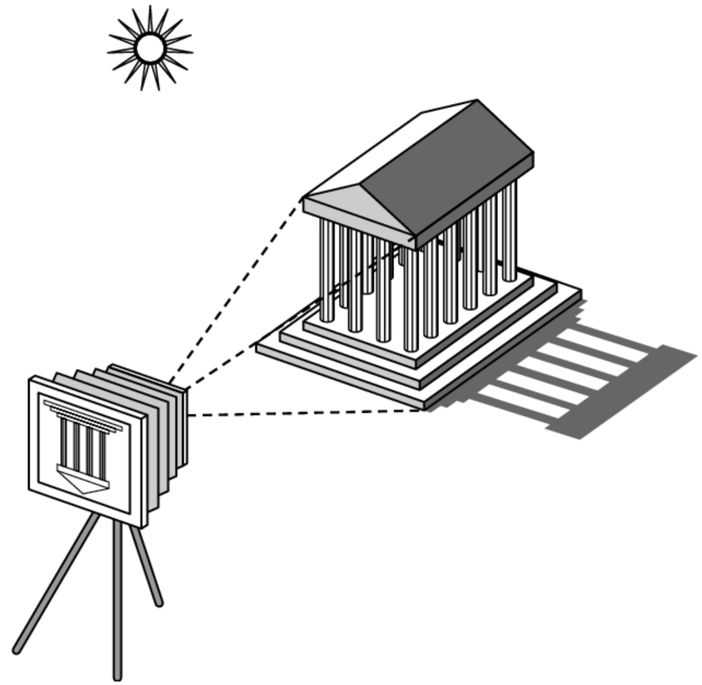
Learning Objectives

- Students completing this lecture will be able to
  - Describe each stage of the graphics pipeline (vertex processor, clipper and primitive assembler, rasterizer, fragment processor) and explain their relationships
  - Explain the key elements for image formation
  - Describe object/model, world, camera/eye, and screen coordinates
  - Differentiate immediate and retained mode graphics, flat and smooth shading

Graphics Pipeline

Elements of Image Formation

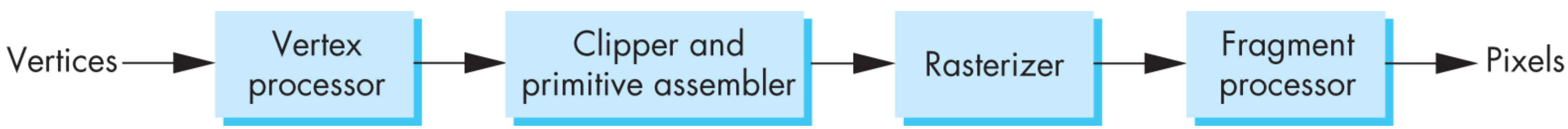
- Objects
- Viewer
- Light source(s)



- Need to consider materials that govern how light interacts with the objects in the scene
- Note the independence of the objects, the viewer, and the light source(s)

Graphics Pipeline

- Process objects one at a time in the order they are generated by the application
- Pipeline architecture



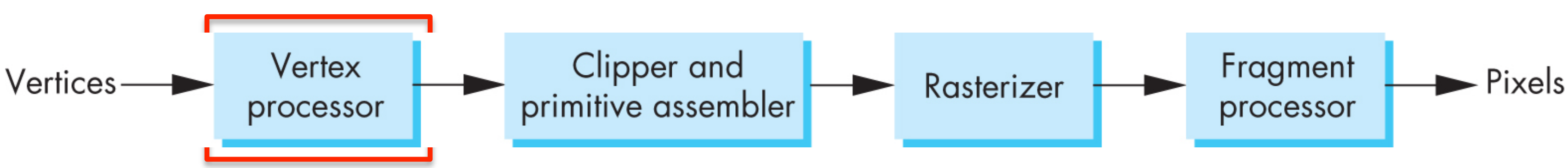
application program

display

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

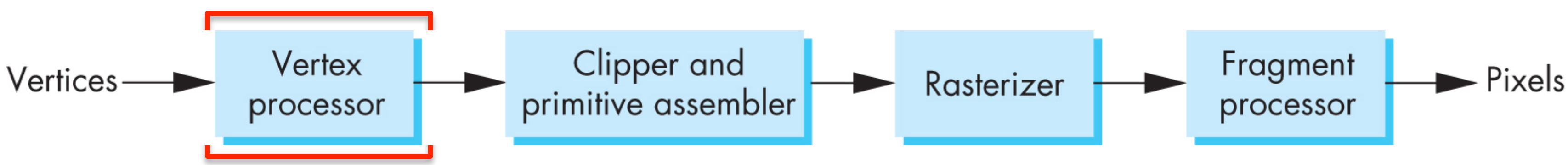
Vertex Processing

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



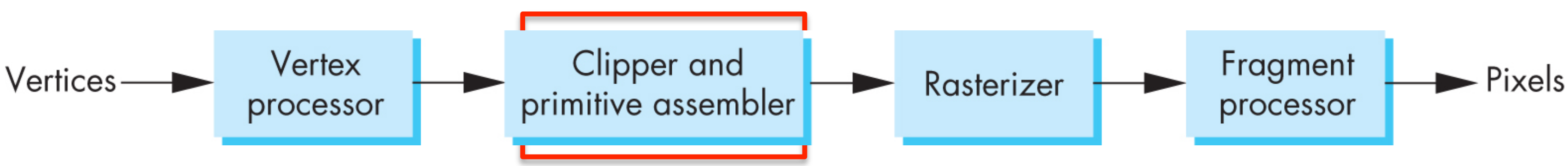
Projection

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



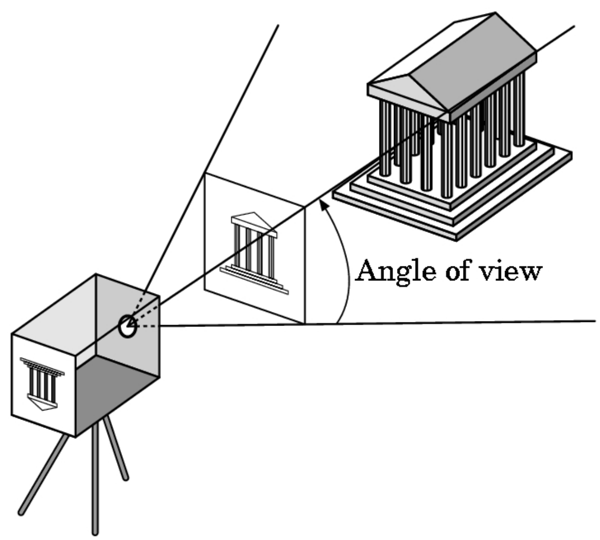
Primitive Assembly

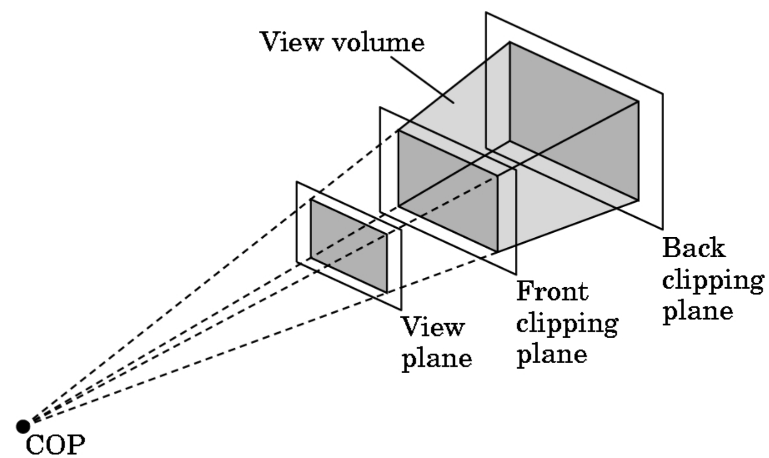
- Vertices must be collected into geometric objects before clipping and rasterization can take place
  - Line segments
  - Polygons
  - Curves and surfaces



