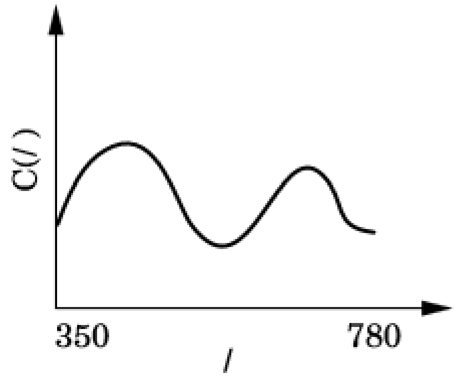
Introduction to Computer Graphics with WebGL

Week2

Instructor: Hooman Salamat

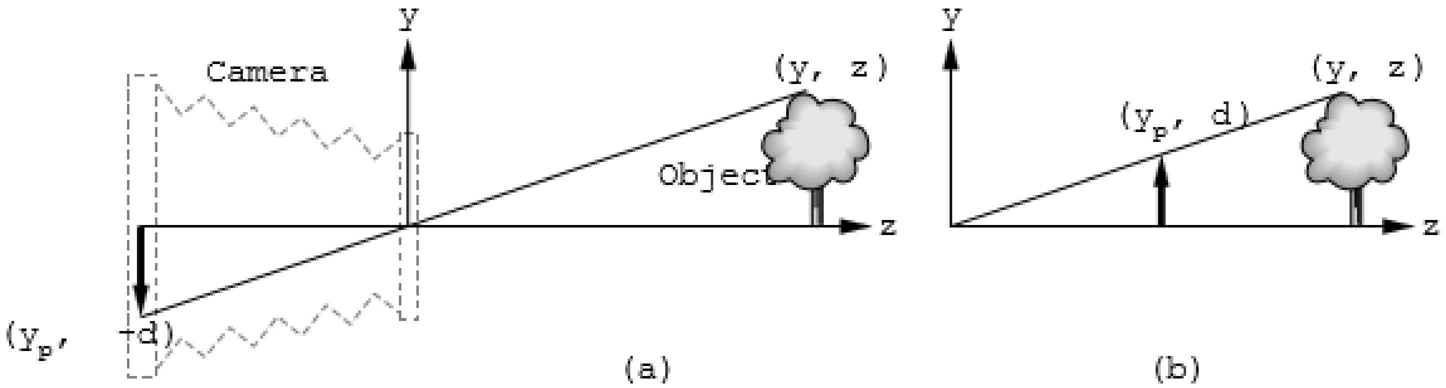
RGB Colors



- We usually use the three-color theory. A color C is taken to be a linear combination of red R, green G, and blue B, (RGB),
- $C = T1R + T2G + T3B$.
- The coefficients T1, T2, T3 are the tristimulus values. This theory only approximates reality, in which a color can be regarded as a function C giving the strength $C(\lambda)$ of each wavelength λ.
- But in practice the brain interprets color through a triple (A1,A2,A3), where $A_i = Z\ S_i(\lambda)C(\lambda)d\lambda$,
- and so the approximation is quite good. Two colors that match visually are called a ‘metameric pair’, though they may not have the same color distributions.

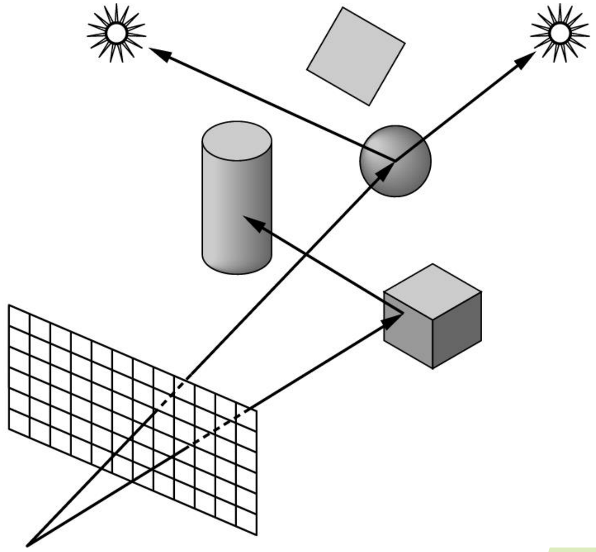
Image Formation Revisited

- Can we mimic the synthetic camera model to design graphics hardware software?
- In the synthetic camera model we avoid the inversion by placing the film plane, called the projection plane, in front of the lens.



Physical Approaches

- **Ray tracing**: follow rays of light from center of projection until they either are absorbed by objects or go off to infinity
  - Can handle global effects
    - Multiple reflections
    - Translucent objects
  - Slow
  - Must have whole data base available at all times



- **Radiosity**: Energy based approach
  - Very slow

Practical Approach

- The four major steps in the imaging process are:
- 1. Transformation 2. Clipping 3. Projection 4. Rasterization
- These operations can be pipelined, i.e. applied in parallel, to speed up the imaging process. Several steps can be represented by 4 × 4 matrices. Such matrices can be multiplied together, or concatenated, a process which lends itself to pipeline architectures and parallelism.

- All steps can be implemented in hardware on the graphics card

Vertex Processing

- Much of the work in the pipeline is in converting object representations from one coordinate system to another
  - Object coordinates
  - Camera (eye) coordinates
  - Screen coordinates
- Every change of coordinates is equivalent to a matrix transformation
- Vertex processor also computes vertex colors

Projection

- *Projection* is the process that combines the 3D viewer with the 3D objects to produce the 2D image
  - Perspective projections: all projectors meet at the center of projection
  - Parallel projection: projectors are parallel, center of projection is replaced by a direction of projection

Primitive Assembly

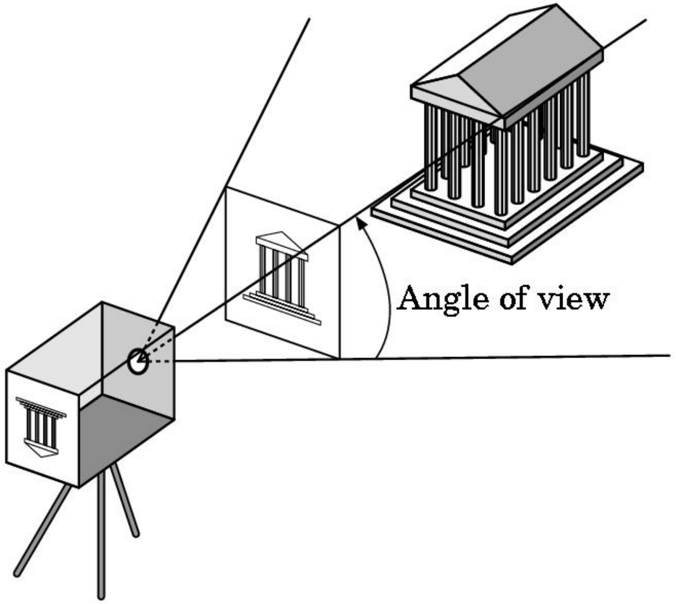
Vertices must be collected into geometric objects before clipping and rasterization can take place

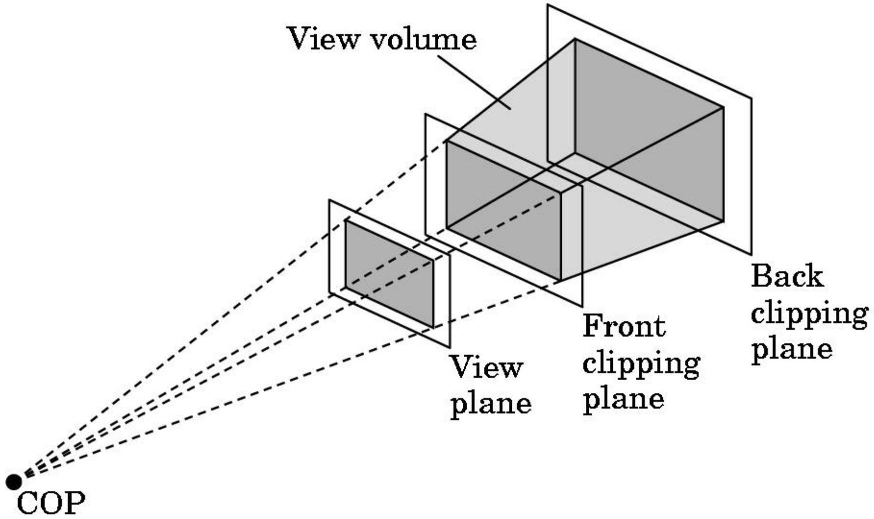
- Line segments
- Polygons
- Curves and surfaces

