

# CS 418: Interactive Computer Graphics

---

## Introduction to Rendering

Eric Shaffer

Slides adapted from  
Professor John Hart's CS 418 Slides

Some slides adapted from  
Angel and Shreiner: Interactive Computer Graphics 7E  
© Addison-Wesley 2015

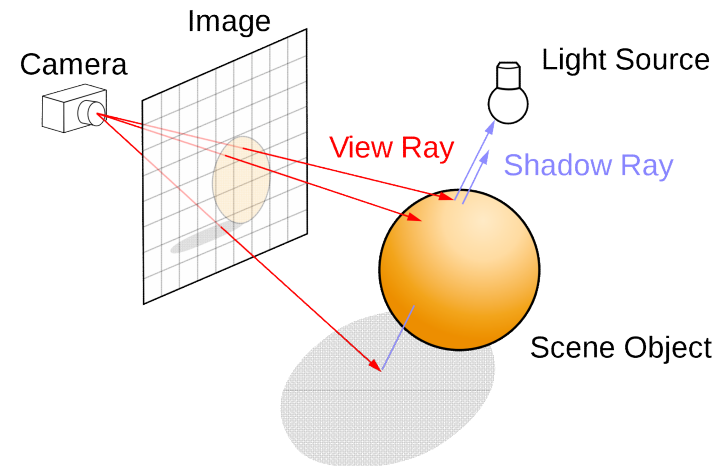
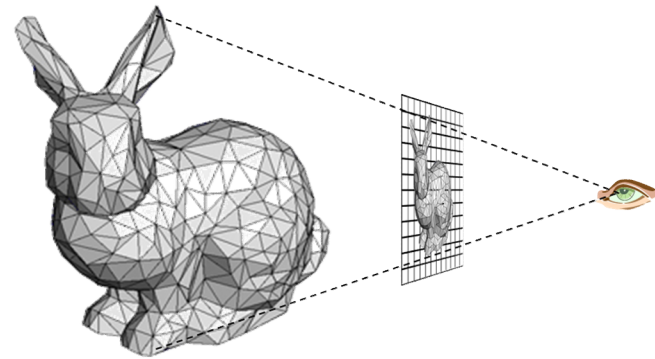
# Rendering

- ▣ **Rendering or image synthesis** Rendering or image synthesis is the automatic process of generating a photorealistic or non-photorealistic image from a 2D or 3D model
- ▣ Rendering methods generally use one of two approaches
  - ▣ Rasterization (focus of CS 418)
  - ▣ Ray Tracing (focus of CS 419)
  - ▣ Though, sometimes you can use both....
  - ▣ ...and there are other methods like radiosity



# Rasterization versus Ray Tracing

- To oversimplify....
- In rasterization, geometric primitives are projected onto an image plane and the rasterizer figures out which pixels get filled.
- In ray-tracing, we the physical transport of light by shooting a sampling ray through each pixel in an image plane and seeing what the ray hits in the scene



