Basic WebGL Graphics

Sorin Babii
sorin.babii@cs.upt.ro

Models and Architectures

Objectives

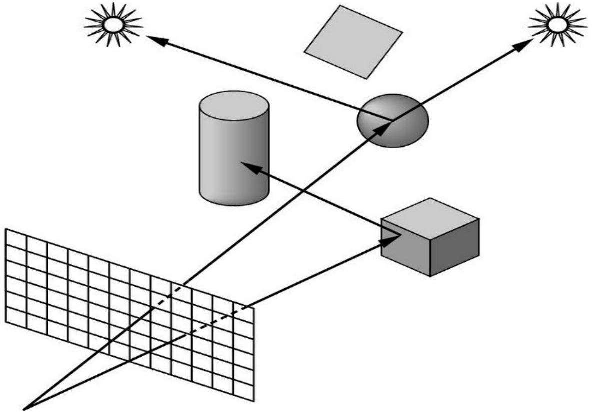
- the basic design of a graphics system
- pipeline architecture
- software components for an interactive graphics system

Image Formation

- Can we mimic the synthetic camera model?
- Application Programmer Interface (API)
- Need only specify
    - Objects
    - Materials
    - Viewer
    - Lights
- How is the API implemented?

Physical Approaches

- **Ray tracing**: follow rays from center of projection
  - Can handle global effects
    - Multiple reflections
    - Translucent objects
  - Slow
  - Must have whole data base available at all times



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

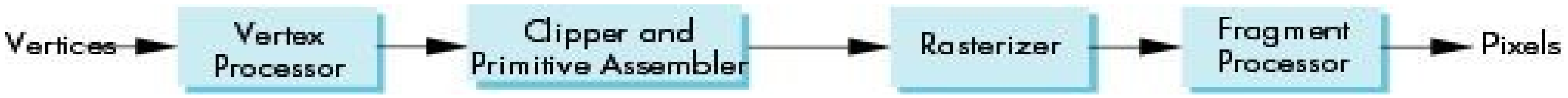
Practical Approach

- Process objects one at a time in the order they are generated by the application
  - Can consider only local lighting

- 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* - 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

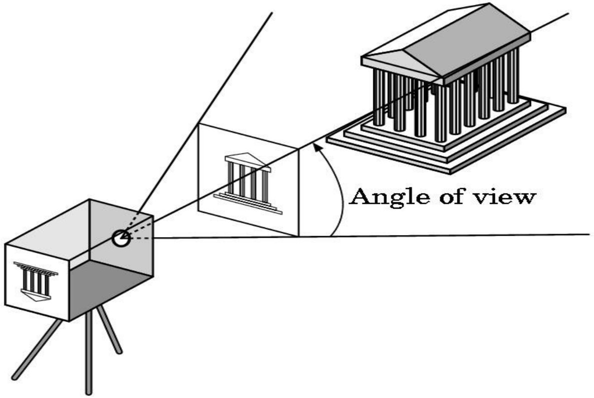
Vertices must be collected into *geometric objects* before clipping and rasterization can take place

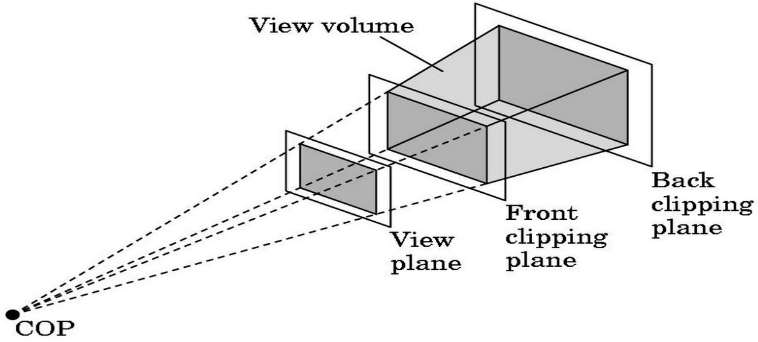
- -Line segments
- -Polygons
- -Curves and surfaces

Clipping

A real camera cannot “see” the whole world → the virtual camera can only see part of the world or object space

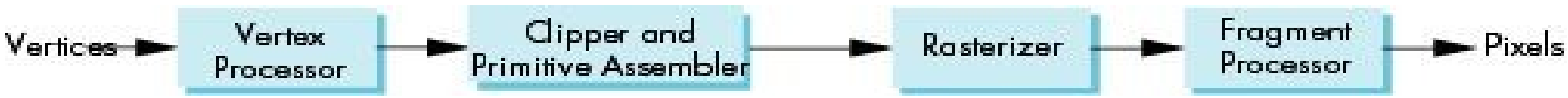
- -Objects are *clipped* out of the scene when they are not within this volume





