDisplayFunc( display );
    init( );
    glutMainLoop( );
}
```

Boilerplate GLUT

- Let's step through each line
 - Call `glutInitDisplayMode()`
 - Sets single or double buffered mode.
 - Sets type of frame buffer we want.

```
#include <GL/glut.h>
int main( int argc, char **argv )
{
    glutInit( &argc, argv );
    glutInitDisplayMode( GLUT_SINGLE, GLUT_RGB );
    glutInitWindowSize( 500, 500 );
    glutInitWindowPosition( 0, 0 );
    glutCreateWindow( "my window" );
    glutDisplayFunc( display );
    init( );
    glutMainLoop( );
}
```

Boilerplate GLUT

- Let's step through each line
 - Call `glutInitWindowSize()`
 - Sets how big, in pixels, our window will be, width and height.

```
#include <GL/glut.h>
int main( int argc, char **argv )
{
    glutInit( &argc, argv );
    glutInitDisplayMode( GLUT_SINGLE, GLUT_RGB );
    glutInitWindowSize( 500, 500 );
    glutInitWindowPosition( 0, 0 );
    glutCreateWindow( "my window" );
    glutDisplayFunc( display );
    init( );
    glutMainLoop( );
}
```

Boilerplate GLUT

- Let's step through each line
 - Call `glutInitWindowPosition()`
 - Sets the location (windowing system dependent) where our window will display on the desktop.
 - Usually bad form to use this...

```
#include <GL/glut.h>
int main( int argc, char **argv )
{
    glutInit( &argc, argv );
    glutInitDisplayMode( GLUT_SINGLE, GLUT_RGB );
    glutInitWindowSize( 500, 500 );
    glutInitWindowPosition( 0, 0 );
    glutCreateWindow( "my window" );
    glutDisplayFunc( display );
    init( );
    glutMainLoop( );
}
```


Boilerplate GLUT

- Let's step through each line
 - Call `glutCreateWindow()`
 - Parameter is goes in the title bar of the window
 - Window does not actually show up – yet
 - Internal setup goes here (particularly for x-windows)

```
#include <GL/glut.h>
int main( int argc, char **argv )
{
    glutInit( &argc, argv );
    glutInitDisplayMode( GLUT_SINGLE, GLUT_RGB );
    glutInitWindowSize( 500, 500 );
    glutInitWindowPosition( 0, 0 );
    glutCreateWindow( "my window" );
    glutDisplayFunc( display );
    init( );
    glutMainLoop( );
}
```

Boilerplate GLUT

- Let's step through each line
 - Call `glutDisplayFunc()`
 - Set callback for when GLUT wants us to render into the window.
 - We'll return to this in a second.

```
#include <GL/glut.h>
int main( int argc, char **argv )
{
    glutInit( &argc, argv );
    glutInitDisplayMode( GLUT_SINGLE, GLUT_RGB );
    glutInitWindowSize( 500, 500 );
    glutInitWindowPosition( 0, 0 );
    glutCreateWindow( "my window" );
    glutDisplayFunc( display );
    init( );
    glutMainLoop( );
}
```

Boilerplate GLUT

- Let's step through each line
 - Call `init()`
 - Not strictly required, just good practice.
 - Perform any application specific OpenGL setup
 - Name does not matter – we'll come back to this too

```
#include <GL/glut.h>
int main( int argc, char **argv )
{
    glutInit( &argc, argv );
    glutInitDisplayMode( GLUT_SINGLE, GLUT_RGB );
    glutInitWindowSize( 500, 500 );
    glutInitWindowPosition( 0, 0 );
    glutCreateWindow( "my window" );
    glutDisplayFunc( display );
    init();
    glutMainLoop( );
}
```

Boilerplate GLUT

- Let's step through each line
 - Call `glutMainLoop()`
 - This is the “event loop”
 - GLUT *never* returns from this function
 - Your code will run via event callbacks from now on.