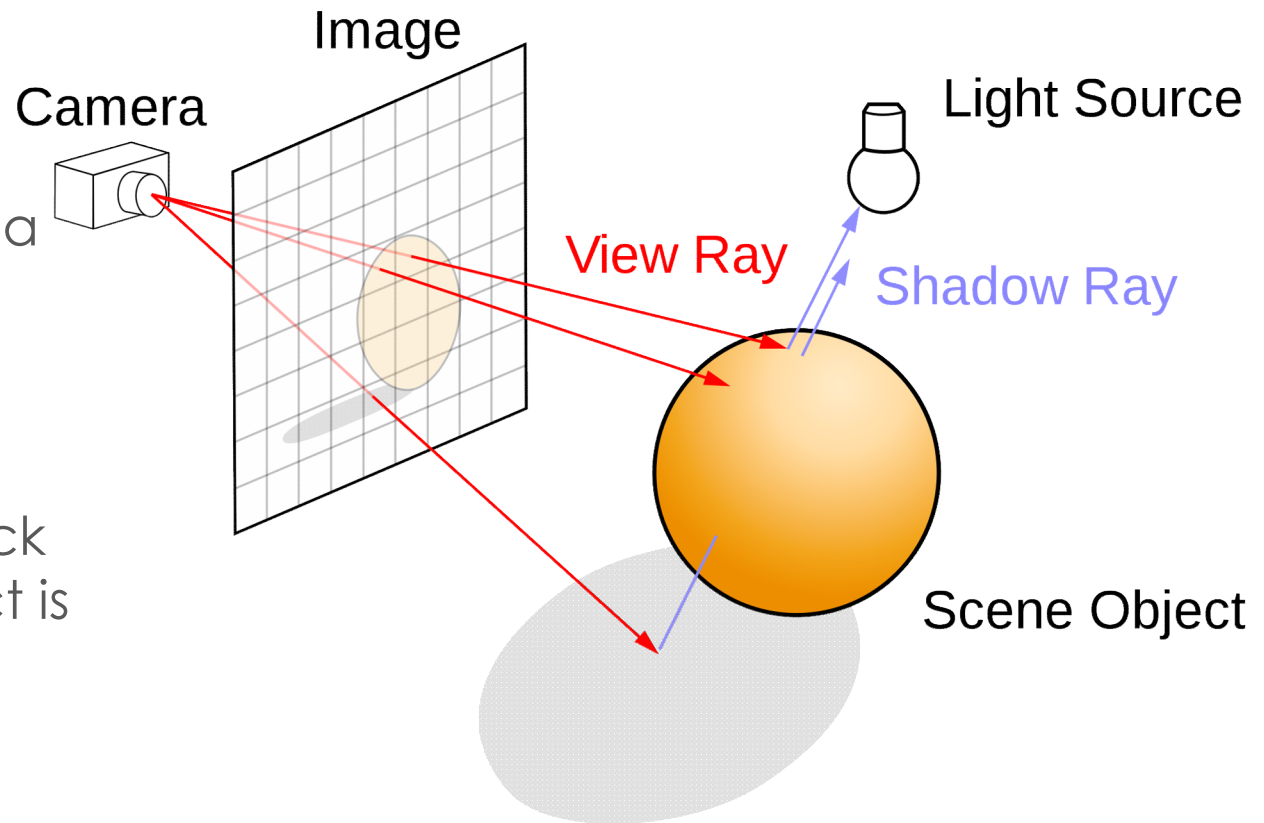
# Ray Tracing (alternative to rasterization)

Follow ray of light....

Can trace from an  
eyepoint through a  
pixel

See what object the  
ray hits...

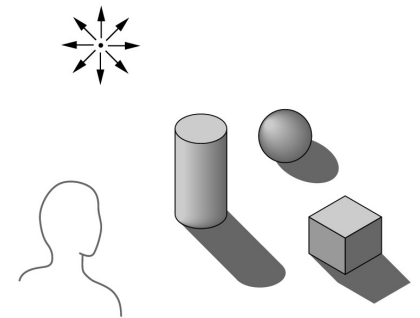
How would you check  
to see if the object is  
lit?



# Global vs Local Lighting

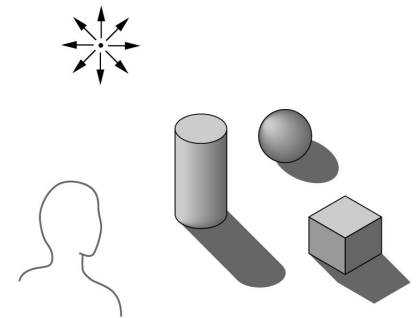
- ▣ For true photo-realism:  
Cannot compute color or shade of each object independently

Why?



# Global vs Local Lighting

- For true photo-realism:  
Cannot compute color or shade of each object independently
  - Some objects are blocked from light
  - Light can reflect from object to object
  - Some objects might be translucent
- Can rasterization produce global lighting effects?
- Can ray tracing?
- The big advantage of rasterization is...?