Clipping

Just as a real camera cannot "see" the whole world, the virtual camera can only see part of the world or object space

- Objects that are not within this volume are said to be *clipped* out of the scene





Rasterization

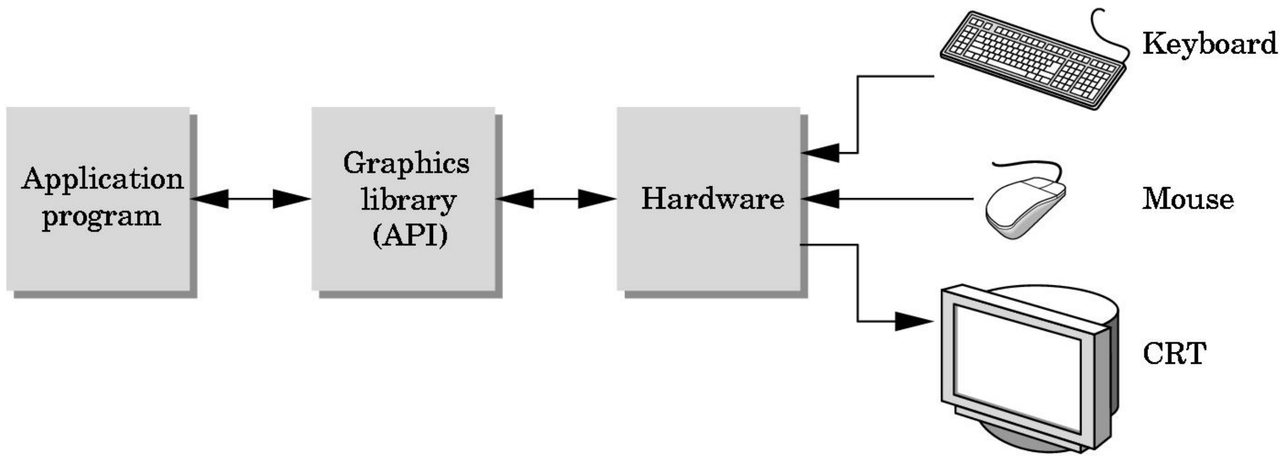
- If an object is not clipped out, the appropriate pixels in the frame buffer must be assigned colors
- Rasterizer produces a set of fragments for each object
- Fragments are “potential pixels”
  - Have a location in frame bufffer
  - Color and depth attributes
- Vertex attributes are interpolated over objects by the rasterizer

Fragment Processing

- Fragments are processed to determine the color of the corresponding pixel in the frame buffer
- Colors can be determined by texture mapping or interpolation of vertex colors
- Fragments may be blocked by other fragments closer to the camera
  - Hidden-surface removal

The Programmer's Interface

- Programmer sees the graphics system through a software interface: the Application Programmer Interface (API)



API Contents

- Functions that specify what we need to form an image
  - Objects
  - Viewer
  - Light Source(s)
  - Materials
- Other information
  - Input from devices such as mouse and keyboard
  - Capabilities of system

Object Specification

- Most APIs support a limited set of primitives including
  - Points (0D object)
  - Line segments (1D objects)
  - Polygons (2D objects)
  - Some curves and surfaces
    - Quadrics
    - Parametric polynomials
- All are defined through locations in space or *vertices*

Example (old style)

type of object

location of vertex

```
glBegin(GL_POLYGON)
  glVertex3f(0.0, 0.0, 0.0);
  glVertex3f(0.0, 1.0, 0.0);
  glVertex3f(0.0, 0.0, 1.0);
glEnd( );
```

end of object definition

Example (GPU based)

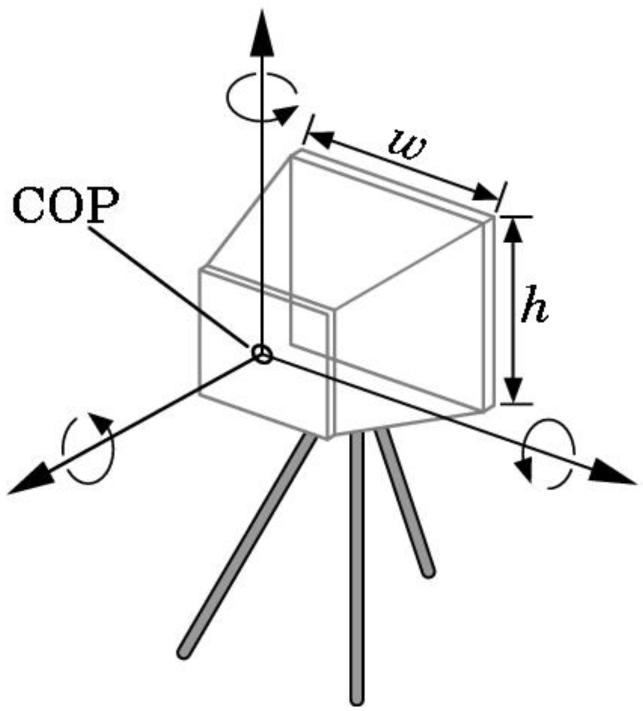
- Put geometric data in an array

```
var points = [
 vec3(0.0, 0.0, 0.0),
 vec3(0.0, 1.0, 0.0),
 vec3(0.0, 0.0, 1.0),
];
```

- Send array to GPU
- Tell GPU to render as triangle

Camera Specification

- Six degrees of freedom
  - Position of center of lens
  - Orientation
- Lens
- Film size
- Orientation of film plane



Lights and Materials

- Types of lights
  - Point sources vs distributed sources
  - Spot lights
  - Near and far sources
  - Color properties
- Material properties
  - Absorption: color properties
  - Scattering
    - Diffuse
    - Specular

Early History of APIs

- IFIPS (1973) formed two committees to come up with a standard graphics API
  - Graphical Kernel System (GKS)
    - 2D but contained good workstation model
  - Core
    - Both 2D and 3D
  - GKS adopted as IS0 and later ANSI standard (1980s)
- GKS not easily extended to 3D (GKS-3D)
  - Far behind hardware development

PHIGS and X

- Programmers Hierarchical Graphics System (PHIGS)
  - Arose from CAD community
  - Database model with retained graphics (structures)
- X Window System
  - DEC/MIT effort
  - Client-server architecture with graphics
- PEX combined the two
  - Not easy to use (all the defects of each)

SGI and GL

- Silicon Graphics (SGI) revolutionized the graphics workstation by implementing the pipeline in hardware (1982)
- To access the system, application programmers used a library called GL
- With GL, it was relatively simple to program three dimensional interactive applications

OpenGL

The success of GL lead to OpenGL (1992), a platform-independent API that was

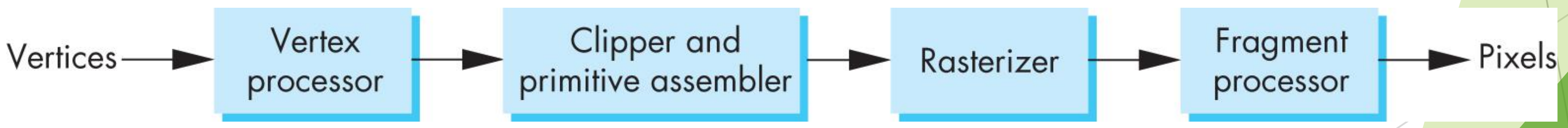
- Easy to use
- Close enough to the hardware to get excellent performance
- Focus on rendering
- Omitted windowing and input to avoid window system dependencies

OpenGL Evolution

- Originally controlled by an Architectural Review Board (ARB)
  - Members included SGI, Microsoft, Nvidia, HP, 3DLabs, IBM,…….
  - Now Kronos Group
  - Was relatively stable (through version 2.5)
    - Backward compatible
    - Evolution reflected new hardware capabilities
      - 3D texture mapping and texture objects
      - Vertex and fragment programs
  - Allows platform specific features through extensions