Clipping

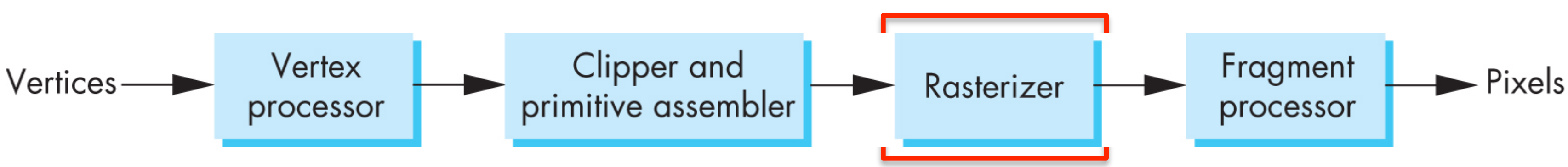
- Just as a real camera cannot "see" the whole world, the virtual camera can only see part of the world or object space
  - Objects that are not within this volume are said to be clipped out of the scene





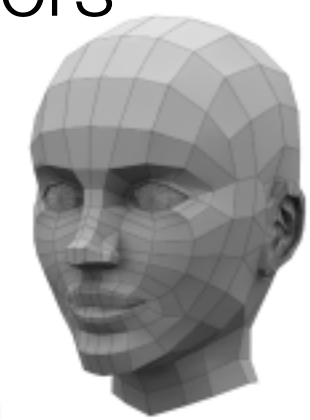
Rasterization

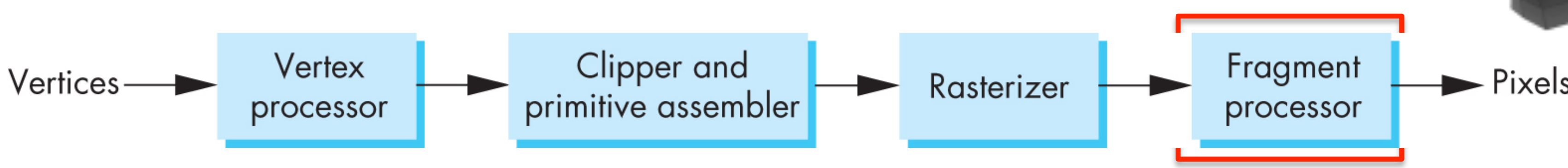
- If an object is not clipped out, the appropriate pixels in the frame buffer must be assigned colors
- Rasterizer produces a set of fragments for each object
- Fragments are "potential pixels"
  - Have a location in frame buffer
  - Color and depth attributes
- Vertex attributes (such as colors) are 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

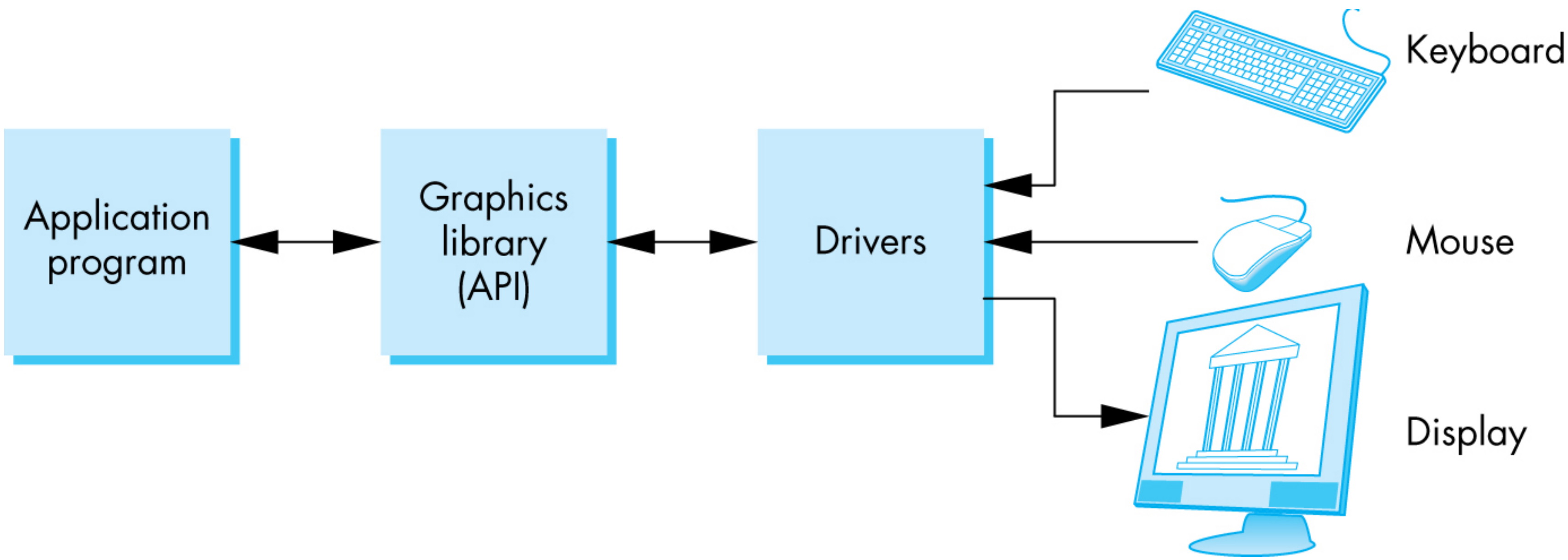




Programmer’s Interface

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 OpenGL)

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 WebGL)