# Rasterization Engines

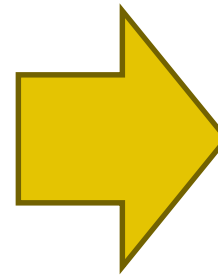
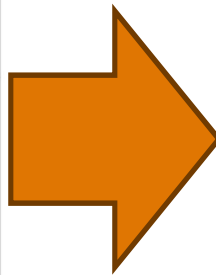
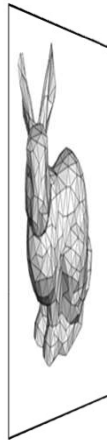
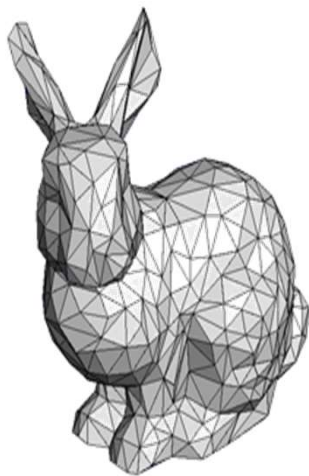
- Most low-level graphics libraries use a synthetic camera model
- Application Programmer Interface (API)
  - Requires user to specify
    - Objects in the scene
    - Materials the objects are made of
    - Viewer (position, view direction, field of view,...)
    - Lights - what parameters do you think typically are used?
- The engine (i.e. the library) will use pipeline-style processing
  - The input geometry flows through several processing stages

