UCLA

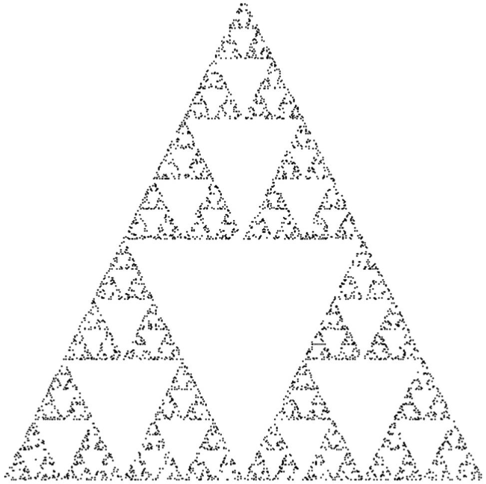
# CS174A : Introduction to Computer Graphics

Kinsey 1240
MW 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 we draw.
  - 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

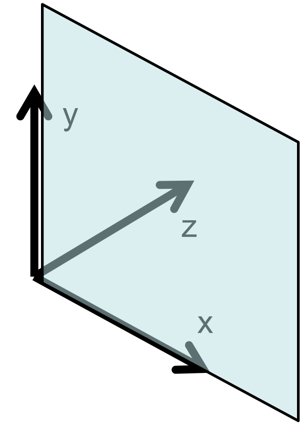
- Immediate Mode
  - Most work, but no storage.
- Retained Mode #1
  - Less work (after first time), but need storage and transfer to GPU every time.
- Retained Mode #2
  - Same work as #1, temporary storage on host, storage on GPU, single transfer time.

# 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 an 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 mathematical concept.
- A vertex $v$ is a thing
  - understood to be part of the description of something more complex.
    - A point can be represented by a vertex.
    - A vertex is part of the definition of many primitives
      - Point, line, triangle, some other GL primitive.