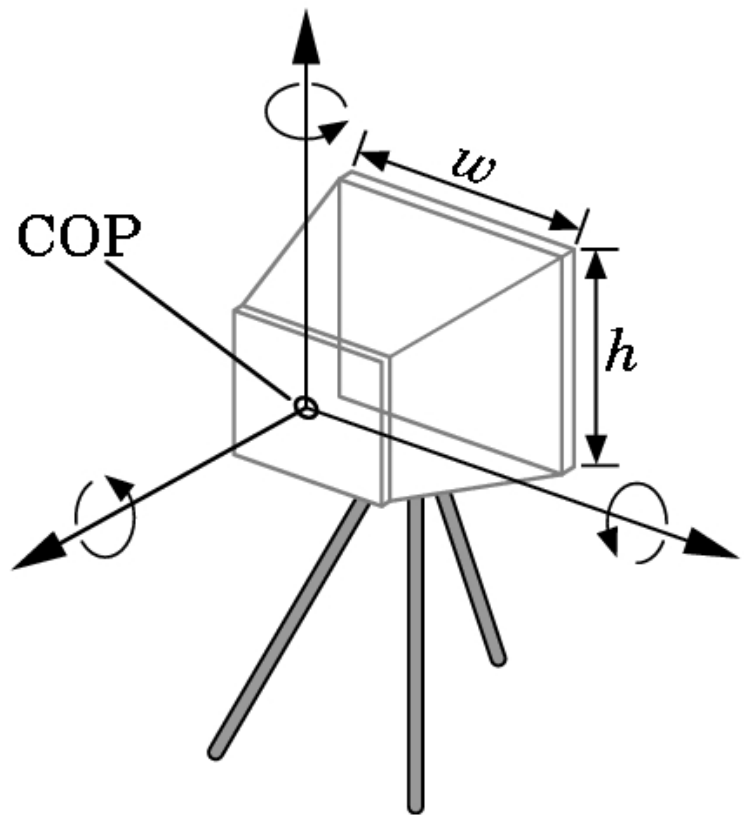
- Put geometric data in an array

```
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 (three angles)
- Lens
- Film size
- Orientation of film plane



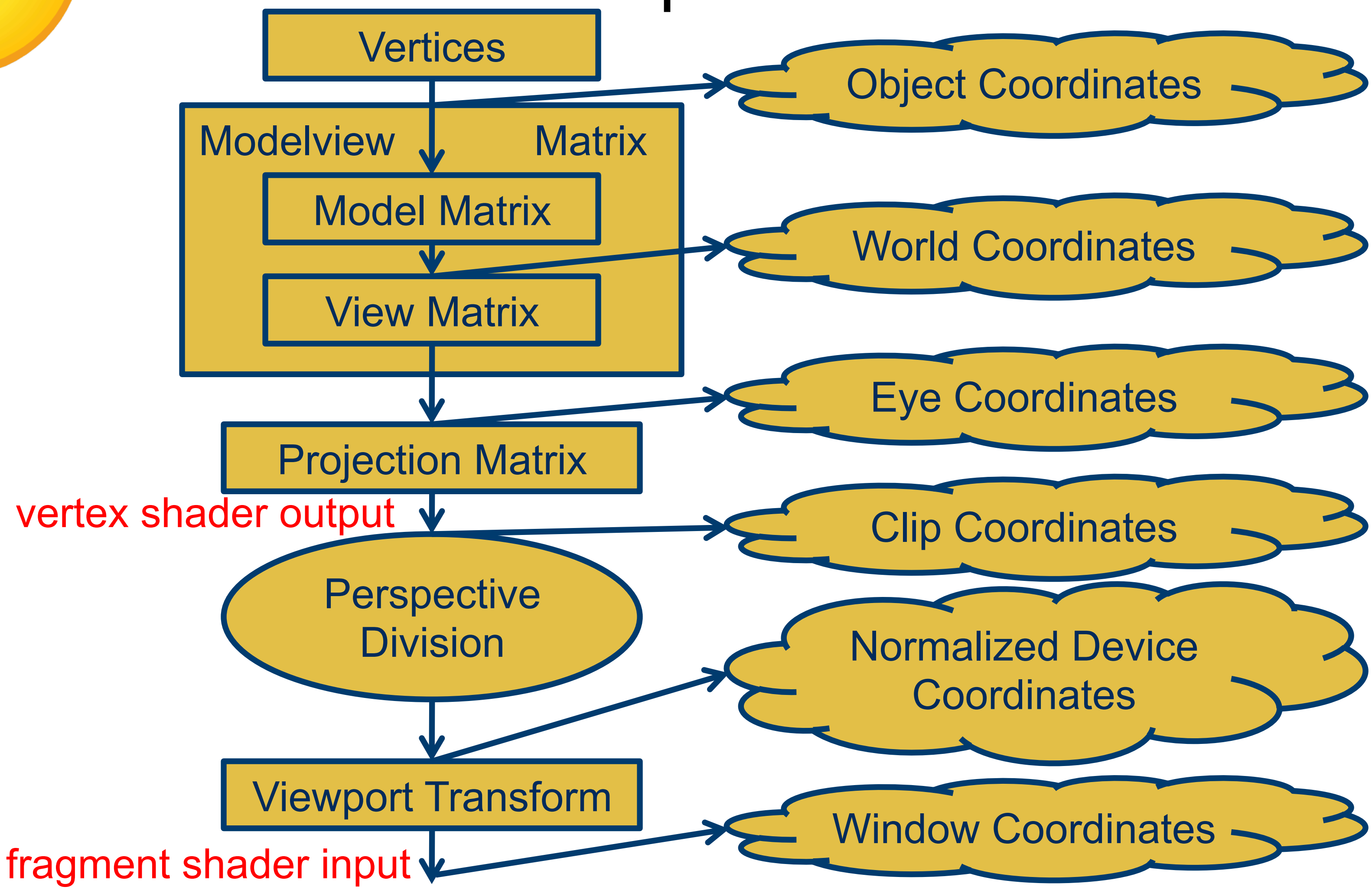
Lights and Materials

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

Coordinate Systems

- The units in `points` are determined by the application and are called object, world, or model 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

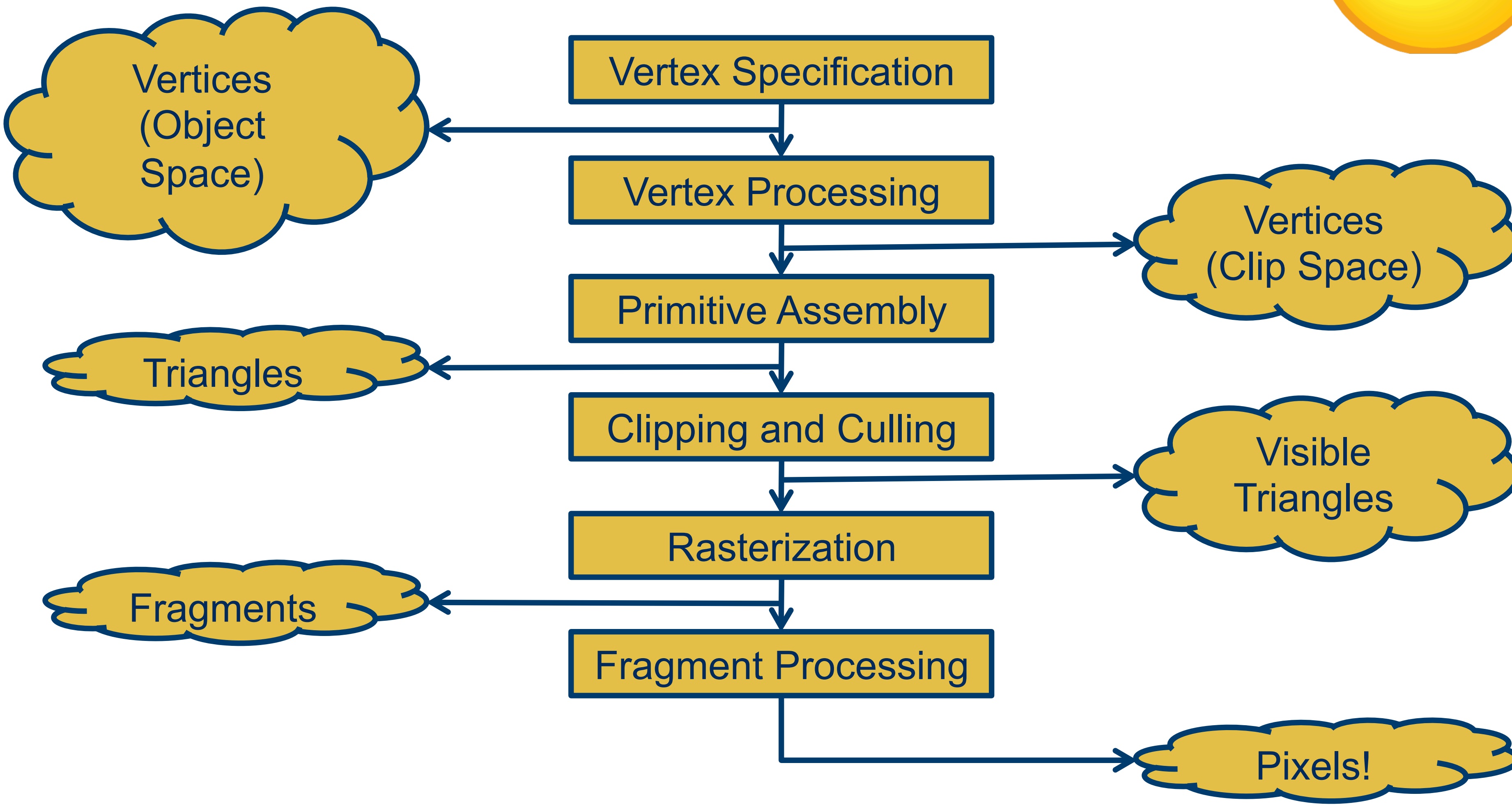
Vertex Transformation Pipeline



Coordinate Systems and Shaders

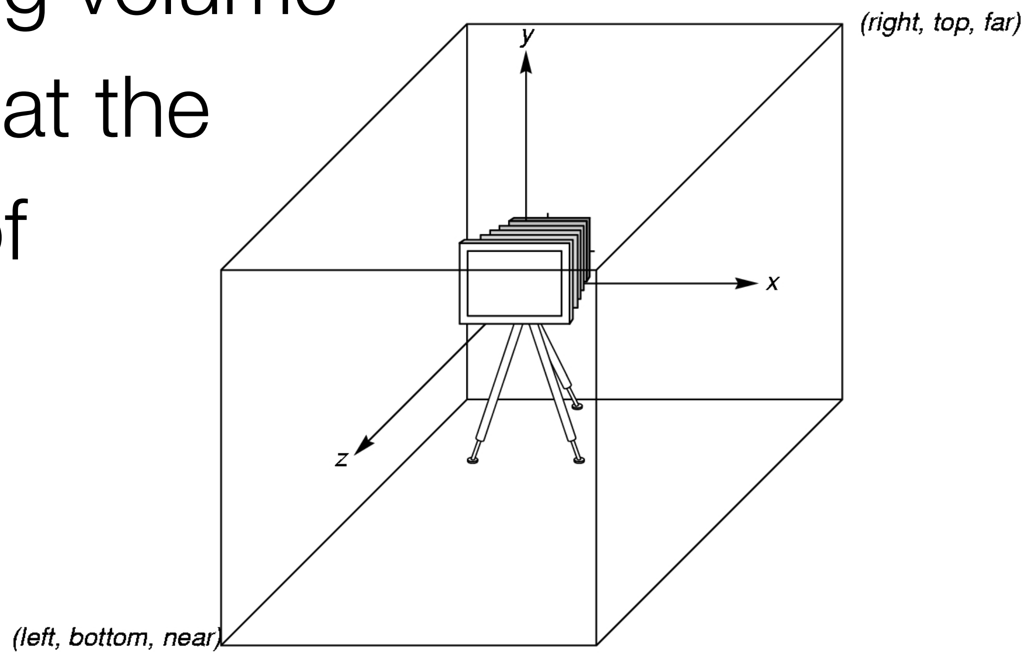
- Vertex shader must output in clip coordinates
- Input to fragment shader from rasterizer is in window coordinates
- Application can provide vertex data in any coordinate system but shader must eventually produce `gl_Position` in clip coordinates

WebGL Rendering Pipeline



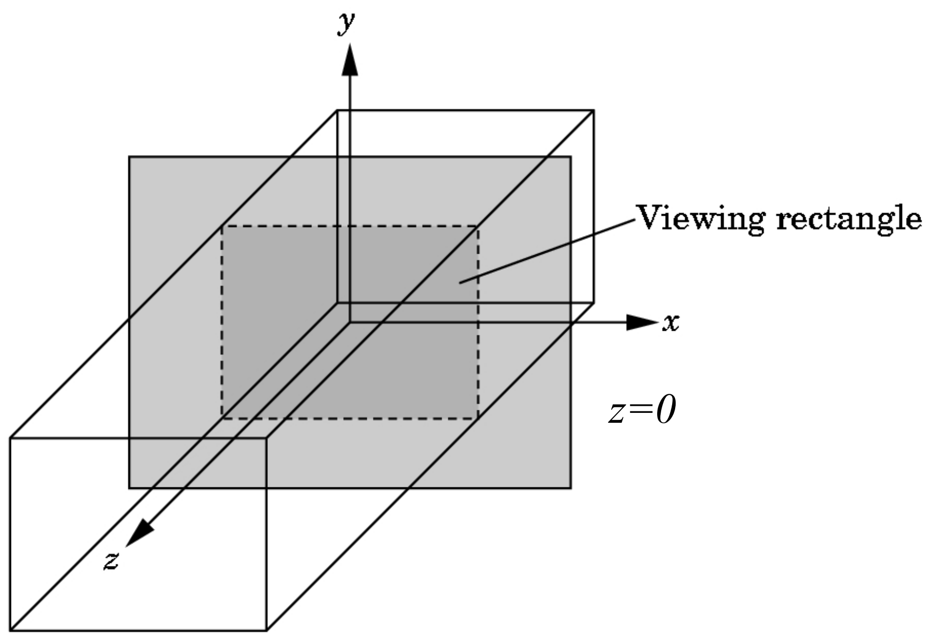
WebGL Camera

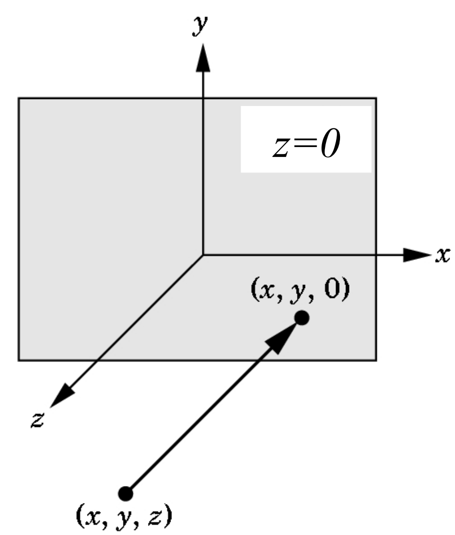
- WebGL places a camera at the origin in object space pointing in the negative z direction