Modern OpenGL

- Performance is achieved by using GPU rather than CPU
- Control GPU through programs called shaders
- Application’s job is to send data to GPU
- GPU does all rendering



Immediate Mode Graphics

- Geometry specified by vertices
  - Locations in space(2 or 3 dimensional)
  - Points, lines, circles, polygons, curves, surfaces
- Immediate mode
  - Each time a vertex is specified in application, its location is sent to the GPU
  - Old style uses `glVertex`
  - Creates bottleneck between CPU and GPU
  - Removed from OpenGL 3.1 and OpenGL ES 2.0

Retained Mode Graphics

- Put all vertex attribute data in array
- Send array to GPU to be rendered immediately
- Almost OK but problem is we would have to send array over each time we need another render of it
- Better to send array over and store on GPU for multiple renderings

OpenGL 3.1

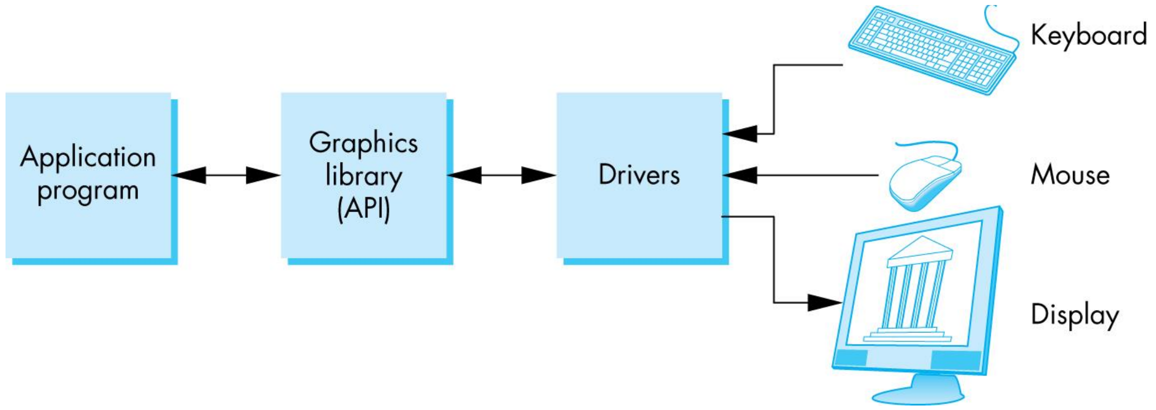
- Totally shader-based
  - No default shaders
  - Each application must provide both a vertex and a fragment shader
- No immediate mode
- Few state variables
- Most 2.5 functions deprecated
- Backward compatibility not required
  - Exists a compatibility extension

Other Versions

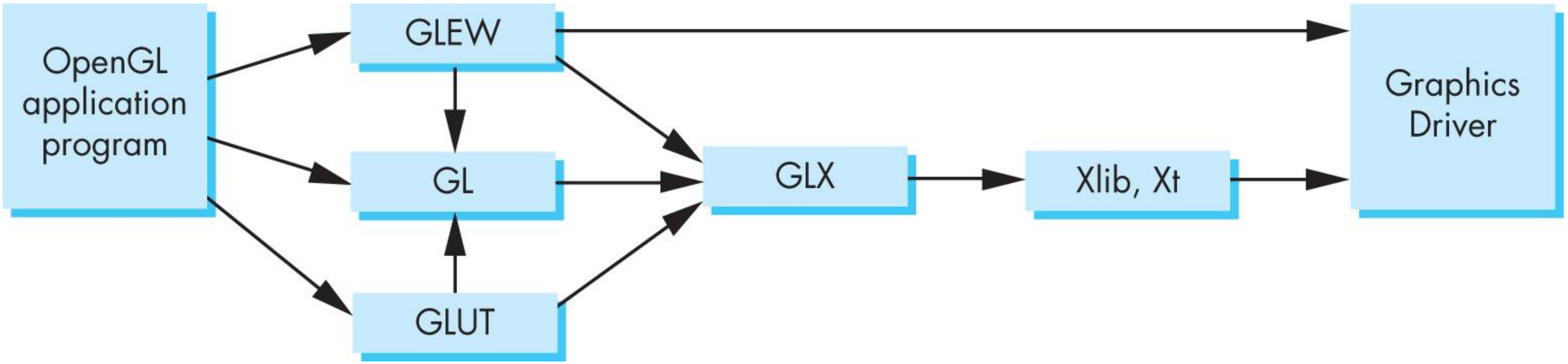
- OpenGL ES
  - Embedded systems
  - Version 1.0 simplified OpenGL 2.1
  - Version 2.0 simplified OpenGL 3.1
    - Shader based
- WebGL
  - Javascript implementation of ES 2.0
  - Supported on newer browsers
- OpenGL 4.1, 4.2, …..
  - Add geometry, tessellation, compute shaders