# Sierpinski Gasket Code

- Code
  - Nothing fancy but will get us something to render on our display. (see example in book)

```
#include "vec.h"
typedef vec2 point2;
main( )
{
    const int NumPoints = 5000;
    // define a triangle in plane z=0
    point2 verticies[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 do we create a window to render into?
  - How much of the image plane will appear?
  - How long will the image appear on the display?

# WebGL API

- WebGL 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 objects we can draw.
- They are the "**what**" of the API
  - Points
  - Lines
  - Triangles

- Any other geometry is made up from these primitives.

# Attribute Functions

- This is the "**how**" of the API
    - Color
    - Lines style
    - Point size
- Attributes can apply to different *contexts*
    - Globally
    - Per vertex
    - Per fragment
        - Fragments are often confused with pixels

# Viewing Functions

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

# Transformation Functions

- WebGL, it sometimes seems, is mostly made up of these transformations.
- Transformation functions for
  - Geometry: translation, scaling, rotation.
  - Viewing: uses transformations heavily.
- Transformations can occur in the application itself or within a *Shader*.
  – Shaders are the little programs that get loaded into your GPU and are part of the WebGL rendering pipeline.

# 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.
- Not actually part of WebGL
    - – Part of browser or desktop/mobile environment

# Control Functions

- Catch-all for handing the OS and GPU.
- Interfacing with the windowing system.
  - This is required for almost any graphics program
  - Can be very complex in and of itself.
  - We will use HTML, CSS
  - There are others
    - GLUT: GL Utility Toolkit
    - GLX, AGL, WGL, others specific to OS.

# Query Functions

- Used to interrogate WebGL *state*.
- Can also sometimes be used to query
    - 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
    - Most browsers support WebGL 1.0

# Browsers

- WebGL
- See: http://caniuse.com/#feat=webgl

- WebGL 2.0 is available in some form
  - Chrome and Firefox
  - See:
    https://www.khronos.org/webgl/wiki/Getting_a_WebGL_Implementation

# 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

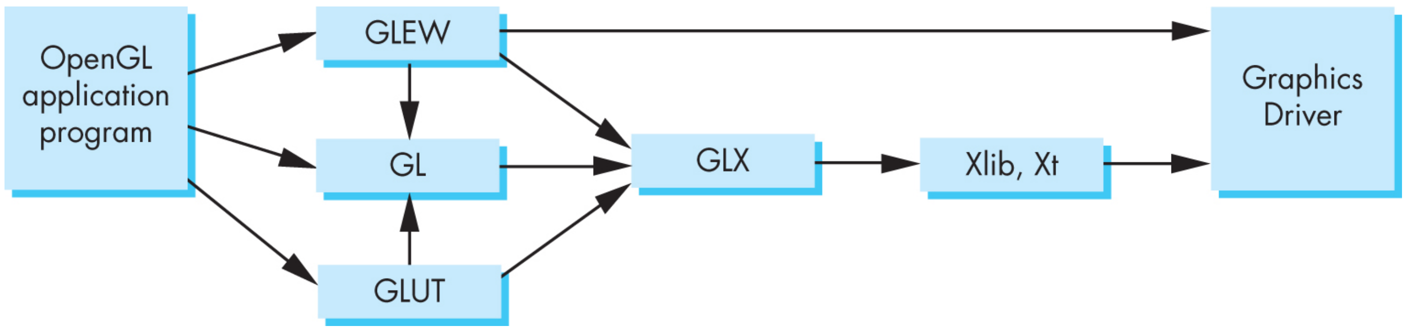
- State manipulation is primarily about
  - Enabling or disabling features
  - Parameter setting
- Older versions of OpenGL (<3.1)
  - Many many internal state variables
- Recent versions of OpenGL (>=3.1)
  - You define what you need and use via *Shaders* that you must also define (program).
  - A lot more work – but a lot 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 WebGL
- State is also not bound to any specific primitive