- The default viewing volume is a box centered at the origin with sides of length 2



Orthographic Viewing

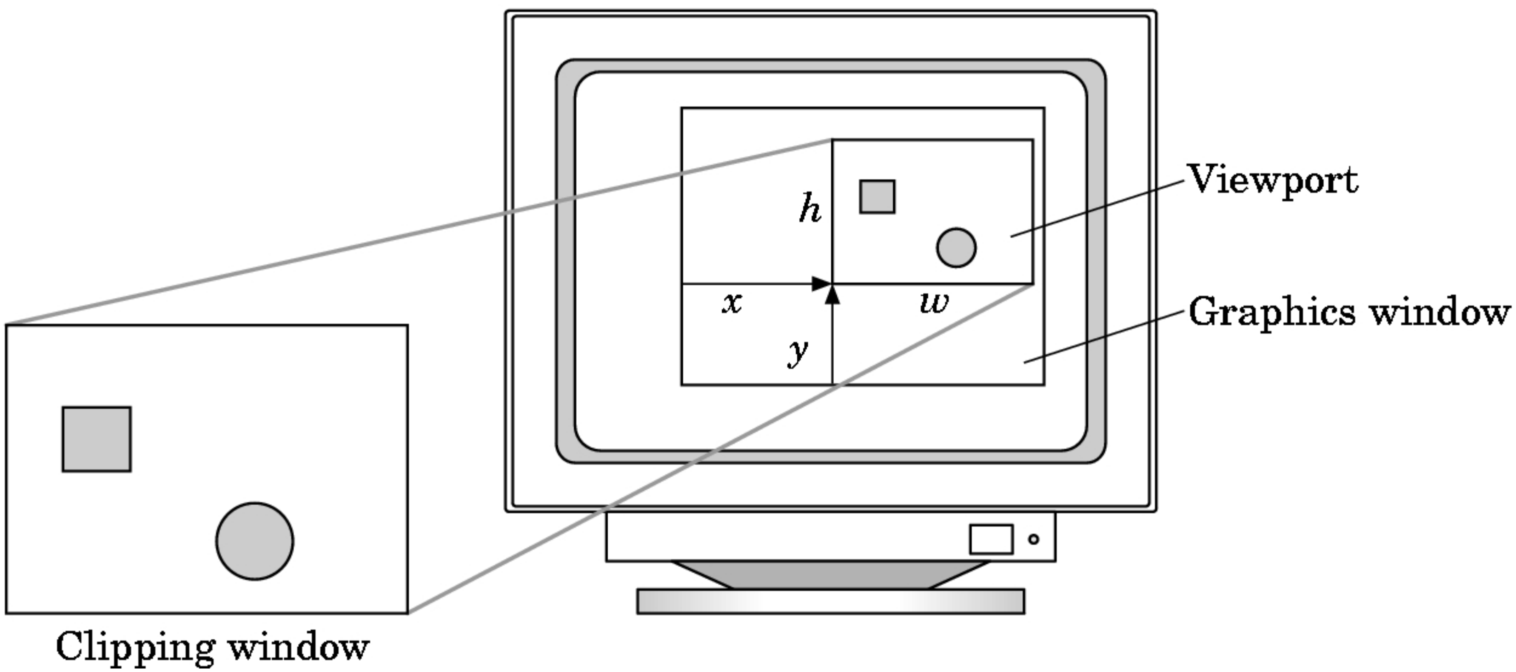
In the default orthographic view, points are projected forward along the z axis onto the plane z = 0





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

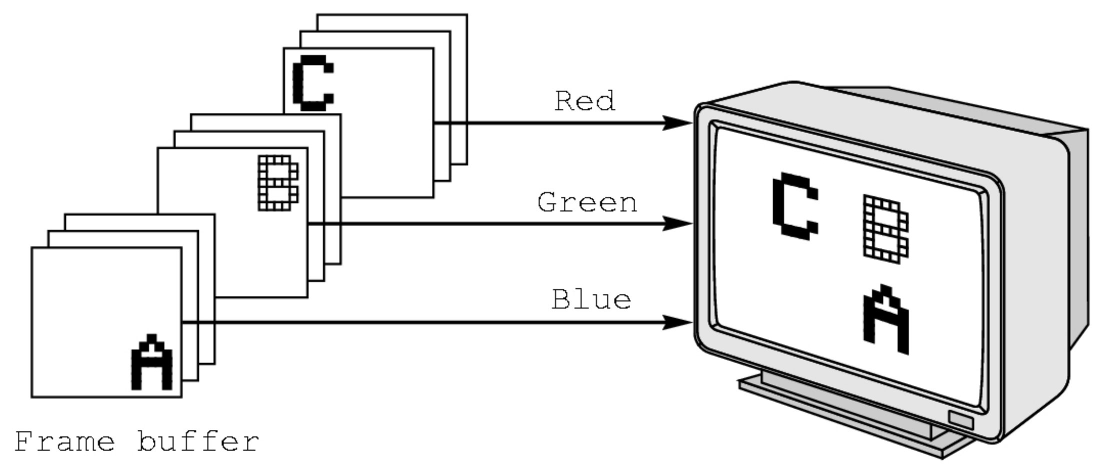
- In WebGL, we usually carry out projection using a projection matrix (transformation) before rasterization
- Transformation functions are also used for changes in coordinate systems
- Pre 3.1 OpenGL had a set of transformation functions which have been deprecated
- Three choices in WebGL
  - Application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

Attributes

- Attributes determine the appearance of objects
  - Color (points, lines, polygons)
  - Size and width (points, lines)
  - Stipple pattern (lines, polygons)
  - Polygon mode
    - Display as filled: solid color or stipple pattern
    - Display edges and vertices
- Only a few (`gl_PointSize`) are supported by WebGL functions

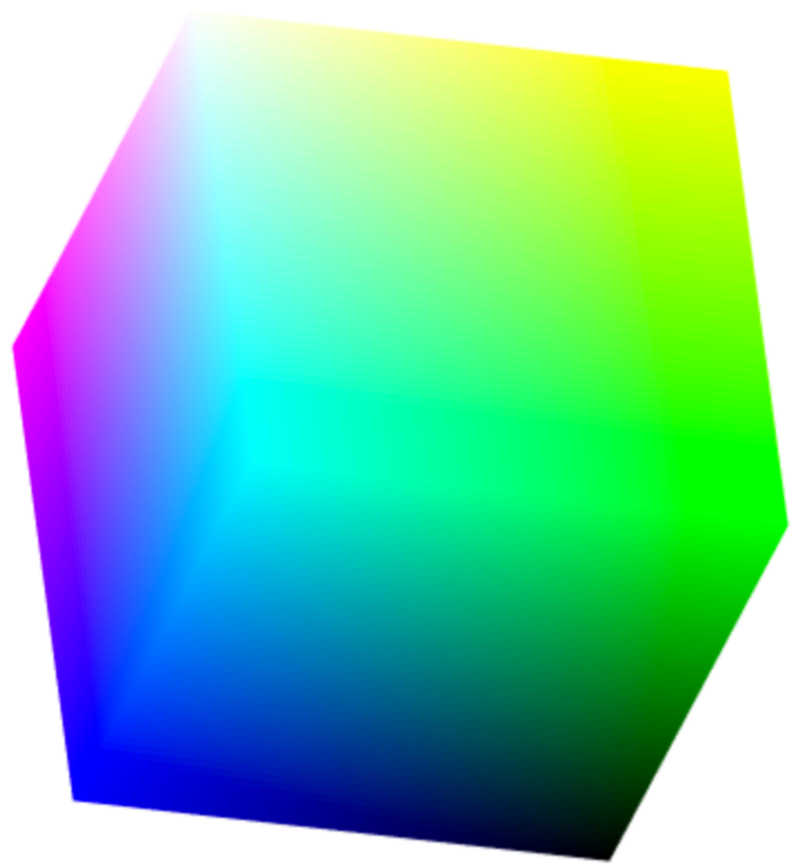
RGB color

- Each color component is stored separately in the frame buffer