Graphics System

1. Processor 2. Memory 3. Frame buffer 4. Output devices 5. Input devices

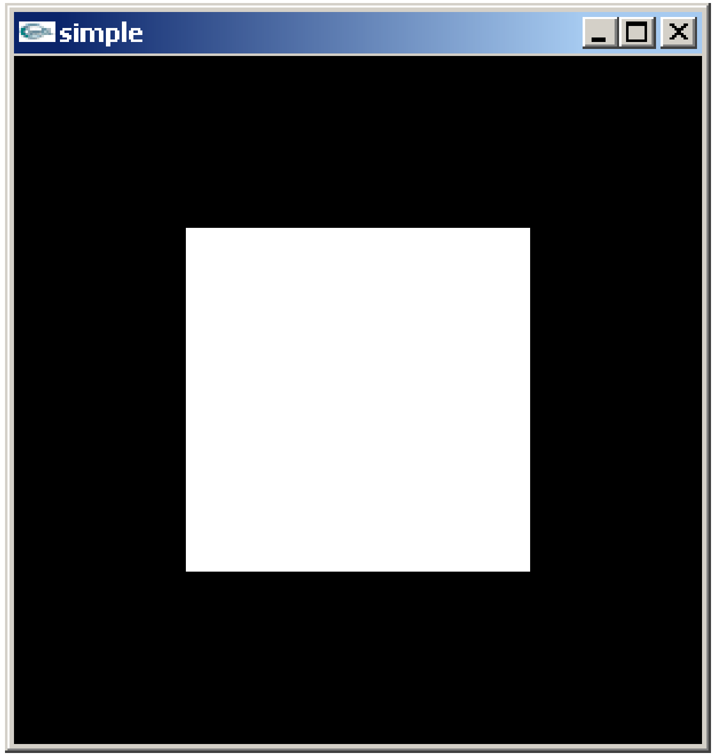


Software Organization



A OpenGL Simple Program

Generate a square on a solid background



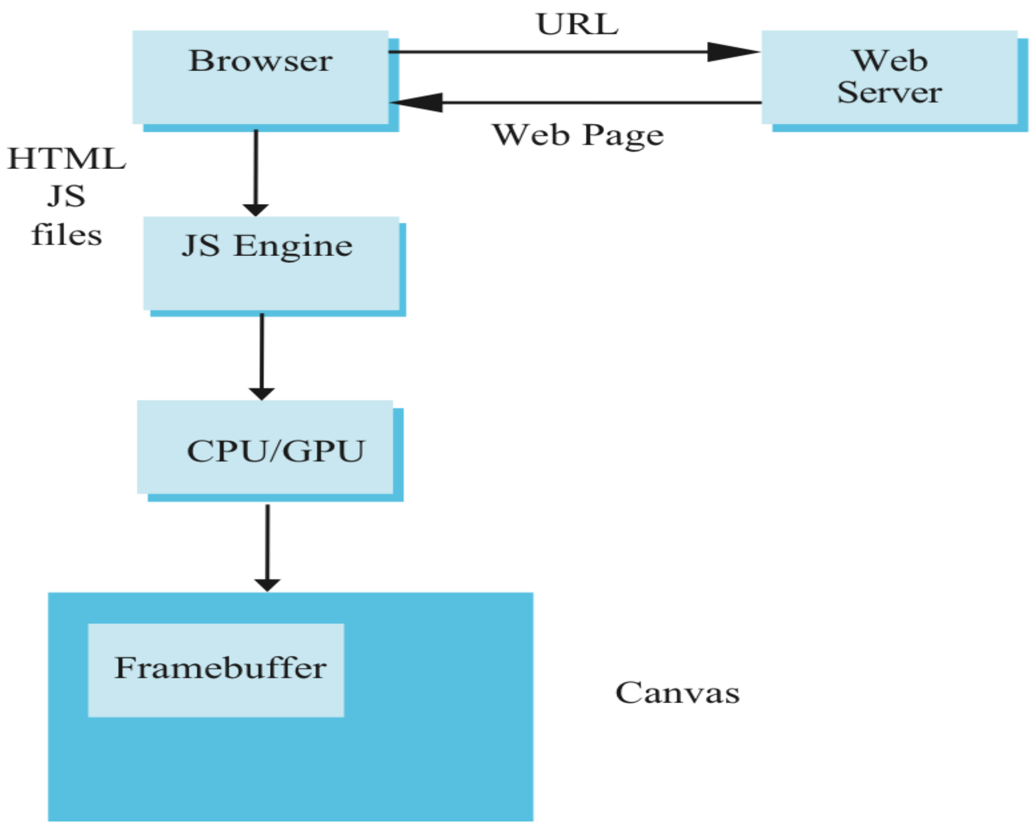
It used to be easy

```
#include <GL/glut.h>
void mydisplay(){
   glClear(GL_COLOR_BUFFER_BIT);
   glBegin(GL_QUAD;
      glVertex2f(-0.5, -0.5);
      glVertex2f(-0,5, 0,5);
      glVertex2f(0.5, 0.5);
      glVertex2f(0.5, -0.5);
   glEnd()
}
int main(int argc, char** argv){
   glutCreateWindow("simple");
   glutDisplayFunc(mydisplay);
   glutMainLoop();
}
```

What happened?

- Most OpenGL functions deprecated
  - immediate vs retained mode
  - make use of GPU
- Makes heavy use of state variable default values that no longer exist
  - Viewing
  - Colors
  - Window parameters
- However, processing loop is the same

Execution in Browser



Event Loop

- Remember that the sample program specifies a render function which is a *event listener* or *callback* function
  - Every program should have a render callback
  - For a static application we need only execute the render function once
  - In a dynamic application, the render function can call itself recursively but each redrawing of the display must be triggered by an event

Lack of Object Orientation

- All versions of OpenGL are not object oriented so that there are multiple functions for a given logical function
- Example: sending values to shaders
  - `gl.uniform3f`
  - `gl.uniform2i`
  - `gl.uniform3dv`
- Underlying storage mode is the same

WebGL function format

function name

dimension

`gl.uniform3f(x,y,z)`

belongs to WebGL canvas

`x,y,z` are variables

`gl.uniform3fv(p)`

`p` is an array

WebGL constants

- Most constants are defined in the canvas object
  - In desktop OpenGL, they were in #include files such as `gl.h`
- Examples
  - `desktop OpenGL`
    - `glEnable(GL_DEPTH_TEST);`
  - `WebGL`
    - `gl.enable(gl.DEPTH_TEST)`
  - `gl.clear(gl.COLOR_BUFFER_BIT)`

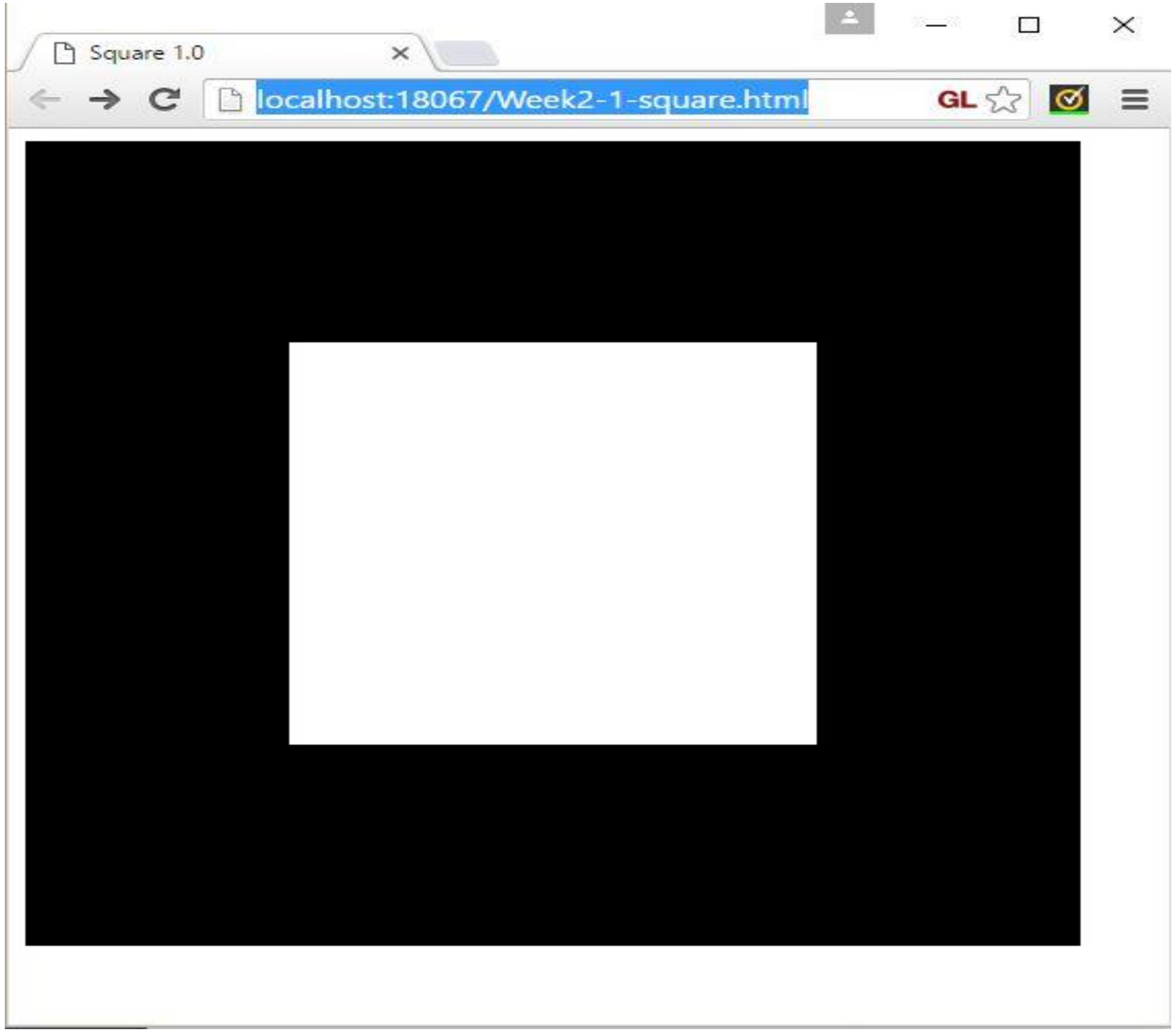
WebGL and GLSL

- WebGL requires shaders and is based less on a state machine model than a data flow model
- Most state variables, attributes and related pre 3.1 OpenGL functions have been deprecated
- Action happens in shaders
- Job of application is to get data to GPU

GLSL

- OpenGL Shading Language
- C-like with
  - Matrix and vector types (2, 3, 4 dimensional)
  - Overloaded operators
  - C++ like constructors
- Similar to Nvidia's Cg and Microsoft HLSL
- Code sent to shaders as source code
- WebGL functions compile, link and get information to shaders

Square Program



WebGL

- Five steps
  - Describe page (HTML file)
    - request WebGL Canvas
    - read in necessary files
  - Define shaders (HTML file)
    - could be done with a separate file (browser dependent)
  - Compute or specify data (JS file)
  - Send data to GPU (JS file)
  - Render data (JS file)

square.html

```
<!DOCTYPE html>
<html>
<head>
<script id="vertex-shader" type="x-shader/x-vertex">

attribute vec4 vPosition;
void main()
{
    gl_Position = vPosition;
}
</script>

<script id="fragment-shader" type="x-shader/x-fragment">

precision mediump float;

void main()
{
    gl_FragColor = vec4( 1.0, 1.0, 1.0, 1.0 );
}
</script>
```