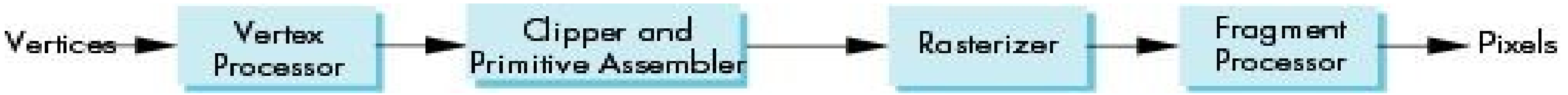
Rasterization

- 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 buffer
  - Color and depth attributes
- Vertex attributes are interpolated over objects by the rasterizer



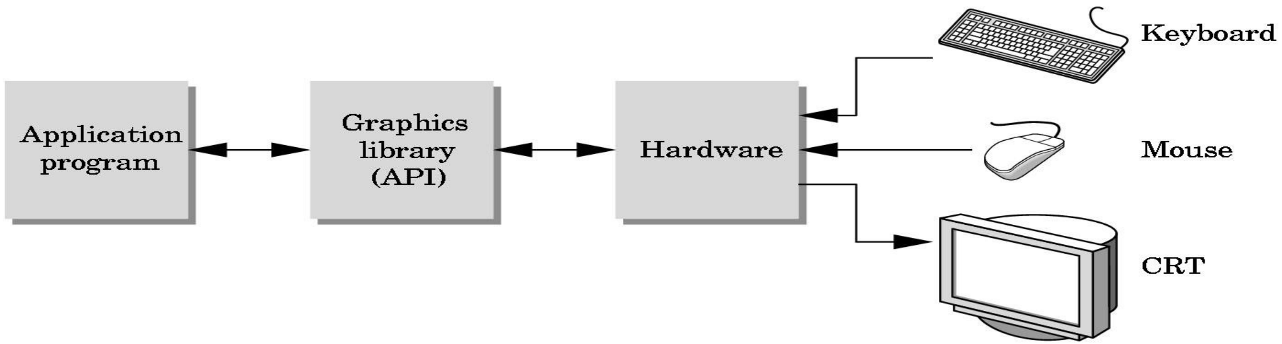
Fragment Processing

- 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)