    - 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!!

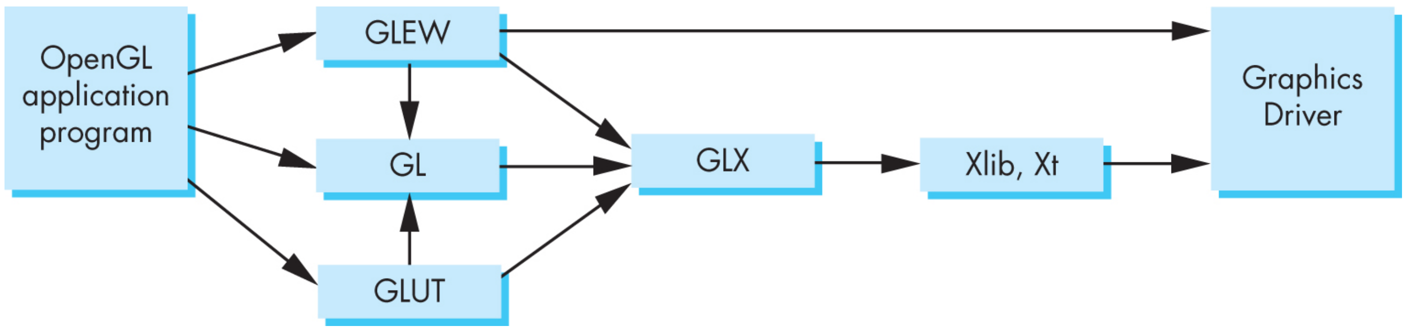
# OpenGL is a Library

- Just like any other library you might use.
  - Usually libGL.{lib,so,ksym,dll,etc…}
    - Or, in the case of WebGL, it's built into your browser (i.e. the library is linked with the browser itself)
  - All functions begin with "gl"
  - Shaders are written in a C-like language, GLSL



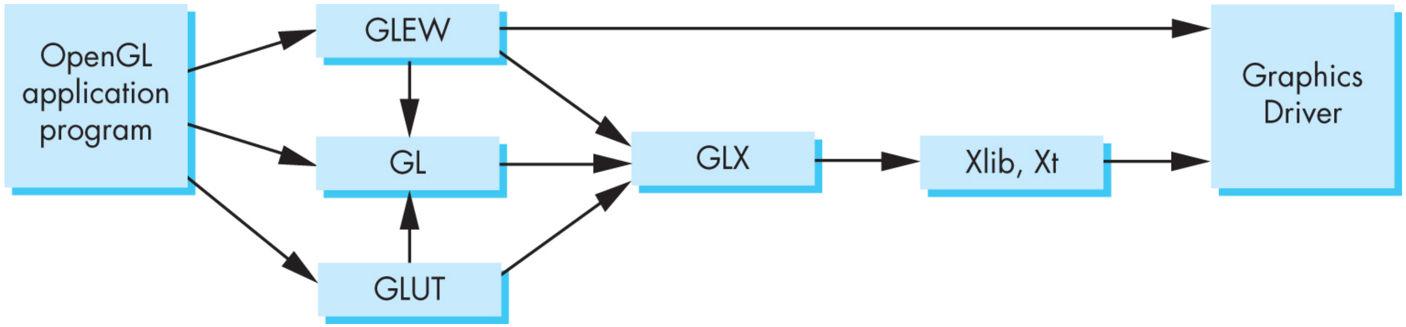
# OpenGL is a Library

- 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, now we use Cocoa
  - Windows it is called wgl (OpenGL is not DirectX)
  - 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 (or similar)`
  - 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.

# WebGL

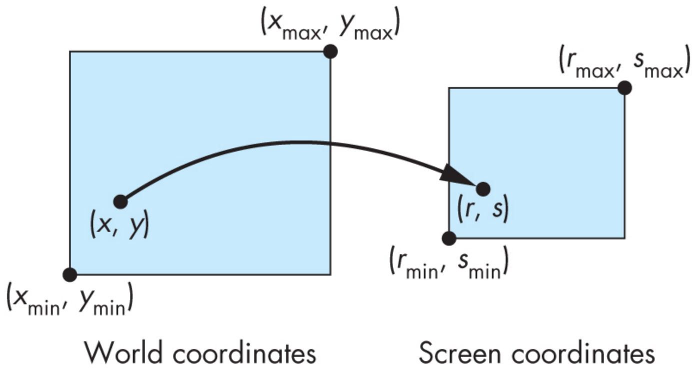
- Hopefully, it is apparent that WebGL
  - Is based on OpenGL
  - Abstracts away the interfaces to your OS and GPU
  - Simpler than OpenGL and more restricted.
    - More restricted than OpenGL ES

# Coordinate systems

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

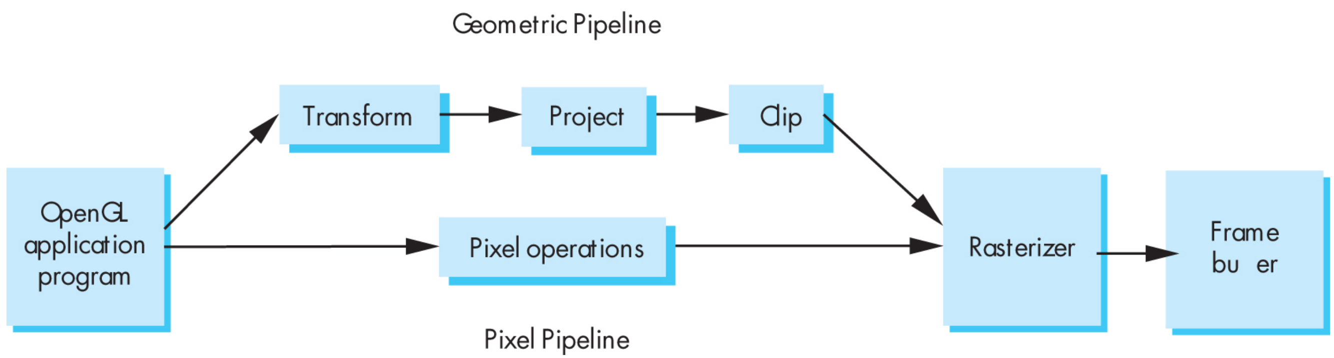
# Coordinate systems

- The can be (are) other coordinate systems …but for now
- Transformations
  - 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 of primitive: geometry and raster
  - Geometry, in world space, has to be transformed, projected and clipped into screen space.
  - Raster data is already in screen space (later)