- Usually 8 bits per component in buffer
- Color values can range from 0.0 (none) to 1.0 (all) using floats or over the range from 0 to 255 using unsigned bytes



Smooth Color

- Default is smooth shading
  - Rasterizer interpolates vertex colors across visible polygons
- Alternative is flat shading
  - Color of first vertex determines fill color

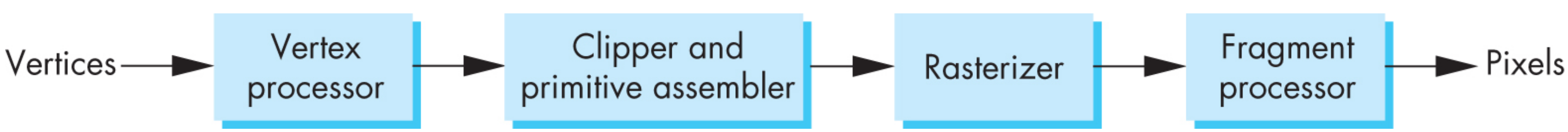


Setting Colors

- Colors are ultimately set in the fragment shader but can be determined in either shader or in the application
- Application color: pass to vertex shader as a uniform variable or as a vertex attribute
- Vertex shader color: pass to fragment shader as varying variable
- Fragment color: can alter via shader code

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 (this is what we do)

OpenGL 3.1

- Totally shader-based
  - No default shaders
  - Each application must provide both a vertex shader and a fragment shader
- No immediate mode
- Few state variables
- Most OpenGL 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 OpenGL ES 2.0
  - Supported on newer browsers
- OpenGL 4.1, 4.2, …