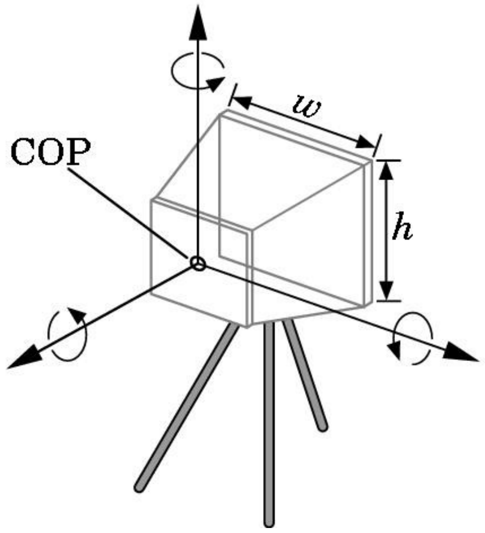
- Put geometric data in an array

```
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

Programming with WebGL

Part 1: Background

Objectives

- Development of the OpenGL API
- OpenGL Architecture
    - OpenGL as a state machine
    - OpenGL as a data flow machine
- Functions
    - Types
    - Formats
- Simple program

Early History

- International Federation of Information Processing Societies (IFIPS): 1973 → 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 ISO 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)
  - -CAD community experience
  - -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 → OpenGL (1992), a platform-independent API:

- Easy to use
- Close enough to the hardware to achieve good 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 D)
  - 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

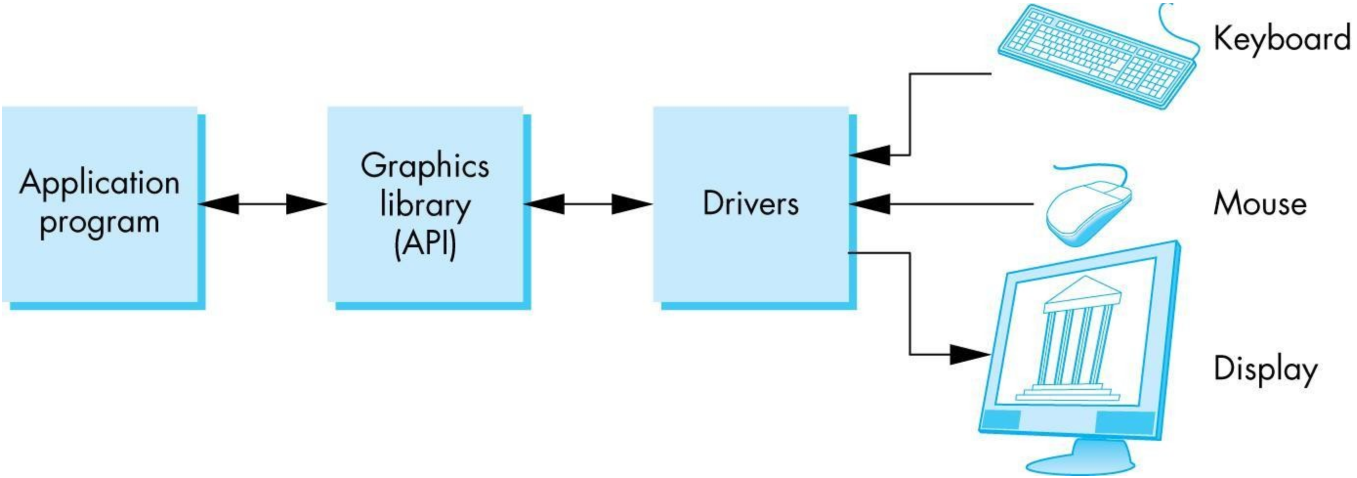
OpenGL 3.1

- 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

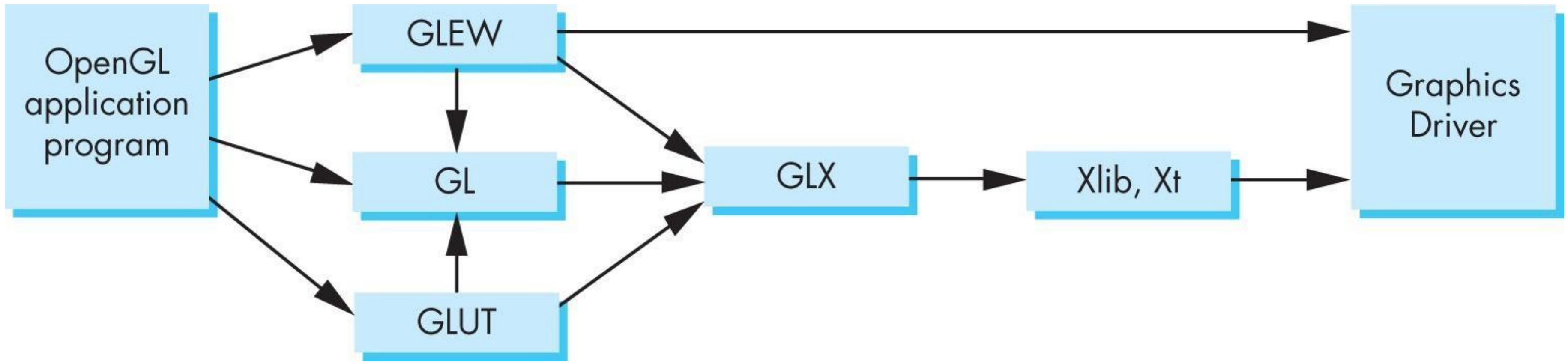
Other Versions

- 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

OpenGL Architecture

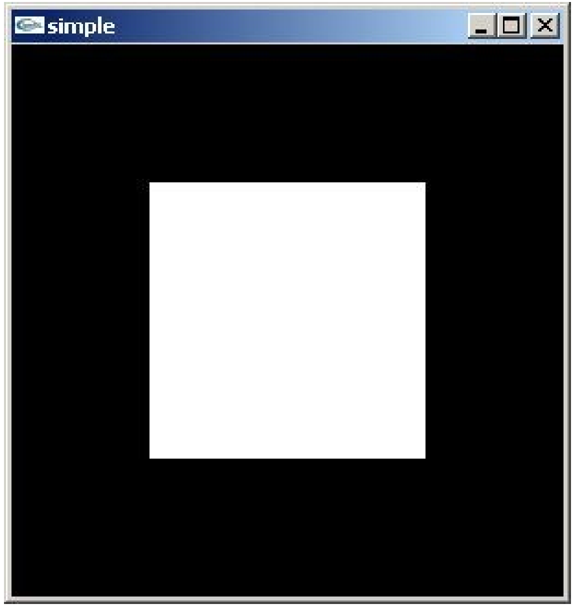


Software Organization



A OpenGL Simple Program

A square on a solid background



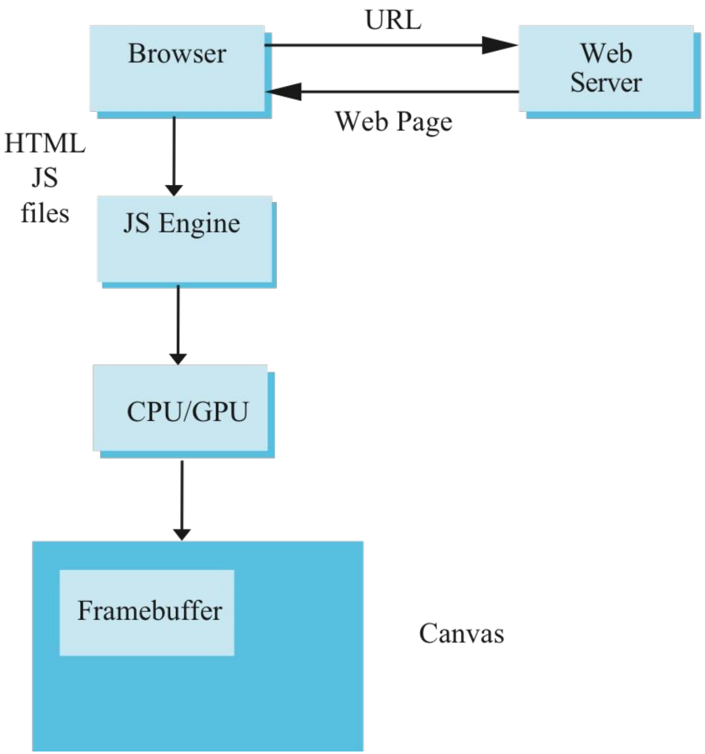
How?