# 3-D Graphics Pipeline

Vertex  
Processing

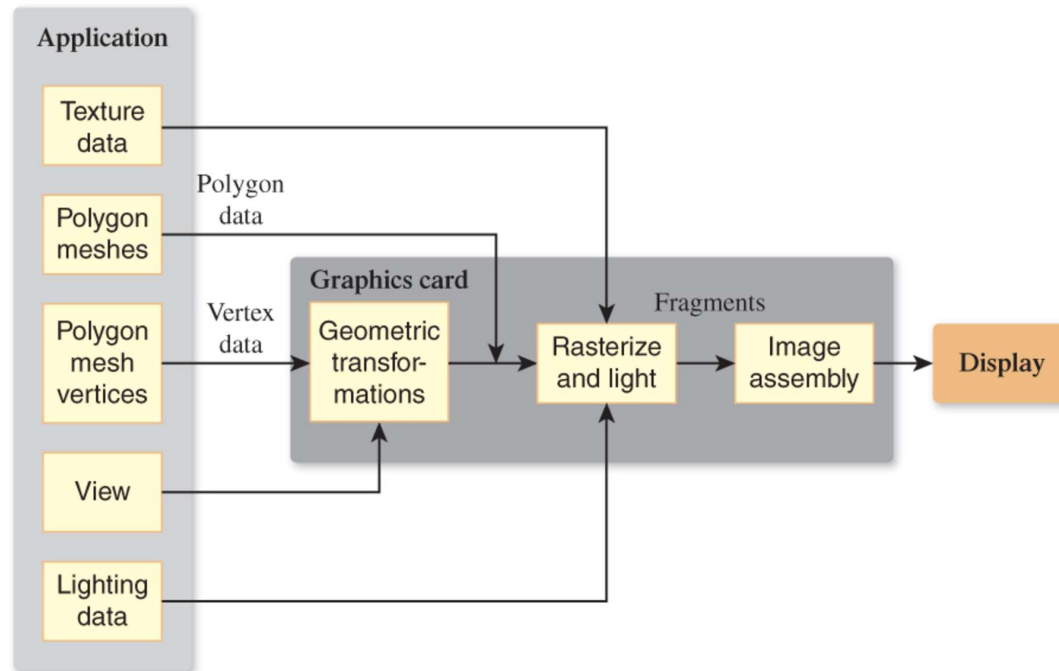
Rasterization

Fragment  
Processing





# Rasterization is a Pipeline



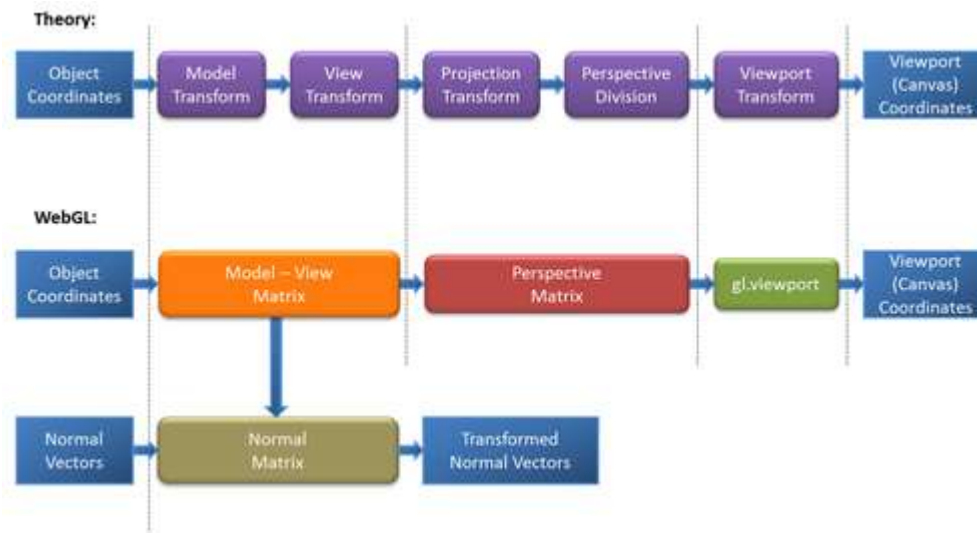
- Data for objects in the scene usually in the form of polygonal meshes
- Most of the work to render an image is done on the Graphics Processing Unit (GPU)
- GPU code will have at least two parts
  - Vertex Shader
  - Fragment Shader

# Vertex Shader

- ▣ Vertex shader typically transforms vertex locations from one coordinate system to another
  - ▣ Transformations can be useful for placing objects in your scene
  - ▣ Also, some operations on the geometry are easier when done in specific coordinate system
- ▣ Change of coordinates equivalent to a matrix transformation
- ▣ Vertex processor often also computes vertex colors

# Pipeline Step: Projection

- *Projection* is the process that generates a 2D image of 3D geometry
  - Perspective projections: all projectors meet at the center of projection
    - Requires 3D viewer position with the 3D object position
  - Parallel projection: projectors are parallel, center of projection is replaced by a direction of projection
- Process involves both your vertex shader code and the webgl library



We'll go over the details later in the semester

# Pipeline Step: Primitive Assembly

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

- For WebGL: Points, Line Segments, Polygons
- Other APIs sometimes support more complex geometry (e.g. curves)

Vertex Shader



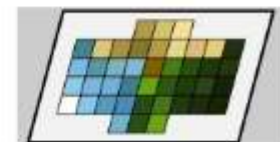
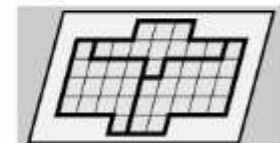
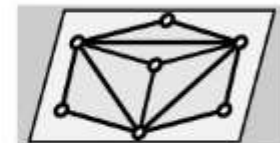
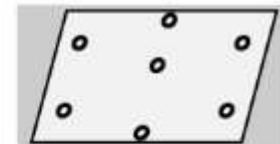
Triangle Assembly