```
#include <GL/glut.h>
int main( int argc, char **argv )
{
    glutInit( &argc, argv );
    glutInitDisplayMode( GLUT_SINGLE, GLUT_RGB );
    glutInitWindowSize( 500, 500 );
    glutInitWindowPosition( 0, 0 );
    glutCreateWindow( "my window" );
    glutDisplayFunc( display );
    init( );
    glutMainLoop( );
}
```

Boilerplate GLUT

- The `display()` function
 - Every time GLUT decides that the contents of the window need to be refreshed it will call the function assigned to the display callback.
 - The function does not take or return any parameters.
 - There are ways to cleanly pass data to the callback.
 - i.e. without using global variables. How?

```
void display( void )
{
    glClear( GL_COLOR_BUFFER_BIT );
    glDrawArrays( GL_POINTS, 0, numPoints );
    glFlush( );
}
```

Boilerplate GLUT

- The `display()` function
 - `glClear()` tells the GPU to set the contents of the frame buffer to a OpenGL state value.
 - You can set this color via `glClearColor()`
 - Other tokens, which we will talk about later, can be or'd together here.

```
void display( void )
{
    glClear( GL_COLOR_BUFFER_BIT );
    glDrawArrays( GL_POINTS, 0, numPoints );
    glFlush( );
}
```

Boilerplate GLUT

- The `display()` function
 - `glDrawArrays()` is a function we saw earlier.
 - It is what actually sends the array of vertices down the GPU pipeline to be rendered.
 - Here we are telling it to `numPoints` of vertices, stored in buffer 0 and to treat the data as points.
 - Need to define and load “buffer 0”

```
void display( void )
{
    glClear( GL_COLOR_BUFFER_BIT );
    glDrawArrays( GL_POINTS, 0, numPoints );
    glFlush( );
}
```

Boilerplate GLUT

- The `display()` function
 - `glFlush()` ensures all the commands we have issued to the GPU have been sent.
 - Remember, the GPU is a separate asynchronous device – commands are fed to it via a queue.
 - The command is also useful in network based windowing systems like x-windows.

```
void display( void )
{
    glClear( GL_COLOR_BUFFER_BIT );
    glDrawArrays( GL_POINTS, 0, numPoints );
    glFlush( );
}
```


Vertex Array

- The `init()` function
 - Is a good place to create the vertex array we reference in the `display()` function.
 - We want to get the points that make up our sierpinski gasket copied into a buffer on the GPU

```
Guint vArray, buffer;
void init( void )
{
    ...
    glGenVertexArrays( 1, &vArray );
    glBindVertexArray( vArray );
    ...
    glGenBuffers( 1, &buffer );
    glBindBuffer( GL_ARRAY_BUFFER, buffer );
    glBufferData( GL_ARRAY_BUFFER, sizeof( points ), points, GL_STATIC_DRAW );
    ...
}
```

Vertex Array

- Initialize a vertex array
 - Call `glGenVertexArrays ()` to get an ID.
 - We pass in a pointer to an array of references.
 - In this case we are only asking for one (1) ID.

```
Guint vArray, buffer;
void init( void )
{
    ...
    glGenVertexArrays( 1, &vArray );
    glBindVertexArray( vArray );
    ...
    glGenBuffers( 1, &buffer );
    glBindBuffer( GL_ARRAY_BUFFER, buffer );
    glBufferData( GL_ARRAY_BUFFER, sizeof( points ), points, GL_STATIC_DRAW );
    ...
}
```

Vertex Array

- Initialize a vertex array
 - Call `glBindVertexArray()` to bind, or enable, the vertex array referenced by `vArray`.
 - Once bound the array stays bound until we change it.
 - Binding is common in OpenGL – it's a form of state.

```
Gluint vArray, buffer;
void init( void )
{
    ...
    glGenVertexArrays( 1, &vArray );
    glBindVertexArray( vArray );
    ...
    glGenBuffers( 1, &buffer );
    glBindBuffer( GL_ARRAY_BUFFER, buffer );
    glBufferData( GL_ARRAY_BUFFER, sizeof( points ), points, GL_STATIC_DRAW );
    ...
}
```

Vertex Array

- Initialize a vertex array
 - We now have a vertex array with nothing in it.
 - Call `glGenBuffers ()` to create a buffer reference.
 - Same pattern as before.