```
#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();
}
```

Evolution

- 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

- The sample program specifies a render function, which is an *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

Missing 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 of type float

`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: `gl.h`
- Examples
  - `desktop OpenGL`
    - `glEnable(GL_DEPTH_TEST);`
  - `WebGL`
    - `gl.enable(gl.DEPTH_TEST)`
  - `gl.clear(gl.COLOR_BUFFER_BIT)`

WebGL and GLSL

- GLSL = OpenGL Shading Language
- WebGL *requires shaders* and is based more on the data flow model, less on the state machine model
- Most state variables, attributes and related pre 3.1 OpenGL functions have been deprecated
- Action happens in shaders
- Job of application: get data to GPU

GLSL

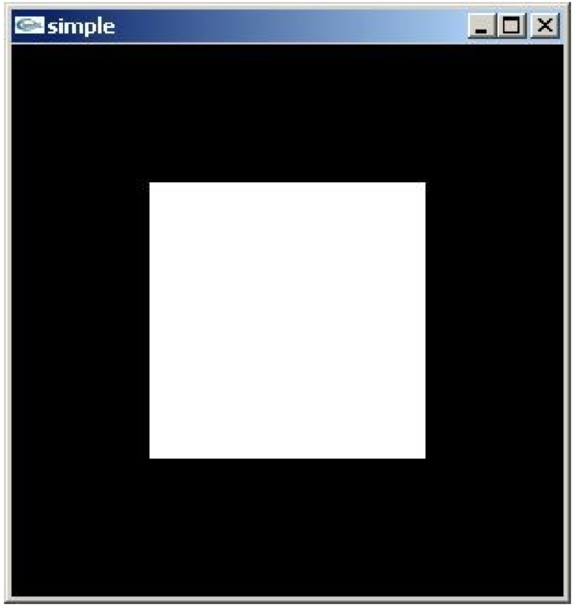
- C-like with
  - -Matrix and vector types (2, 3, 4 dimensional)
  - -Overloaded operators
  - -C++ like constructors
- Similar to Nvidia's Cg and Microsoft HLSL
- Code sent to shaders as source code
- WebGL functions compile, link and get information to shaders

Complete Programs

Objectives

- Build a complete first program
  - Introduce shaders
  - Introduce a standard program structure
- Simple viewing
  - Two-dimensional viewing as a special case of three-dimensional viewing
- Initialization steps and program structure

Remember the 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 will 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

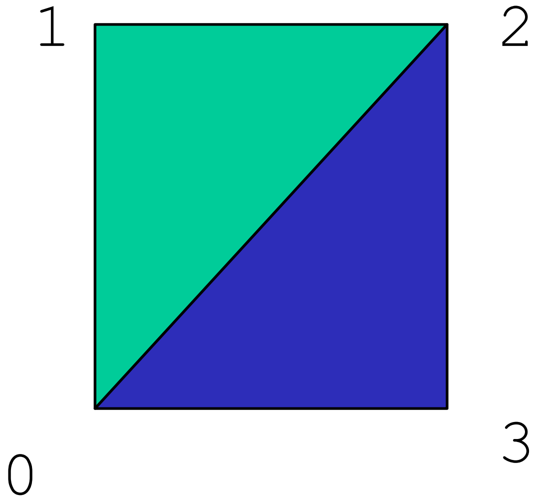
- `initShaders` used to load, compile and link shaders to form a program object
- Load data onto GPU by creating a **vertex buffer object** on the GPU

  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

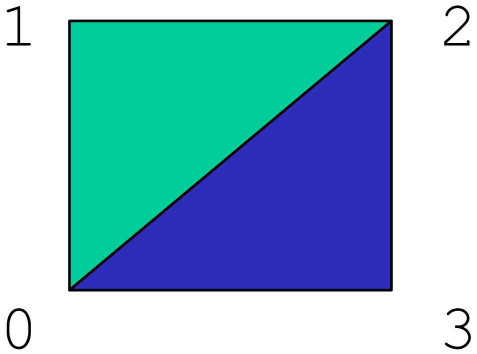
square.js (cont)

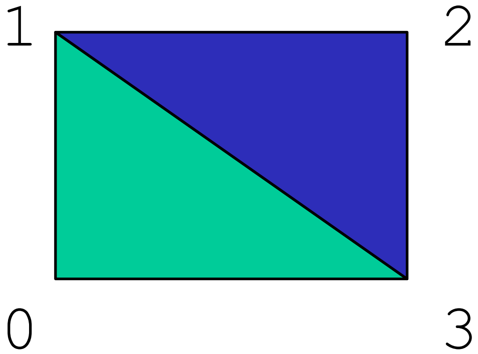
```
    render();
};