Rasterization

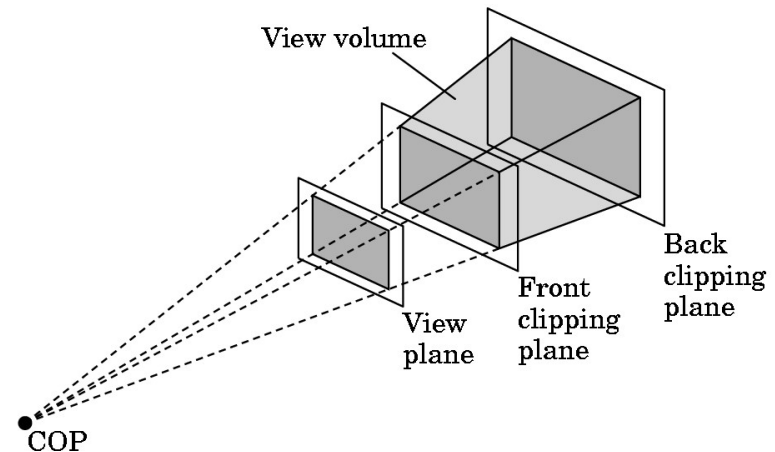
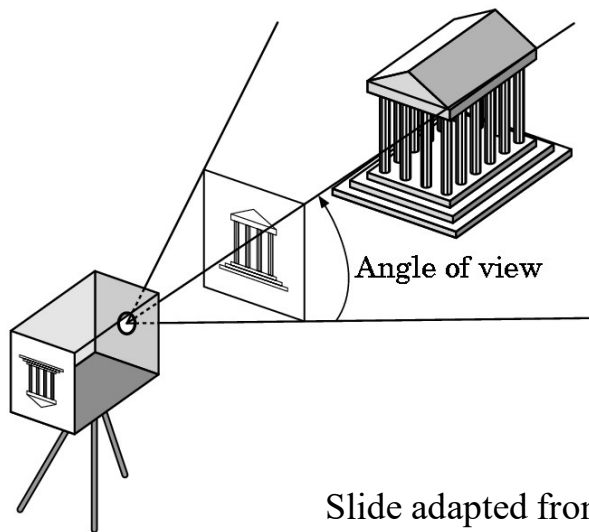


Fragment Shader



# Pipeline Step: Clipping

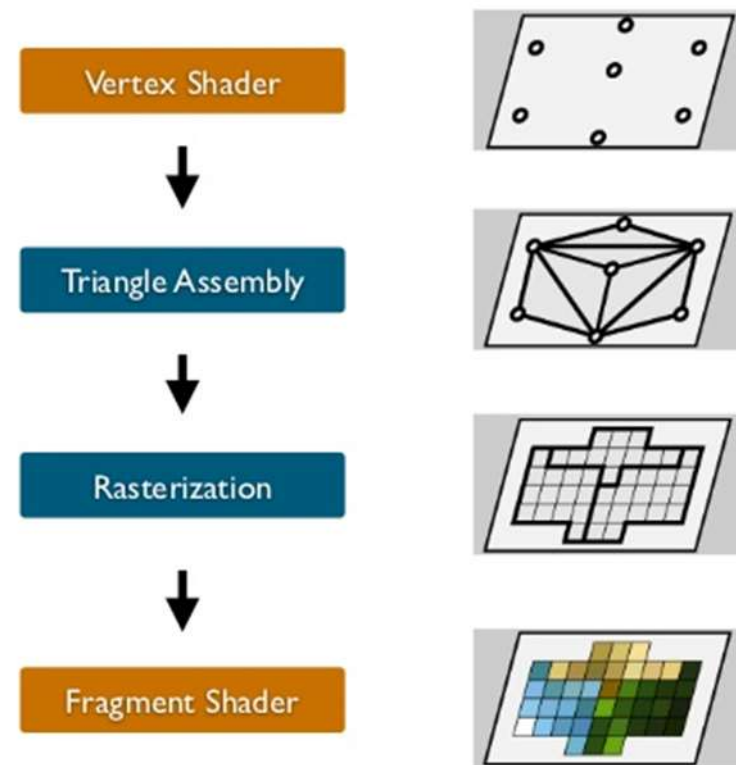
- Just as a real camera cannot “see” the whole world, the virtual camera can only see part of the world or object space
- Objects not within this volume are said to be *clipped* out of the scene