Shaders

- We assign names to the shaders that we can use in the JS file
- These are trivial pass-through (do nothing) shaders that which set the two required built-in variables
  - gl_Position
  - gl_FragColor
- Note both shaders are full programs
- Note vector type vec2
- Must set precision in fragment shader

square.html (cont)

```
<script type="text/javascript" src="Common/webgl-utils.js"></script>
<script type="text/javascript" src="Common/initShaders.js"></script>
<script type="text/javascript" src="Common/MV.js"></script>
<script type="text/javascript" src="square.js"></script>
</head>

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

Files

- `Common/webgl-utils.js`: Standard utilities for setting up WebGL context in Common directory on website
- `Common/initShaders.js`: contains JS and WebGL code for reading, compiling and linking the shaders
- `Common/MV.js`: our matrix-vector package
- `square.js`: the application file

square.js

```
var gl;
var points;

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

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

    var vertices = [
        vec2( -0.5, -0.5 ),
        vec2(  -0.5,  0.5 ),
        vec2(  0.5, 0.5 ),
        vec2( 0.5, -0.5)
    ];
```

Notes

- `onload`: determines where to start execution when all code is loaded
- canvas gets WebGL context from HTML file
- vertices use vec2 type in MV.js
- JS array is not the same as a C or Java array
  - object with methods
  - vertices.length // 4
- Values in clip coordinates

square.js (cont)

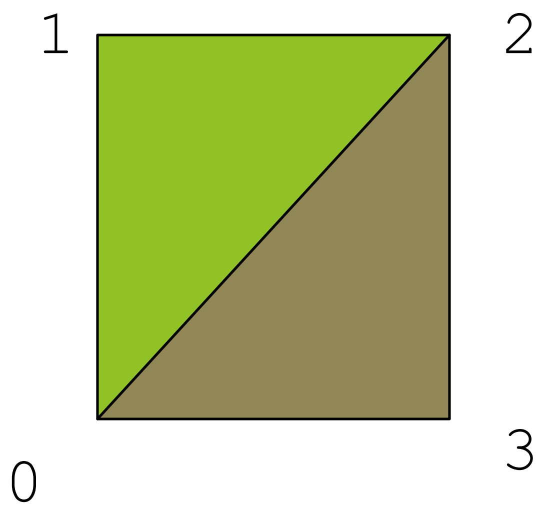
```
 //  Configure WebGL
 gl.viewport( 0, 0, canvas.width, canvas.height );
 gl.clearColor( 0.0, 0.0, 0.0, 1.0 );

  //  Load shaders and initialize attribute buffers
 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(vertices), gl.STATIC_DRAW
);

  // Associate out shader variables with our data buffer
 var vPosition = gl.getAttribLocation( program, "vPosition" );
 gl.vertexAttribPointer( vPosition, 2, gl.FLOAT, false, 0, 0 );
 gl.enableVertexAttribArray( vPosition );
```

Notes

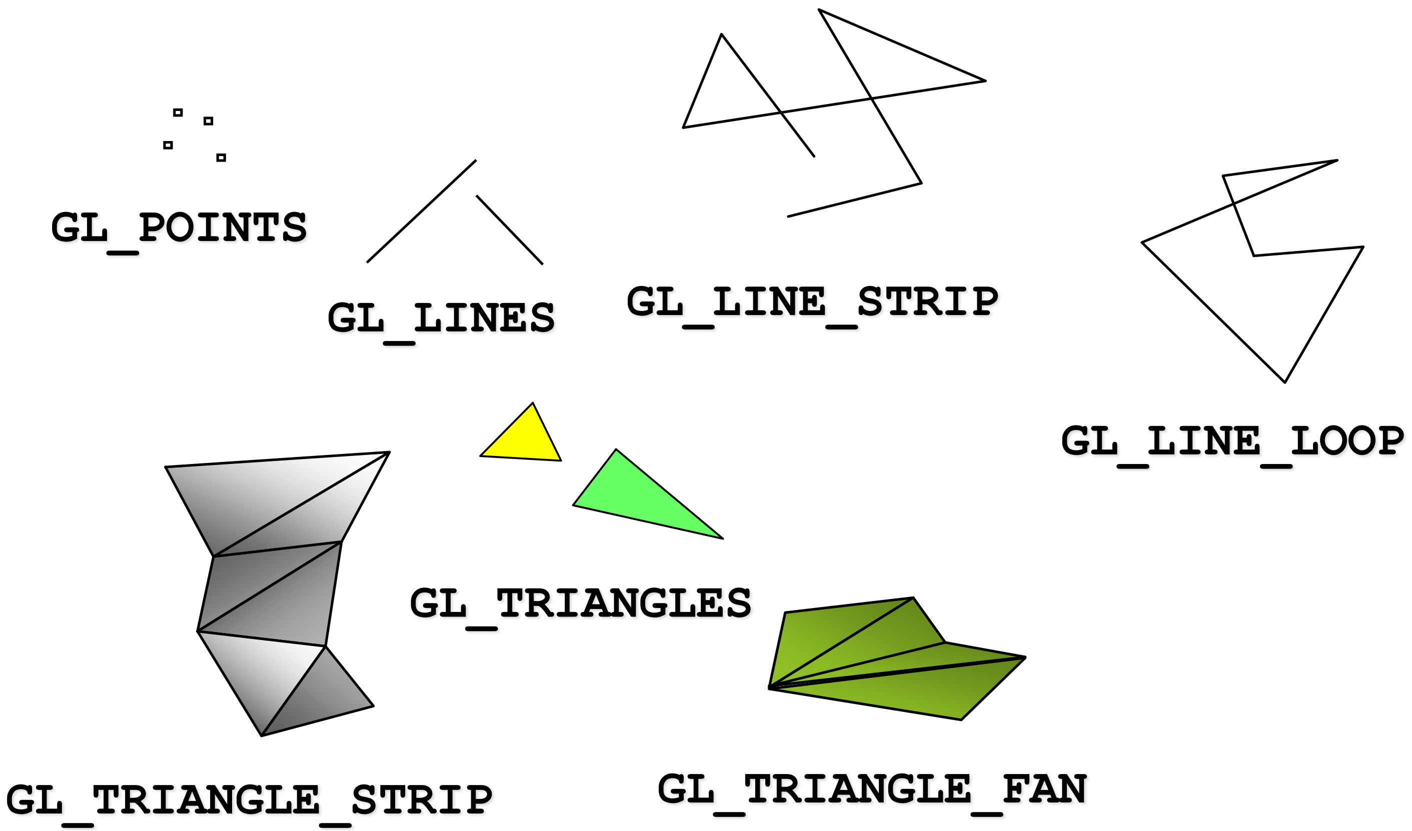
- `initShaders` used to load, compile and link shaders to form a program object
- Load data onto GPU by creating a **vertex buffer object** on the GPU
  - Note use of flatten() to convert JS array to an array of float32's
- Finally we must connect variable in program with variable in shader
  - need name, type, location in buffer

square.js (cont)