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



Triangles, Fans or Strips

```
gl.drawArrays( gl.TRIANGLES, 0, 6 ); // 0, 1, 2, 0, 2, 3
gl.drawArrays( gl.TRIANGLE_FAN, 0, 4 ); // 0, 1 , 2, 3
```






```
gl.drawArrays( gl.TRIANGLE_STRIP, 0, 4 ); // 0, 1, 3, 2
```

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

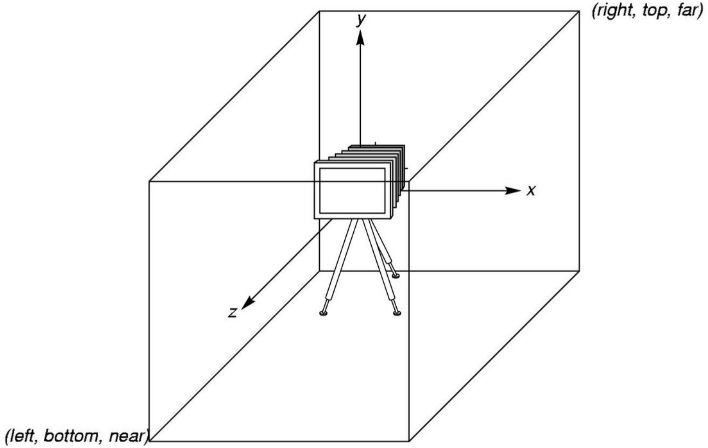
- The units in `points` are determined by the application and are called *object, world, model* or *problem coordinates*
- Viewing specifications usually are also in object coordinates
- Eventually pixels will be produced in *window coordinates*
- WebGL also uses some internal representations that usually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