# Primitives of course!

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

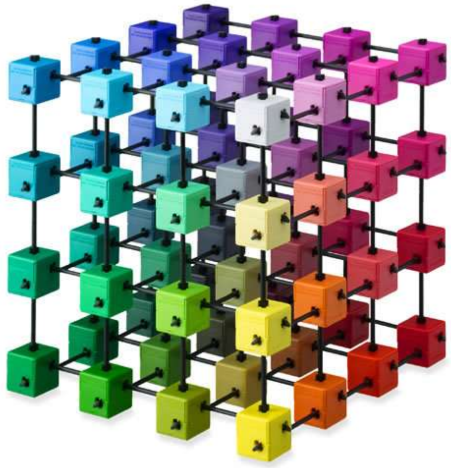
# Primitives of course!

- Given an WebGL primitive type
  - GL_POINT, GL_LINE, GL_TRIANGLE, etc.
- A primitive can be drawn with command
  - `gl.drawArrays(GL_POINT,0,numPoints)`
  - OMG – first chunk of WebGL at slide 34!

- Actually quite a bit of effort is required before this function will even work, unfortunately.

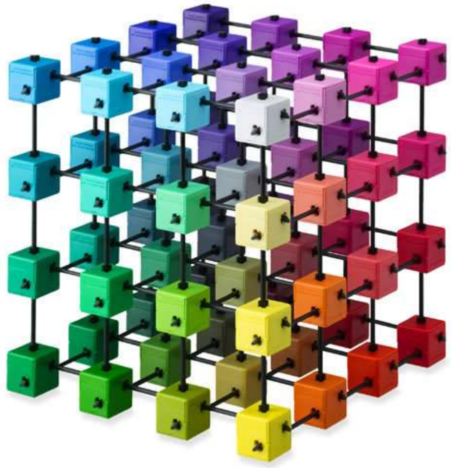
# For starters…

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



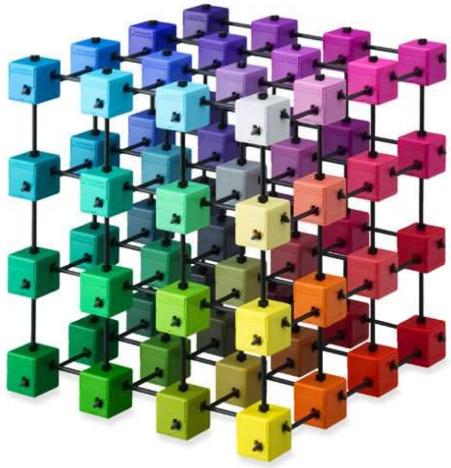
# Super Basic Color

- 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 primaries at max intensity
    - Black: all primaries at zero
  - GPU stores as primaries as bits
    - Typically 8-bits for each
    - Plus one more for “alpha”
  - You will see RGB and RGBA
  - We will revisit the A later on



# Super Basic Color

- WebGL 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

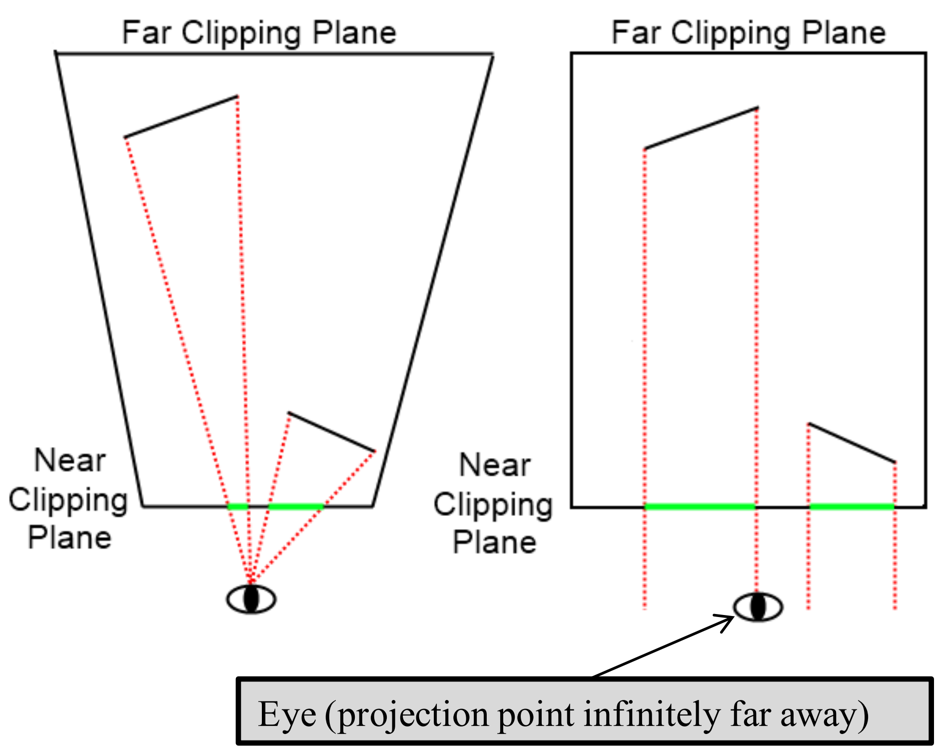


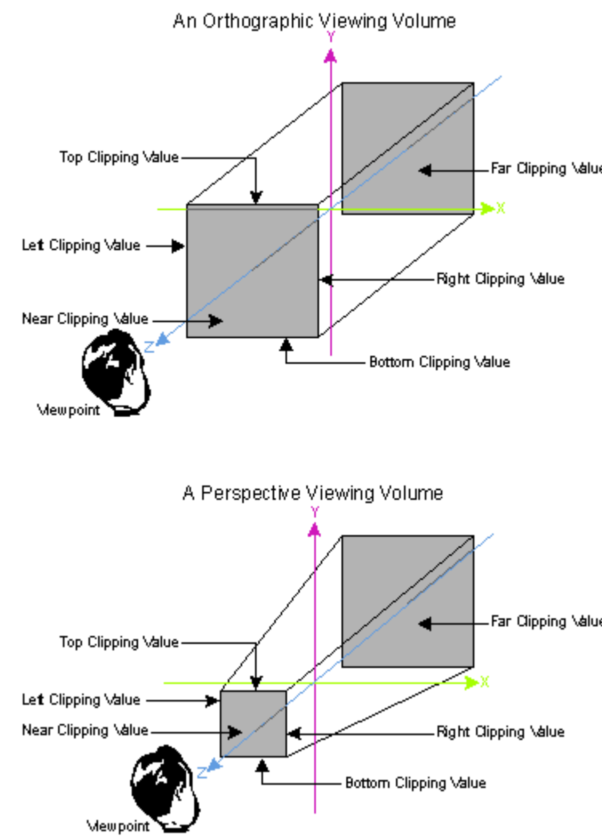
# Viewing

- Two types of transformation projections are useful for viewing
  - Perspective
    - What we talked about last time (without actually naming it)
    - 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

# Viewing

- Two views to illustrate the difference.



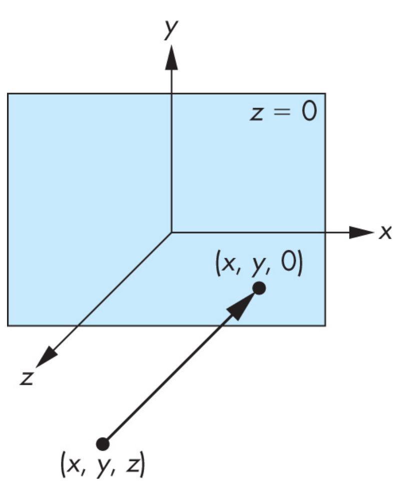