```
Guint vArray, buffer;
void init( void )
{
    ...
    glGenVertexArrays( 1, &vArray );
    glBindVertexArray( vArray );
    ...
    glGenBuffers( 1, &buffer );
    glBindBuffer( GL_ARRAY_BUFFER, buffer );
    glBufferData( GL_ARRAY_BUFFER, sizeof( points ), points, GL_STATIC_DRAW );
    ...
}
```

Vertex Array

- Initialize a vertex array
 - Now we bind the buffer (make it active) and describe what kind of buffer it is – `GL_ARRAY_BUFFER`
 - Once bound it will remain active until we change it.

```
Guint vArray, buffer;
void init( void )
{
    ...
    glGenVertexArrays( 1, &vArray );
    glBindVertexArray( vArray );
    ...
    glGenBuffers( 1, &buffer );
    glBindBuffer( GL_ARRAY_BUFFER, buffer );
    glBufferData( GL_ARRAY_BUFFER, sizeof( points ), points, GL_STATIC_DRAW );
    ...
}
```

Vertex Array

- Initialize a vertex array
 - Finally, we make the connection to our data.
 - We give the size of the *array* and a pointer.
 - `GL_STATIC_DRAW` tells the driver that the data doesn't change between calls to `glDrawArrays()`

```
Guint vArray, buffer;
void init( void )
{
    ...
    glGenVertexArrays( 1, &vArray );
    glBindVertexArray( vArray );
    ...
    glGenBuffers( 1, &buffer );
    glBindBuffer( GL_ARRAY_BUFFER, buffer );
    glBufferData( GL_ARRAY_BUFFER, sizeof( points ), points, GL_STATIC_DRAW );
    ...
}
```

Shaders

- Vertex shader
 - When we call `glDrawArrays()` each vertex triggers an execution of the vertex shader.
 - We are just passing the data through for now.
 - The shader is written in GLSL, a C-like language
 - The input is `vPosition` and the output is `gl_position`

```
in vec4 vPosition;  
void main( )  
{  
    gl_position = vPosition;  
}
```

Shaders

- Vertex shader
 - `gl_position` is a state variable in OpenGL
 - It's used to pass the final (transformed) vertex down the pipeline to the rasterizer from the shader.
 - The input to the shader is indicated by the `in` modifier to `vPosition`
 - Later we will see how the connection is made to our vertex array.

```
in vec4 vPosition;  
void main( )  
{  
    gl_position = vPosition;  
}
```


Shaders

- Fragment shader
 - The fragment shader executes on the output of the rasterizer.
 - The rasterizer will generate fragments corresponding to the primitives sent down the OpenGL pipeline destined for the frame buffer.
 - At a minimum each execution must output a color for the fragment by setting `gl_frag_color`

```
void main( )  
{  
    gl_frag_color = vec4( 1.0, 0.0, 0.0, 1.0 );  
}
```

Shaders

- The shaders are actual programs
 - That execute *on the GPU*
 - That means they have to be compiled and linked.

```
...
GLuint program glCreateProgram( void );
GLuint vShader = glCreateShader( GL_VERTEX_SHADER );
glShaderSource( vShader, 1, (const GLchar**)&vSource, NULL );
glCompileShader( vShader );
// should check for success here, see book code
glAttachShader( program, vShader );
GLuint fShader = glCreateShader( GL_FRAGMENT_SHADER );
glShaderSource( fShader, 1, (const GLchar**)&fSource, NULL );
glCompileShader( fShader );
// should check for success here, see book code
glAttachShader( program, vShader );
glLinkProgram( program );
// should check for success here, see book code
glUseProgram( program );
...
```

Shaders

- A set of shaders exist within a *program*
 - First, we create the program with `glCreateProgram()`