```
    render();
};

function render() {
    gl.clear( gl.COLOR_BUFFER_BIT );
    gl.drawArrays( gl.TRIANGLE_FAN, 0, 4 );
}
```



WebGLPrimitives



gl.drawArrays() method

- The prototype: void drawArrays(Glenum mode, Glint first, Glsizei count)
- mode specifies the primitive you want to render (gl.POINTS, gl.LINES, gl.LINE_LOOP, gl.LINE_STRIP, gl.TRIANGLES, gl.TRIANGLE_STRIP, gl.TRIANGLE_FAN
- first specifies which index in the array of vertex data should be used first as the first index
- count specifies how many vertices should be used.
- You need to have the vertices for the primitives that should be rendered in correct order.
- It's not good when you have an object that consists of a mesh of triangles where each vertex is often shared by several triangles.

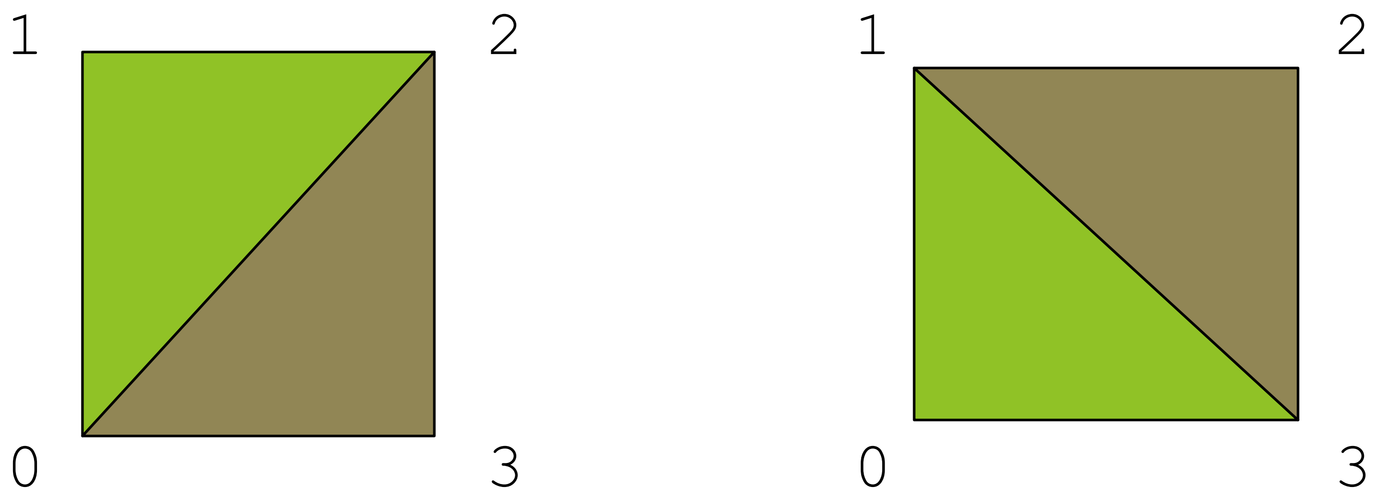
gl.drawArrays()

- Before you can call gl.drawArrays(), you must do the following:

1. Create a WebGL Buffer object with gl.createBuffer().
2. Bind WebGLBuffer object to the target gl.ARRAY_BUFFER using gl.bindBuffer()
3. Load vertex data into the buffer using gl.bufferData()
4. Enable the generic vertex attribute array using gl.enableVertexAttribArray().
5. Connect the attribute in the vertex shadder with correct data in the WebGLBuffer object by calling gl.vertexAttribPointer().

Triangles, Fans or Strips

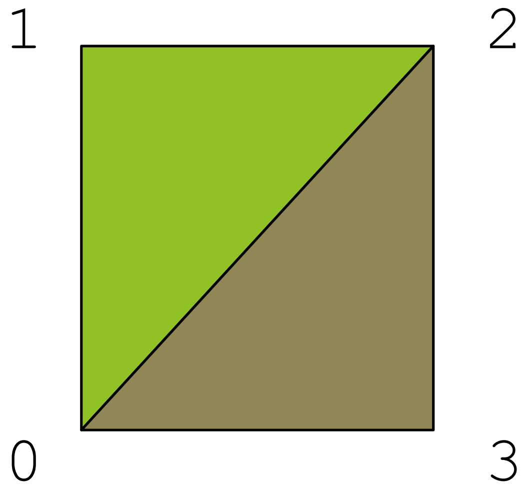
```
gl.drawArrays( gl.TRIANGLES, 0, 6 ); // v0, v1, v2, v0, v2, v3