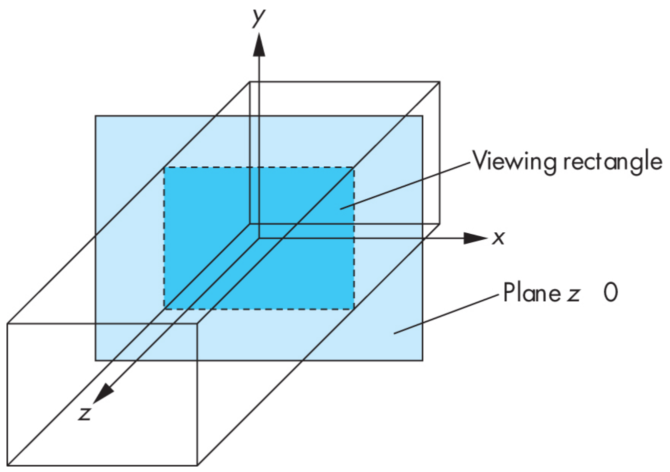
# Viewing

- 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)

# Viewing

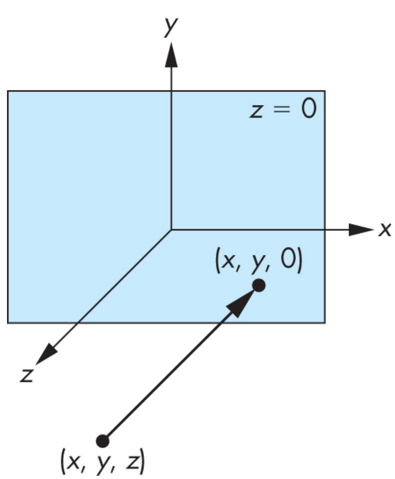
- A 2D image plane can be defined by setting z=0 in 3D space (WebGL always in a 3D space)
  - WebGL defines a canonical view volume and projection point.
  - We can use it without any additional transformations.
  - That default “looks” down the –z axis.

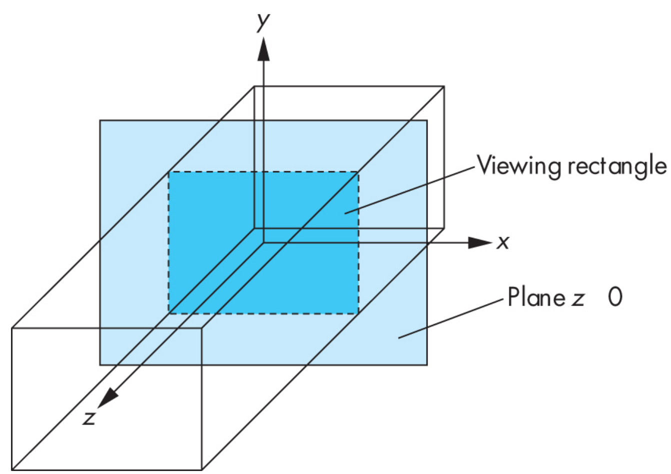


# Viewing

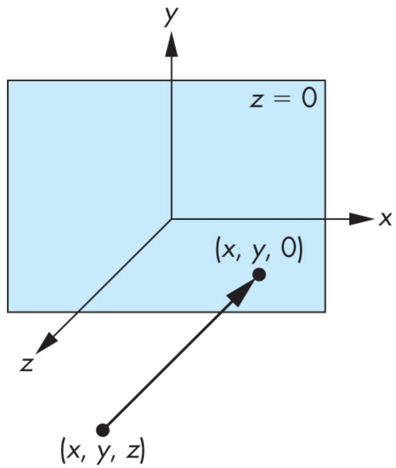
- The canonical view volume
  - Defines what is ultimately projected onto the window and what is clipped.
  - Defined by default as a cube, (-1,-1,-1) to (1,1,1)

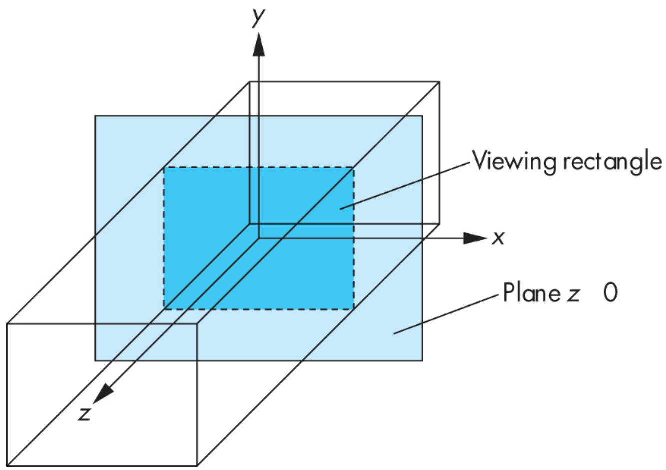


# Viewing

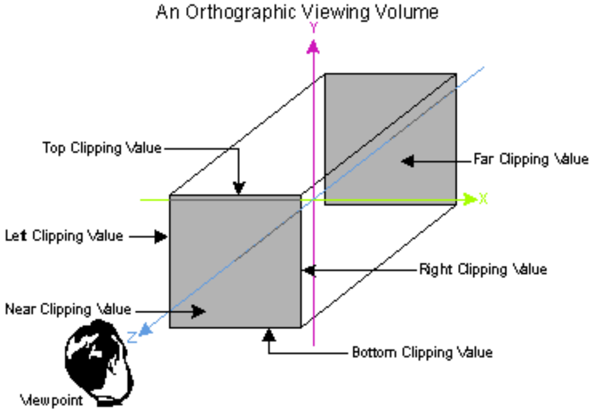
- Normally, a transformation would be needed to convert our desired projection view volume into the canonical view volume. (i.e. Perspective to Canonical)

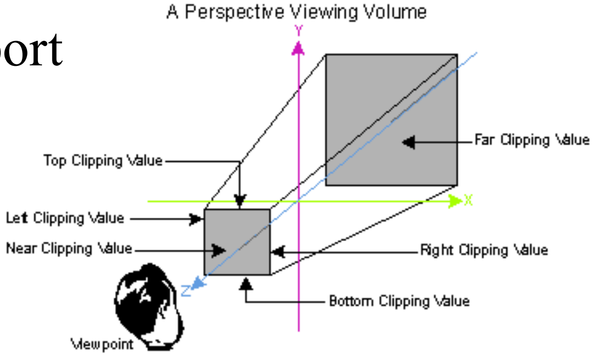


# Viewing

- 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



# 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 Thing™, for now

# Boilerplate

- Let's write some code already…
- We are going to use WebGL
  - So let's see what is at the core of every WebGL based program so we can put this all together.
  - TAs will make some basic utility files available
    - Feel free to use them or not.

# Boilerplate WebGL

- Here is the basic WebGL HTML5 file