  - Add geometry, tessellation, compute shaders

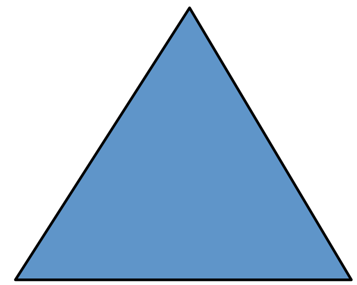
3D Sierpinski Gasket

Three-dimensional Applications

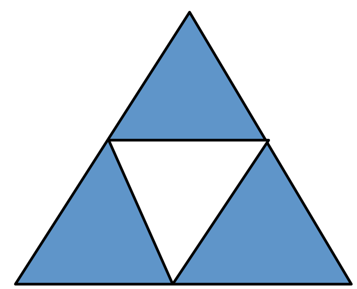
- In WebGL, two-dimensional applications are a special case of three-dimensional graphics
- Going to 3D
  - Not much changes
  - Use `vec3`, `gl.uniform3f`
  - Have to worry about the order in which primitives are rendered or use hidden-surface removal

Sierpinski Gasket (2D)

- Start with a triangle



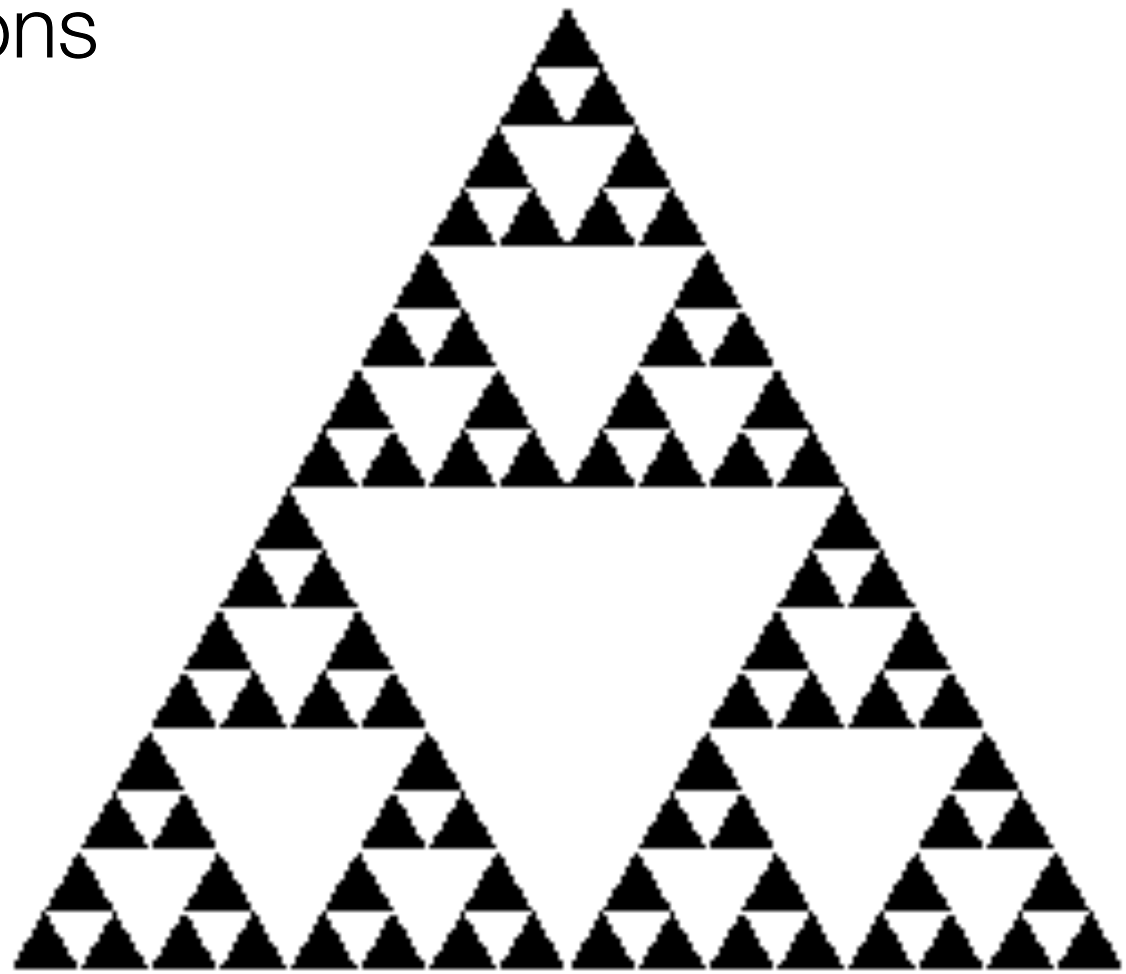
- Connect bisectors of sides and remove central triangle



- Repeat

Example

- Five subdivisions



The Gasket as a Fractal

- Consider the filled area (black) and the perimeter (the length of all the lines around the filled triangles)
- As we continue subdividing
  - the area goes to zero
  - but the perimeter goes to infinity
- This is not an ordinary geometric object
  - It is neither two- nor three-dimensional
- It is a fractal (fractional dimension) object

Gasket Program

- HTML file
  - Same as in other examples
  - Pass through vertex shader
  - Fragment shader sets color
  - Read in JS file

Gasket Program