gl.drawArrays( gl.TRIANGLE_FAN, 0, 4 ); // v0, v1 , v2, v3
```




```
gl.drawArrays( gl.TRIANGLE_STRIP, 0, 4 ); // v0, v1, v3, v2
```

gl.drawElements() method

- Sometimes referred to as index drawing
- Uses the array buffers that contain the vertex data, but in addition, it uses an “element” array buffer that contains the indices into the array buffer with vertex data.
- Vertex data can be in any order in the array buffer.
- Array buffer [V0, V1, V2, V3] Element Array Buffer [1,0,2,2,0,3]



gl.drawElements() method

- The prototype: void drawElements(Glenum mode, Glsizei count, Glenum type, Glintptr offset)
- mode specifies the primitive you want to render (gl.POINTS, gl.LINES, gl.LINE_LOOP, gl.LINE_STRIP, gl.TRIANGLES, gl.TRIANGLE_STRIP, gl.TRIANGLE_FAN
- count specifies how many indices you have in the buffer bound to the gl.ELEMENT_ARRAY_BUFFER target.
- type specifies the type for element indices that are stored in the buffer bound to gl.ELEMENT_ARRAY_BUFFER. The types you can specify are either gl.UNSIGNED_BYTE or gl.UNSIGNED_SHORT
- offset specifies the offsets into the buffer bound to gl.ELEMENT_ARRAY_BUFFER where the indices start.

gl.drawElements()

- Before you can call gl.drawElements(), you must do the following:

1. Create a WebGL Buffer object with gl.createBuffer().
2. Bind WebGLBuffer object to the target gl.ELEMENT_ARRAY_BUFFER using gl.bindBuffer()
3. Load indices thatdecide the order in which the vertex data is used into the buffer using gl.bufferData()

- You should always try to have as few calls as possible to gl.drawArrays or gl.drawElements().
- It's more efficient to have one call to gl.drawArrays or gl.drawElements() with an array that contains 200 triangles that to have 100 draw calls that each draws two triangles.

Degenerate Triangles

- If you are using gl.TRIANGLE_STRIP, you can combine different strips when there is a discontinuity.
- How? A degenerate triangle has at least two indices(or vertices) that are the same, and therefore the triangle has zero area.
- The degenerate triangles are detected by the GPU and destroyed.
- Assuming that you want to keep the same winding order:
  - The first strip consists of an even number of triangles. To connect the second strip, you need to add two extra indices.
  - The first strip consists of an odd number of triangles. To connect the second strip, you need to add three extra indices if you want to keep the winding order.

Degenerate Triangles

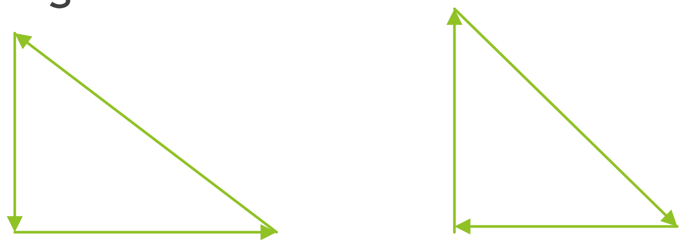
- To connect [0,1,2,3] to [4,5,6,7] in the element array buffer
- The result is [0,1,2,3,3,4,4,5,6,7]
- Generated triangles
  - [0,1,2]
  - [2,1,3]
  - [2,3,3] ➔ degenerate
  - [3,3,4] ➔ degenerate
  - [3,4,4] ➔ degenerate
  - [4,4,5] ➔ degenerate
  - [4,5,6]
  - [6,5,7]

Degenerate triangles

- To connect [0,1,2,3,4] to [5,6,7,8] in the element array buffer
- The result is [0,1,2,3,4,4,4,5,5,6,7,8]
- Generated triangles
  - [0,1,2]
  - [2,1,3]
  - [2,3,4]
  - [4,3,4] ➔ degenerate
  - [4,4,4] ➔ degenerate
  - [4,4,5] ➔ degenerate
  - [4,5,5] ➔ degenerate
  - [5,5,6] ➔ degenerate
  - [5,6,7]
  - [7,6,8]

Winding Order

- An important property for a triangle in WebGL is called the winding order.
- CCW(Counter clockwise) winding order occurs when vertices build up the triangle in CCW order. Clockwise (CW) winding order occurs when the vertices buildup the triangle in clockwise.
- The winding order decides whether or not triangles face the viewer.
- Triangles face the viewer are called front-facing and triangles that are not facing the viewer called back-facing.



Culling

- In many cases, there is no need for WebGL to rasterize triangles that are back-facing.
- You can tell WebGL to cull the faces that cannot be seen.
  - gl.frontFace(gl.CCW) tells WebGL that triangles with CCW winding are front facing
  - gl.enable(gl.CULL_FACE) enables the culling for faces
  - gl.cullFace(gl.BACK) cull the back-facing triangles
- gl.frontFace(gl.CW) tells WebGL that triangles with CW are front-facing
- gl.cullFace(gl.Front) cull the front-facing triangles
- If you have a scene that consists of objects whose back side the users cannot see, it's a good idea to enable back-facing culling. This can increase performance since the GPU will not need to rasterize triangles that are not visible.

Program Execution

- WebGL runs within the browser