```
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;charset=utf-8" >
<title>WebGL</title>
</head>
<body>
<canvas id="gl-canvas" width="512" height="512">
Oops ... your browser doesn't support the HTML5 canvas element
</canvas>
</body>
</html>
```

# Boilerplate WebGL

- WebGL code exists in <script> form (Javascript)
  - "gl-canvas" is from the id of the <canvas>
  - We pass this reference to WebGL
    - WebGLUtils is a utility from Google which we have to include in our HTML file (gasket.js)

```
var gl;

window.onload = function init()
{
    var canvas = document.getElementById( "gl-canvas" );

    gl = WebGLUtils.setupWebGL( canvas );
    if ( !gl ) { alert( "WebGL isn't available" ); }
...
```

# Boilerplate WebGL

- Need to include our scripts into HTML
  - Could also do this inline (but messy)
    - Note dependency order!

```
…
</head>

<script type="text/javascript" src=”webgl-utils.js"></script>
<script type="text/javascript" src="gasket.js"></script>

<body>
<canvas id="gl-canvas" width="512" height="512">
Oops ... your browser doesn't support the HTML5 canvas element
</canvas>
</body>
</html>
```

# Code to generate verticies

```
var gl;
var points;
var NumPoints = 5000;

window.onload = function init()
{
    var canvas = document.getElementById( "gl-canvas" );
    gl = WebGLUtils.setupWebGL( canvas );
    if ( !gl ) { alert( "WebGL isn't available" ); }
    var vertices = [ vec2( -1, -1 ), vec2(  0,  1 ), vec2(  1, -1 ) ];
    var u = add( vertices[0], vertices[1] );
    var v = add( vertices[0], vertices[2] );
    var p = scale( 0.25, add( u, v ) );
    points = [ p ];
    for ( var i = 0; points.length < NumPoints; ++i ) {
        var j = Math.floor(Math.random() * 3);
        p = add( points[i], vertices[j] );
        p = scale( 0.5, p );
        points.push( p );
    }
…
```

# Boilerplate WebGL

- Set the viewport
  - Notice we set it to the pixel size of the window
  - Canvas width and height
- When we clear the viewport we want it to be white
  - Remember, white is all 1's – notice we are NOT actually clearing anything here! (we are setting state)

```
window.onload = function init()
{
        …
        points.push( p );
    }
    gl.viewport( 0, 0, canvas.width, canvas.height );
    gl.clearColor( 1.0, 1.0, 1.0, 1.0 );
    …
```

# Boilerplate WebGL

- Let initialize our Shaders
  - Utility funcion ‘initShaders’ to connect/compile/link our shaders into our program

```
…
</head>
<script type="text/javascript" src=”webgl-utils.js"></script>
<script type="text/javascript" src=”initShaders.js"></script>
<script type="text/javascript" src="gasket.js"></script>
<body>
…
```

```
…
    gl.viewport( 0, 0, canvas.width, canvas.height );
    gl.clearColor( 1.0, 1.0, 1.0, 1.0 );

    var program = initShaders( gl, "vertex-shader", "fragment-shader" );
    gl.useProgram( program ); // setting state of which shaders to use
…
```

# Boilerplate WebGL

- Where are these Shaders anyway?
  - In the HTML file (one way, you can also put them in their own file)

```
<title>WebGL</title>
<script id="vertex-shader" type="x-shader/x-vertex">
attribute vec4 vPosition;
Void main()
{
    gl_PointSize = 1.0;
    gl_Position = vPosition;
}
</script>
</head>
<script type="text/javascript" src=”webgl-utils.js"></script>
<script type="text/javascript" src=”initShaders.js"></script>
<script type="text/javascript" src="gasket.js"></script>
<body>
…
```

# Boilerplate WebGL

- This is the vertex shader
  - As primitives are rendered their vertices pass through the vertex shader
  - Shaders use a syntax similar to C that is called GLSL
  - The vertex passed into the vertex shader is called vPosition
  - We set two *output* variables
    - gl_PointSize : sets the pixel size of points (both of these names are defined by GLSL)
    - gl_Position : this is the, potentially, modified vertex which will be passed through the rest of the pipeline (Here we just pass the value through unchanged)

```
<script id="vertex-shader" type="x-shader/x-vertex">
attribute vec4 vPosition;
void main()
{
    gl_PointSize = 1.0;
    gl_Position = vPosition;
}
</script>
```

# Boilerplate WebGL

- This is the fragment shader
  - As primitives are rasterized their fragments pass through the fragment shader
  - Here we have no input for this example because we are setting fixed color
    - Output color is RED (R,G,B,A) = (1.0,0.0,0.0,1.0)
  - Precision has to do with the precision of the GPU floating point calcs
  - We set one output variable
    - gl_FragColor : this is the output fragment color (name defined by GLSL)

```
<script id="fragment-shader" type="x-shader/x-fragment">
precision mediump float;
void main()
{
    gl_FragColor = vec4( 1.0, 0.0, 0.0, 1.0 );
}
</script>
```

# Boilerplate WebGL

- WebGL is setup, lets draw the points
  - We tell WebGL that the shaders we loaded are to be used (useProgram)
  - Create a buffer (this is in the GPU), notice we don't specify a size, this is just a handle.
  - We Bind that buffer (set the state of which buffer we will be using, similar to use Program – OpenGL/WebGL is not super consistent this way)
  - Then we transfer the actual vertex data we wish to draw into the buffer.