```
...
Gluint program glCreateProgram( void );
Gluint vShader = glCreateShader( GL_VERTEX_SHADER );
glShaderSource( vShader, 1, (const Glchar**)&vSource, NULL );
glCompileShader( vShader );
// should check for success here, see book code
glAttachShader( program, vShader );
Gluint fShader = glCreateShader( GL_FRAGMENT_SHADER );
glShaderSource( fShader, 1, (const Glchar**)&fSource, NULL );
glCompileShader( fShader );
// should check for success here, see book code
glAttachShader( program, vShader );
glLinkProgram( program );
// should check for success here, see book code
glUseProgram( program );
...
```

Shaders

- Create a shader by selecting the type
 - First, the vertex shader `GL_VERTEX_SHADER`
 - Save the reference for later steps.

```
...
GLuint program glCreateProgram( void );
GLuint vShader = glCreateShader( GL_VERTEX_SHADER );
glShaderSource( vShader, 1, (const Glchar**)&vSource, NULL );
glCompileShader( vShader );
// should check for success here, see book code
glAttachShader( program, vShader );
GLuint fShader = glCreateShader( GL_FRAGMENT_SHADER );
glShaderSource( fShader, 1, (const Glchar**)&fSource, NULL );
glCompileShader( fShader );
// should check for success here, see book code
glAttachShader( program, vShader );
glLinkProgram( program );
// should check for success here, see book code
glUseProgram( program );
...
```

Shaders

- Next, assign the source to the shader.
 - `glShaderSource()` attaches source strings to the shader.

```
...
GLuint program glCreateProgram( void );
GLuint vShader = glCreateShader( GL_VERTEX_SHADER );
glShaderSource( vShader, 1, (const GLchar**)&vSource, NULL );
glCompileShader( vShader );
// should check for success here, see book code
glAttachShader( program, vShader );
GLuint fShader = glCreateShader( GL_FRAGMENT_SHADER );
glShaderSource( fShader, 1, (const GLchar**)&fSource, NULL );
glCompileShader( fShader );
// should check for success here, see book code
glAttachShader( program, vShader );
glLinkProgram( program );
// should check for success here, see book code
glUseProgram( program );
...
```

Shaders

- Compile the shader.
 - After which you should check the result by querying the shader – see the book example.

```
...
GLuint program glCreateProgram( void );
GLuint vShader = glCreateShader( GL_VERTEX_SHADER );
glShaderSource( vShader, 1, (const GLchar**)&vSource, NULL );
glCompileShader( vShader );
// should check for success here, see book code
glAttachShader( program, vShader );
GLuint fShader = glCreateShader( GL_FRAGMENT_SHADER );
glShaderSource( fShader, 1, (const GLchar**)&fSource, NULL );
glCompileShader( fShader );
// should check for success here, see book code
glAttachShader( program, vShader );
glLinkProgram( program );
// should check for success here, see book code
glUseProgram( program );
...
```

Shaders

- Attach the shader to the program.

```
...
GLuint program glCreateProgram( void );
GLuint vShader = glCreateShader( GL_VERTEX_SHADER );
glShaderSource( vShader, 1, (const GLchar**)&vSource, NULL );
glCompileShader( vShader );
// should check for success here, see book code
glAttachShader( program, vShader );
GLuint fShader = glCreateShader( GL_FRAGMENT_SHADER );
glShaderSource( fShader, 1, (const GLchar**)&fSource, NULL );
glCompileShader( fShader );
// should check for success here, see book code
glAttachShader( program, vShader );
glLinkProgram( program );
// should check for success here, see book code
glUseProgram( program );
...
```

Shaders

- Repeat the process for the fragment shader.

```
...
GLuint program glCreateProgram( void );
GLuint vShader = glCreateShader( GL_VERTEX_SHADER );
glShaderSource( vShader, 1, (const GLchar**)&vSource, NULL );
glCompileShader( vShader );
// should check for success here, see book code
glAttachShader( program, vShader );
GLuint fShader = glCreateShader( GL_FRAGMENT_SHADER );
glShaderSource( fShader, 1, (const GLchar**)&fSource, NULL );
glCompileShader( fShader );
// should check for success here, see book code
glAttachShader( program, vShader );
glLinkProgram( program );
// should check for success here, see book code
glUseProgram( program );
...
```


Shaders

- Link the program's two shaders
 - You should, again, check the result – see the book example.