```
var points = [];
var NumTimesToSubdivide = 5;

// initial triangle

 var vertices = [
        vec2( -1, -1 ),
        vec2(  0,  1 ),
        vec2(  1, -1 )
    ];

divideTriangle( vertices[0], vertices[1],
     vertices[2], NumTimesToSubdivide);
```

Draw One Triangle

```
// display one triangle

function triangle( a, b, c ){
    points.push( a, b, c );
}
```

Triangle Subdivision

```
function divideTriangle( a, b, c, count ){
  // check for end of recursion
  if ( count === 0 ) {
    triangle( a, b, c );
  }
  else {
    // bisect the sides
    var ab = mix( a, b, 0.5 );
    var ac = mix( a, c, 0.5 );
    var bc = mix( b, c, 0.5 );
    --count;
    // three new triangles
    divideTriangle( a, ab, ac, count );
    divideTriangle( c, ac, bc, count );
    divideTriangle( b, bc, ab, count );
  }
}
```

init()

```
var program = initShaders( gl, "vertex-shader",
  "fragment-shader" );
gl.useProgram( program );

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 );

render();
```

Render Function

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

Moving to 3D

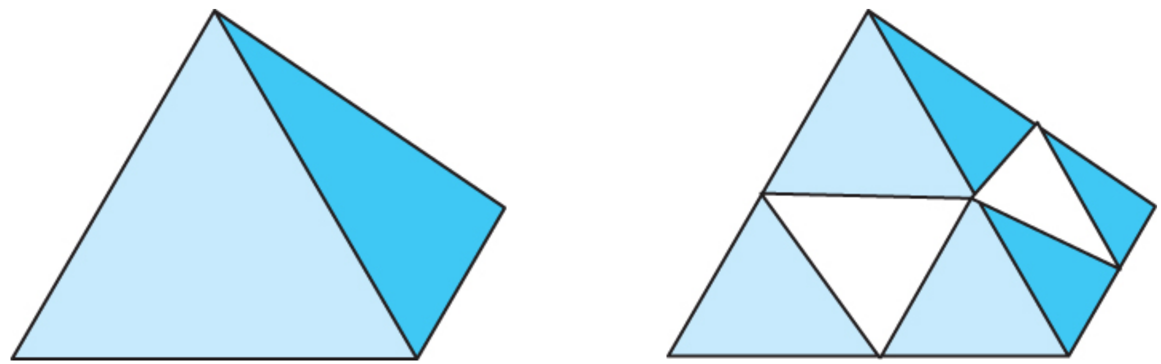
- We can easily make the program three-dimensional by using three dimensional points and starting with a tetrahedron