```
…
    var program = initShaders( gl, "vertex-shader", "fragment-shader" );
    gl.useProgram( program );
    // Load the data into the GPU
    var bufferId = gl.createBuffer();
    gl.bindBuffer( gl.ARRAY_BUFFER, bufferId );
    gl.bufferData( gl.ARRAY_BUFFER, flatten(points), gl.STATIC_DRAW );
…
```

# Boilerplate WebGL

- WebGL is setup, lets draw the points
  - The bufferData function implicitly sets the size.
    - You can ignore flatten(), it's an artifact of the authors utility library to get the values into a single array
  - gl.STATIC_DRAW – this is a hint to WebGL that we will not be changing this data. The hint helps optimize where the buffer is actually placed in memory (in CPU or GPU memory).

```
…
   var program = initShaders( gl, "vertex-shader", "fragment-shader" );
   gl.useProgram( program );
   // Load the data into the GPU
   var bufferId = gl.createBuffer();
   gl.bindBuffer( gl.ARRAY_BUFFER, bufferId );
   gl.bufferData( gl.ARRAY_BUFFER, flatten(points), gl.STATIC_DRAW );
…
```

# Boilerplate WebGL

- WebGL is setup, lets draw the points (uh, not yet)
  - The buffer holding our data is not enough.
  - We have to tell WebGL what kind of data it is and how to pass it through to the vertex shader.
  - First we get a reference to our vPosition variable from our vertex shader
  - Then we attach it to and describe our data (an array of 2D points)
  - We finally enable the vertex array (enable, bind, use, etc. try not to get confused – this is just more state information)

```
...
    var bufferId = gl.createBuffer();
    gl.bindBuffer( gl.ARRAY_BUFFER, bufferId );
    gl.bufferData( gl.ARRAY_BUFFER, flatten(points), gl.STATIC_DRAW );

    var vPosition = gl.getAttribLocation( program, "vPosition" );
    gl.vertexAttribPointer( vPosition, 2, gl.FLOAT, false, 0, 0 );
    gl.enableVertexAttribArray( vPosition );
...
```

# Boilerplate WebGL

- WebGL is setup, lets draw the points
  - Wait a second, "attach it to and describe our data"…
  - How does it know which data? We did not pass any reference to our Buffer!?!?
  - Because of state! When we "bound" our buffer to the bufferId that buffer became the active buffer (confused? Remember, it's all about state)
  - Other functions, like vertexAttribPointer, etc. *assume* a bound buffer
  - If you didn't have one WebGL would be in an error state.

```
...
    var bufferId = gl.createBuffer();
    gl.bindBuffer( gl.ARRAY_BUFFER, bufferId );
    gl.bufferData( gl.ARRAY_BUFFER, flatten(points), gl.STATIC_DRAW );

    var vPosition = gl.getAttribLocation( program, "vPosition" );
    gl.vertexAttribPointer( vPosition, 2, gl.FLOAT, false, 0, 0 );
    gl.enableVertexAttribArray( vPosition );
…
```

# Boilerplate WebGL

- A word about error state
- When you have an error generally all you will see is…

# Boilerplate WebGL

- So you need to check

# Boilerplate WebGL

- WebGL is setup, lets draw the points
  - Still nothing on the screen... sigh
  - Create a function to contain the code to actually draw something
    - A function is not strictly needed here but will be useful in more complex programs
  - First clear the color buffer (it will be cleared to the value we set earlier)
  - Actually draw the data as Point primitives. (using the bound data buffer)

```
…
    var vPosition = gl.getAttribLocation( program, "vPosition" );
    gl.vertexAttribPointer( vPosition, 2, gl.FLOAT, false, 0, 0 );
    gl.enableVertexAttribArray( vPosition );
    render();
};

function render() {
    gl.clear( gl.COLOR_BUFFER_BIT );
    gl.drawArrays( gl.POINTS, 0, points.length );
}
```

# 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.