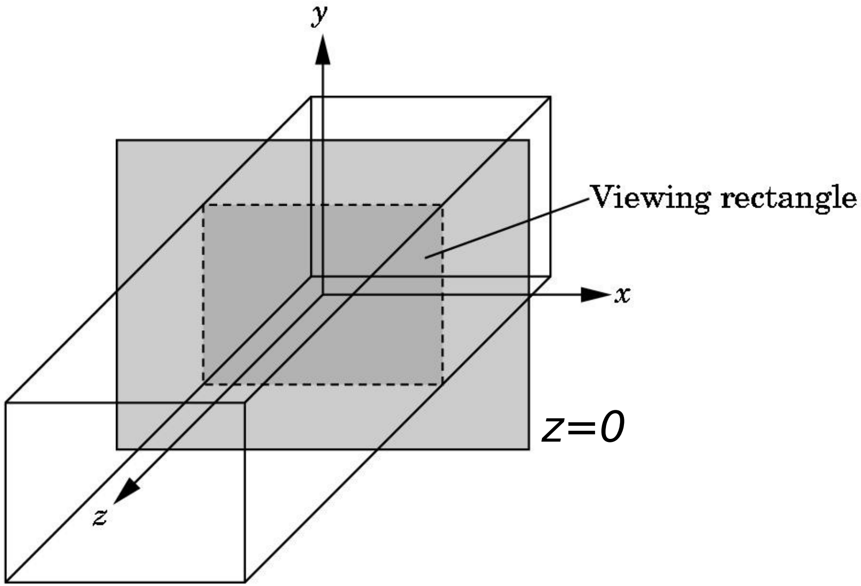
WebGL Camera

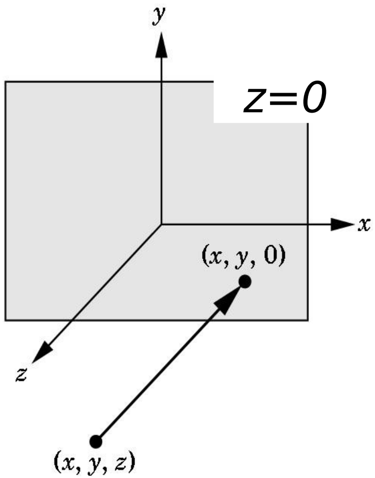
- WebGL places a camera at the origin in object space pointing in the negative *z* direction
- The default viewing volume is a box centered at the origin with sides of length 2



Orthographic Viewing

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



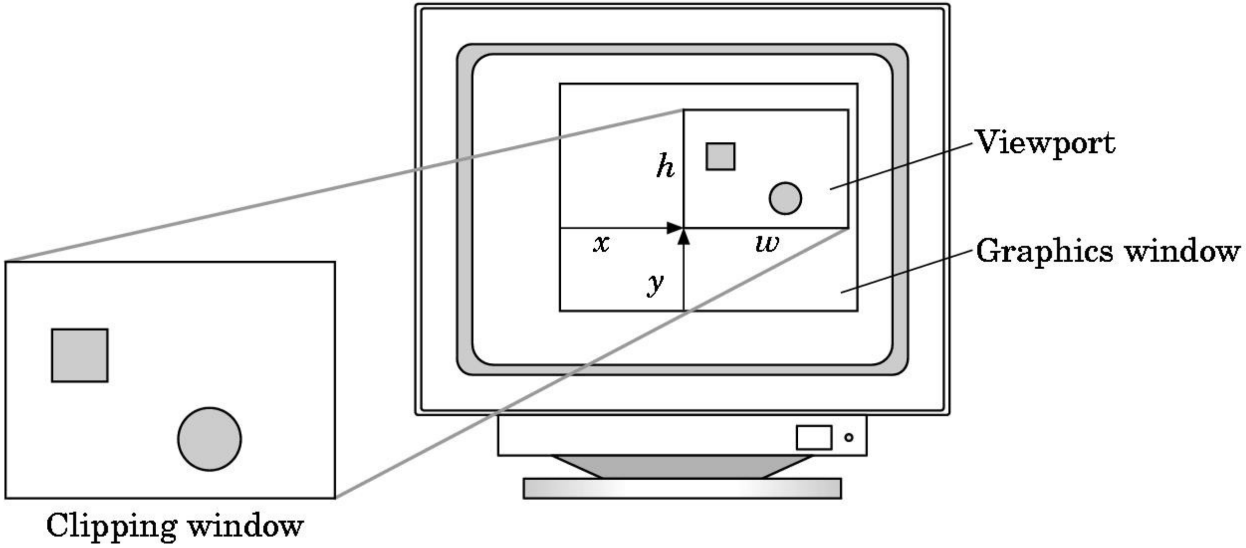


Viewports

- Do not have to 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