```
...
GLuint program glCreateProgram( void );
GLuint vShader = glCreateShader( GL_VERTEX_SHADER );
glShaderSource( vShader, 1, (const GLchar**)&vSource, NULL );
glCompileShader( vShader );
// should check for success here, see book code
glAttachShader( program, vShader );
GLuint fShader = glCreateShader( GL_FRAGMENT_SHADER );
glShaderSource( fShader, 1, (const GLchar**)&fSource, NULL );
glCompileShader( fShader );
// should check for success here, see book code
glAttachShader( program, vShader );
glLinkProgram( program );
// should check for success here, see book code
glUseProgram( program );
...
```

Shaders

- Finally we *use* the program
 - Use has the same effect as bind.
 - It stays in effect until we *use* something else.

```
...
GLuint program glCreateProgram( void );
GLuint vShader = glCreateShader( GL_VERTEX_SHADER );
glShaderSource( vShader, 1, (const GLchar**)&vSource, NULL );
glCompileShader( vShader );
// should check for success here, see book code
glAttachShader( program, vShader );
GLuint fShader = glCreateShader( GL_FRAGMENT_SHADER );
glShaderSource( fShader, 1, (const GLchar**)&fSource, NULL );
glCompileShader( fShader );
// should check for success here, see book code
glAttachShader( program, vShader );
glLinkProgram( program );
// should check for success here, see book code
glUseProgram( program );
...
```

Shaders

- Whew, but not done yet...
 - We still have to make the connection from our vertex array to our shiny new shader program.
 - The first step is to retrieve the location of the variable we are using for input in the vertex shader.

```
#define BUFFER_OFFSET( bytes ) ((Glvoid *)(bytes))  
GLuint location;  
location = glGetUniformLocation( program, "vPosition" );  
glVertexAttribPointer( location, 2, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET( 0 ) );  
glEnableVertexAttribArray( location );
```

Shaders

- Attach our data to the shader
 - The 2 and GL_FLOAT describe our data in the vertex array – 2D points.
 - The next two values specify whether the data should be normalized and the stride of the data.
 - Lastly, we specify a pointer to the data.
 - This is not a pointer to the memory itself but a pointer to the location in the buffer object – can be confusing.

```
#define BUFFER_OFFSET( bytes ) ((Glvoid *)(bytes))  
Gluint location;  
location = glGetAttribLocation( program, "vPosition" );  
glVertexAttribPointer( location, 2, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET( 0 ) );  
glEnableVertexAttribArray( location );
```

All the pieces

- At this point, there is enough to get something on the display.
- This is a lot of work for something simple.
 - If you are thinking this, you're right, it is.
 - For such a simple program this is overkill and I won't show you how easy it is to do the "old way".
 - However, time marches on and it is time to learn the "new way" since no one would ever write such a simple program anyway, except for learning.
 - Plus this approach allows for much more control.

Interaction Basics

- GLUT provides callbacks for user input.
 - Mouse input is collected by registering a function with `glutMouseFunc ()`
 - Keyboard input is collected by registering a function with `glutKeyboardFunc ()`

```
void mouseHandler( int button, int state, int x, int y )
{
    if ( button == GLUT_RIGHT_BUTTON && state == GLUT_DOWN ) exit( 0 );
}

void keyHandler( unsigned char key, int x, int y )
{
    if ( key == 'q' || key == 'Q' ) exit( 0 );
}
```

Assignment #1

- Due Wednesday (10/15) by 11:55pm
 - Upload to CCLE
- Implement Sierpinski Gasket in 2D.
 - Details in assignment which you can download from.
 - Get started! It's simple, but you may run into problems.

Next Time

- More FUN!
- Move on to setting up first real assignment