  - complex interaction among the operating system, the window system, the browser and your code (HTML and JS)
- Simple model
  - Start with HTML file
  - files read in asynchronously
  - start with onload function
    - event driven input

Coordinate Systems

- The units in `points` are determined by the application and are called *object*, *world*, *model* or *problem coordinates*
- Viewing specifications usually are also in object coordinates
- Eventually pixels will be produced in *window coordinates*
- WebGL also uses some internal representations that usually are not visible to the application but are important in the shaders
- Most important is *clip coordinates*

Coordinate Systems and Shaders

- Vertex shader must output in clip coordinates
- Input to fragment shader from rasterizer is in window coordinates
- Application can provide vertex data in any coordinate system but shader must eventually produce gl_Position in clip coordinates
- Simple example uses clip coordinates

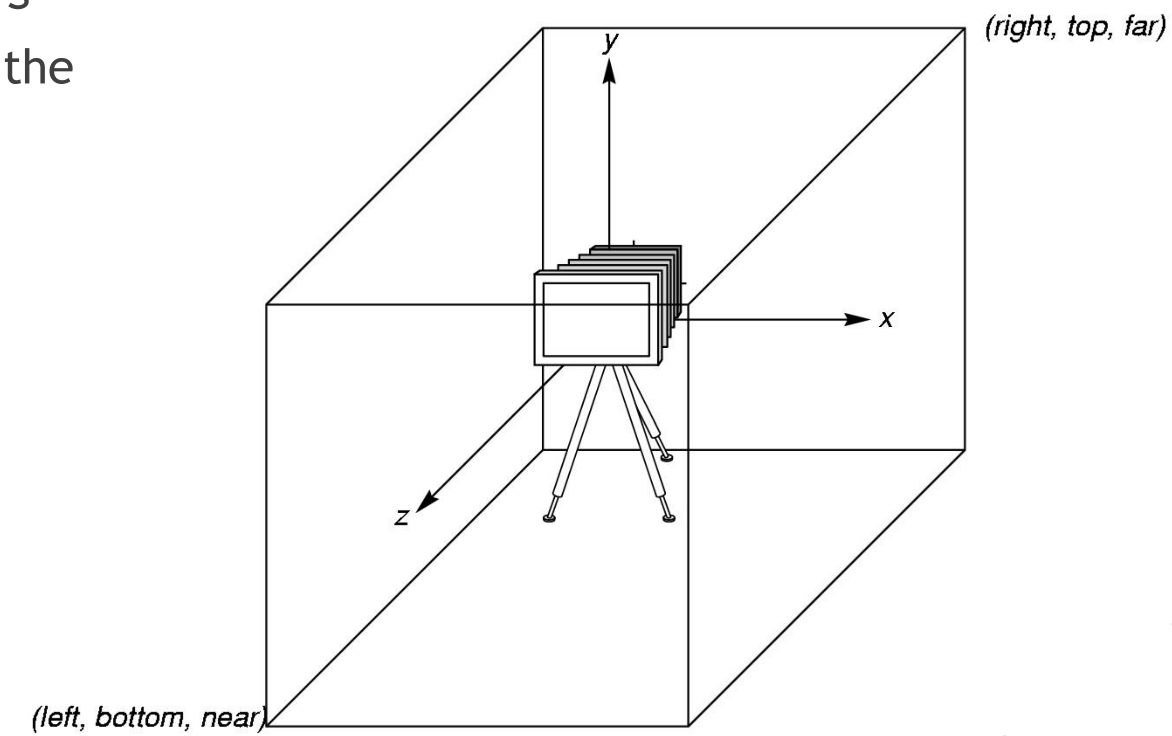
WebGL Camera

- WebGL places a camera at the origin in object space pointing in the negative z direction
- The default viewing volume

  is a box centered at the

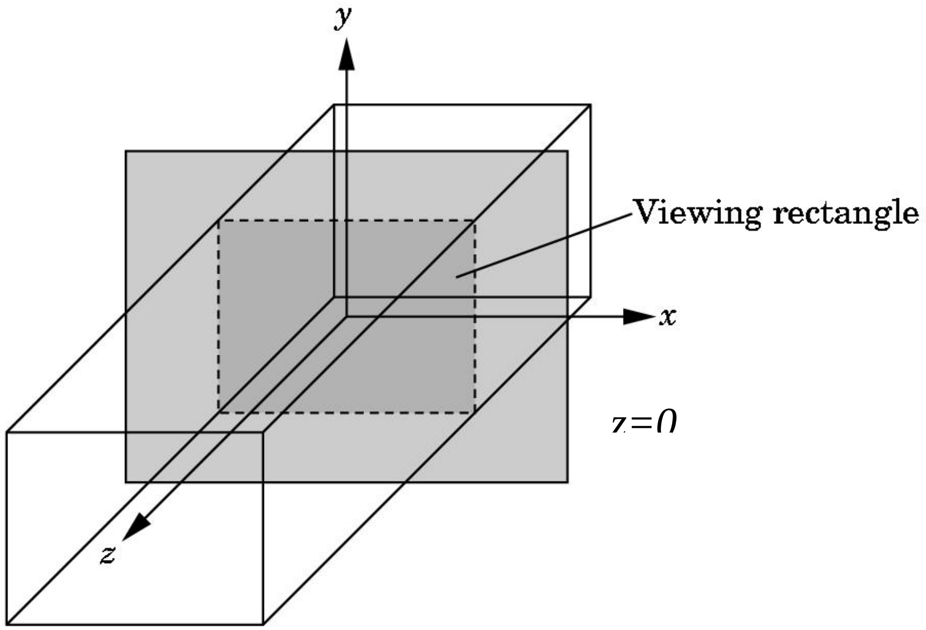
  origin with sides of

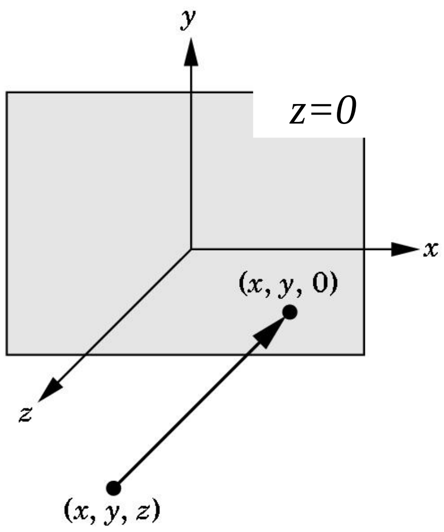
  length 2



Orthographic Viewing

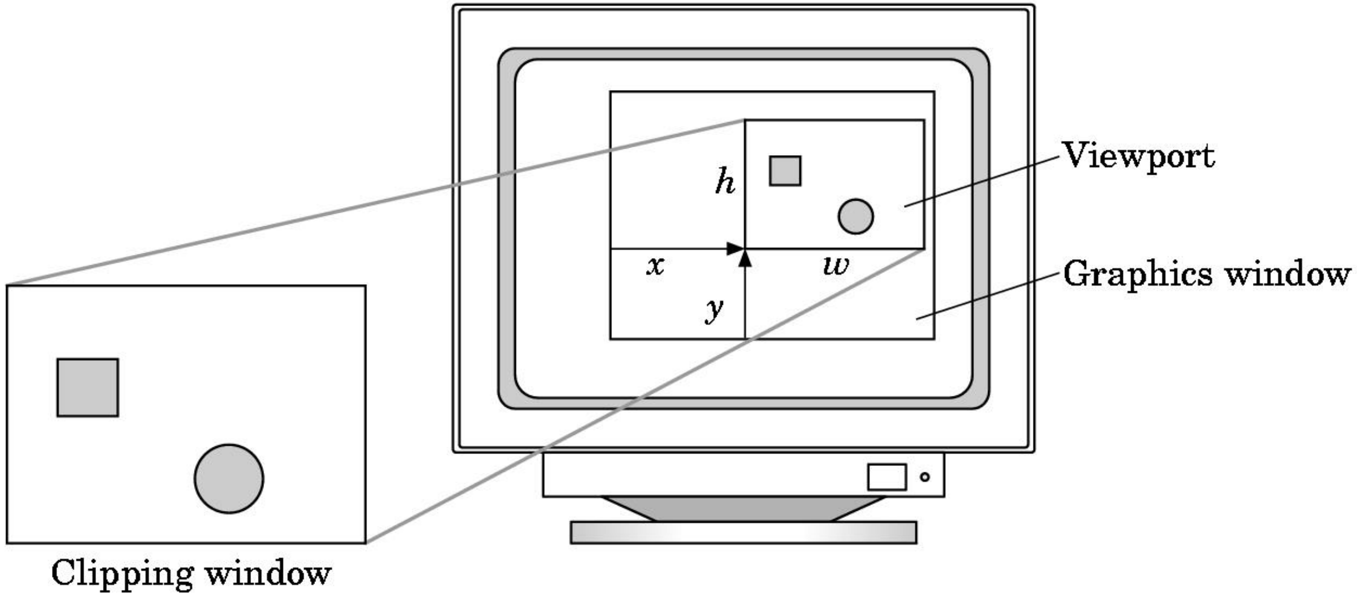
In the default orthographic view, points are projected forward along the z axis onto the plane $z=0$





Viewports

- Do not have use the entire window for the image: `gl.viewport(x,y,w,h)`
- Values in pixels (window coordinates)



Transformations and Viewing

- In WebGL, we usually carry out projection using a projection matrix (transformation) before rasterization
- Transformation functions are also used for changes in coordinate systems
- Pre 3.1 OpenGL had a set of transformation functions which have been deprecated
- Three choices in WebGL
  - Application code
  - GLSL functions
  - MV.js

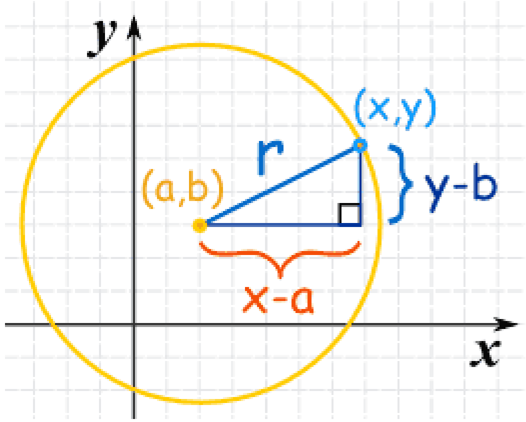
Circle

- All points are the same distance from the center
- If the center is (a,b), the circle is all the points that are “r” (radius) away from the center (a,b)
- We make a right-angled triangle (as shown),
- and then use Pythagoras ($a^2 + b^2 = c^2$):
- $(x-a)^2 + (y-b)^2 = r^2$
- The equation can be written in parametric form using

  the trigonometric functions sine and cosine as

  $x = a+r\cos(t)$

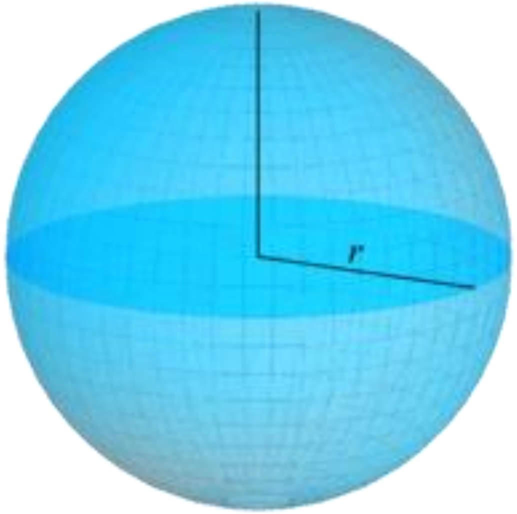
  $y = b+r\sin(t)$



- Unit Circle: If we place the circle centre at
- and set the radius to 1 we get: $x^2 + y^2 = 1$

Sphere

- Sphere is a perfectly round geometrical object in three dimensional space.
- If the center is (a,b,c), the circle is all the points that are “r” (radius) away from the center (a,b,c)
- $(x-a)^2 + (y-b)^2 + (z-c)^2 = r^2$



$x = a + r*\cos(\theta)*\sin(\Phi)$

$y = b + r*\sin(\theta)*\sin(\Phi)$

$z = c + r*\cos(\Phi)$

$(0<=\theta<=2\prod \text{ and } 0<=\Phi<= \prod)$

Tessellation and Twist

- Consider rotating a 2D point about the origin

$$x' = x\cos\theta - y\sin\theta$$

$$y' = x\sin\theta + y\cos\theta$$

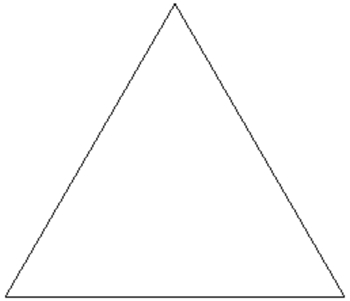
- Now let amount of rotation depend on distance from origin giving us **twist**

$$x' = x\cos(d\theta) - y\sin(d\theta)$$

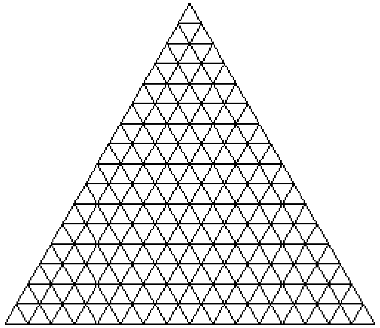
$$y' = x\sin(d\theta) + y\cos(d\theta)$$

$$d \propto \sqrt{x^2 + y^2}$$

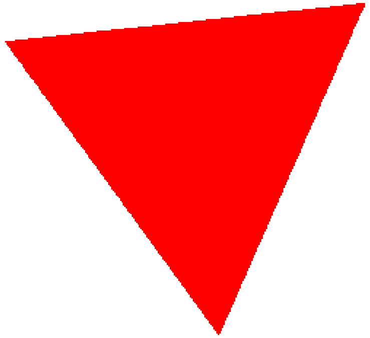
Example



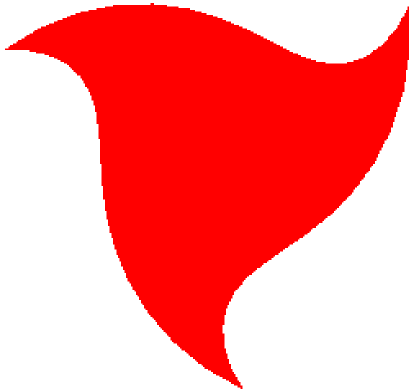
triangle



tessellated triangle



twist without tessellation



twist after tessellation

Setting up the buffer for interleaved vertex data

- Stride is the number of bytes between the start of 2 vertex elements :16 Bytes
- The WebGLBuffer will have the following structure:
- The first three elements occupy 12 bytes in total, which is the offset from the start of the buffer to the first color component.

x	y	z	R	G	B	A	x	y	z	R	G	B	A	…
4 Bytes	4 Bytes	4 Bytes	1 Byte	1 Byte	1 Byte	1 Byte								
Position Vertex 0			Color Vertex 0				Position Vertex 1			Color Vertex 1				

- How the positions are organized in the vertex array?

```
gl.vertexAttribPointer(vertexPositionAttribute,positionSize, gl.FLOAT, false, 16, 0);
```

- How colors are organized in the vertex array

```
gl.vertexAttribPointer(vertexColorAttribute,colorSize, gl.UNSIGNED_BYTE, true, 16, 12);
```

Interleaved Vertex Data