```
var vertices = [
  vec3(  0.0000,  0.0000, -1.0000 ),
  vec3(  0.0000,  0.9428,  0.3333 ),
  vec3( -0.8165, -0.4714,  0.3333 ),
  vec3(  0.8165, -0.4714,  0.3333 )
];
```

- Subdivide each face

3D Gasket

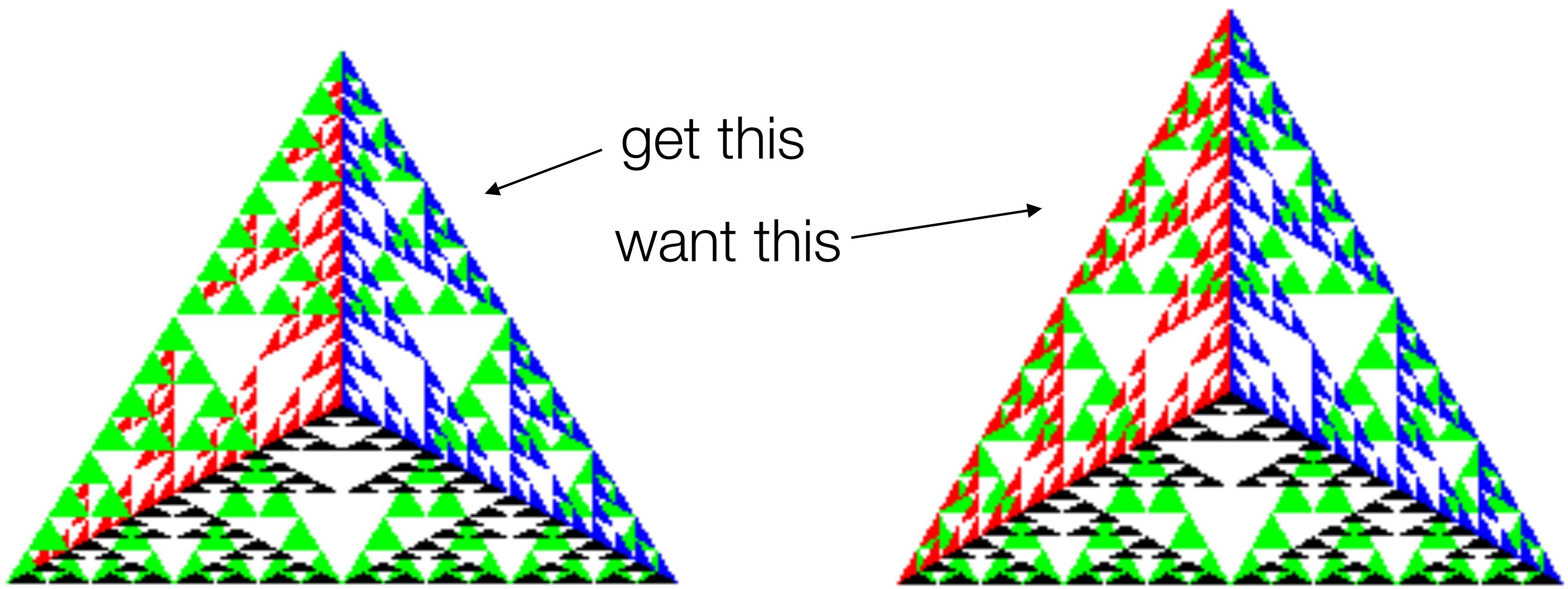
- We can subdivide each of the four faces



- Appears as if we remove a solid tetrahedron from the center leaving four smaller tetrahedra
- Code almost identical to 2D example

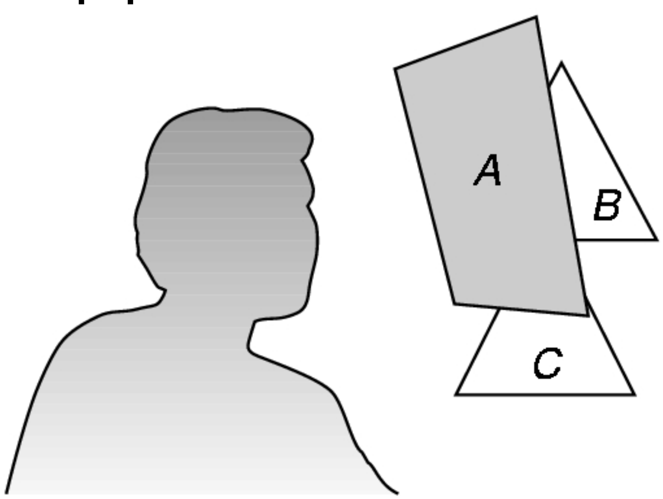
Almost Correct

- Because the triangles are drawn in the order they are specified in the program, the front triangles are not always rendered in front of triangles behind them



Hidden-Surface Removal

- We want to see only those surfaces in front of other surfaces
- OpenGL uses a hidden-surface method called the z-buffer algorithm that saves depth information as objects are rendered so that only the front objects appear in the image



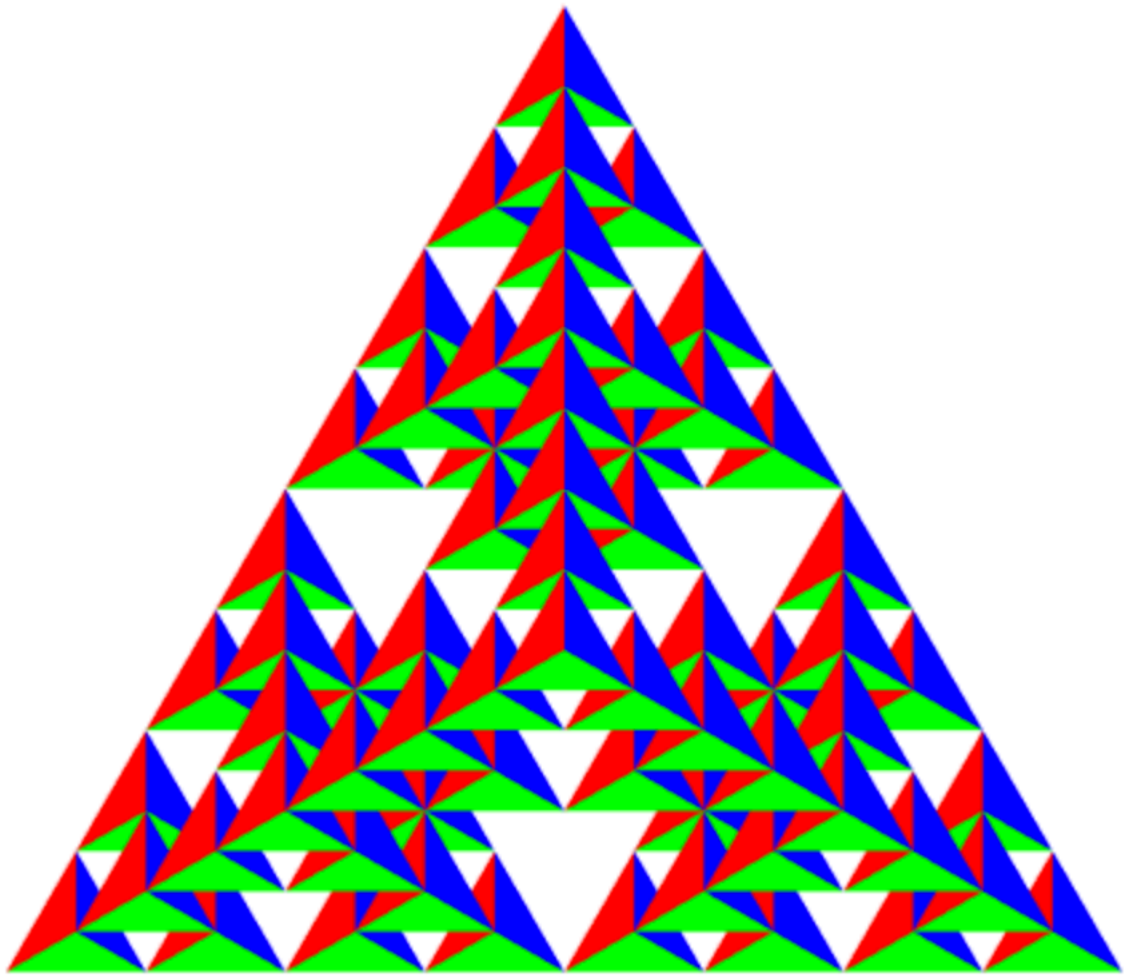
Using the z-buffer Algorithm

- The algorithm uses an extra buffer, the z-buffer, to store depth information as geometry travels down the pipeline
- Depth buffer is required to be available in WebGL
- It must be
  – Enabled
    - `gl.enable(gl.DEPTH_TEST)`
  – Cleared in for each render
    - `gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT)`

Surface vs. Volume Subdvision

- In our example, we divided the surface of each face
- We could also divide the volume using the same midpoints
- The midpoints define four smaller tetrahedrons, one for each vertex
- Keeping only these tetrahedrons removes a volume in the middle
- See text for code

Volume Subdivision



How to run things locally?

- You can write shaders into separate files and read in from JS application (see example code `gasket1v2`)
- For most browsers, the default security setting does not allow you to read files locally
- But you can change that:
  - Check the WebGL programming notes
- For security purpose, you may want to check the setting back after running the program

Exercise

- Go over each of the examples for gasket and explain what each program does
- Make sure that you can run the example `gasket1v2` by proper setting the browser (we read shader files from local directory)
- For the example `gasket4`, take out code for hidden-surface removal, run it and explain the result you see