Slide adapted from  
Angel and Shreiner: Interactive Computer Graphics  
7E © Addison-Wesley 2015

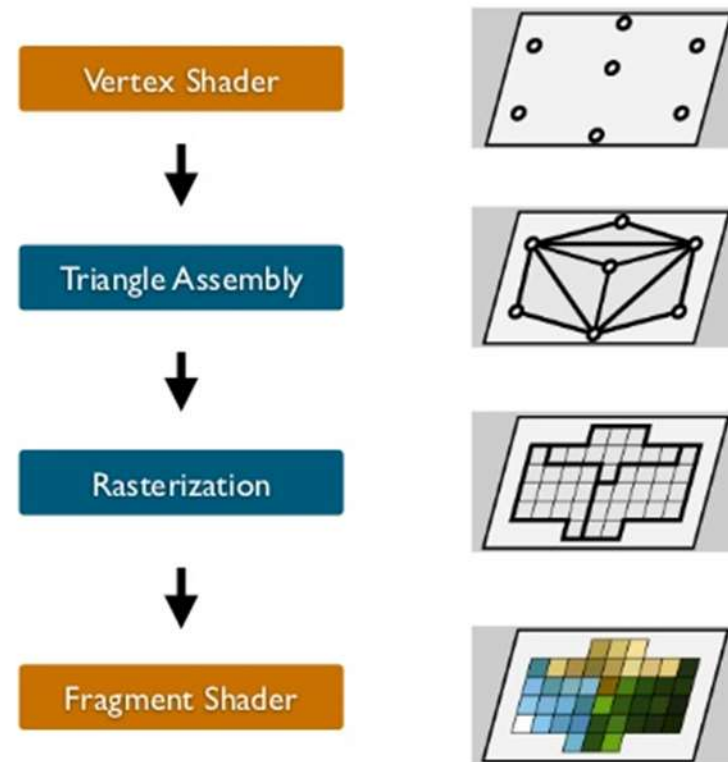
# Rasterization

- ❑ 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



# Pipeline Step: Fragment Processing

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



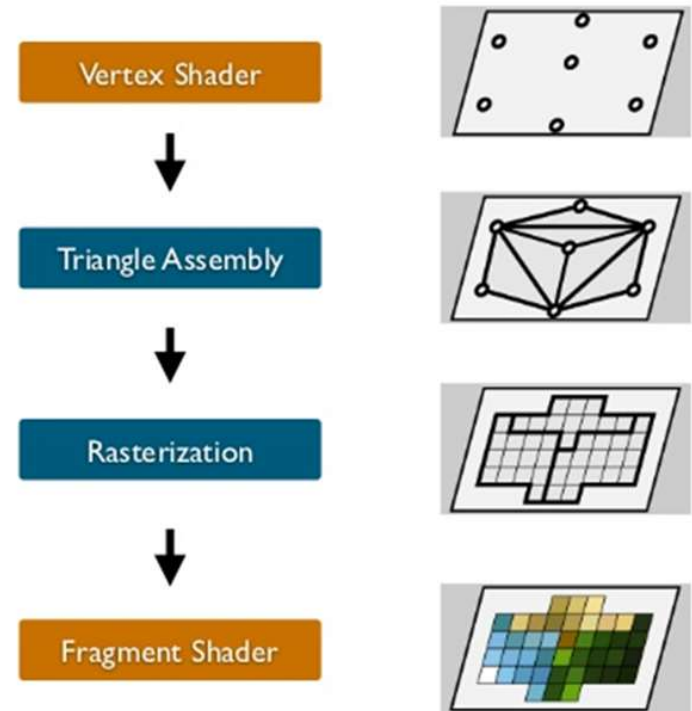
# Creating Your Scene

## Step 1: Specifying Geometry

- Put geometric data in an array

```
var triangleVertices = [  
    0.0, 0.5, 0.0,  
    -0.5, -0.5, 0.0,  
    0.5, -0.5, 0.0  
];
```

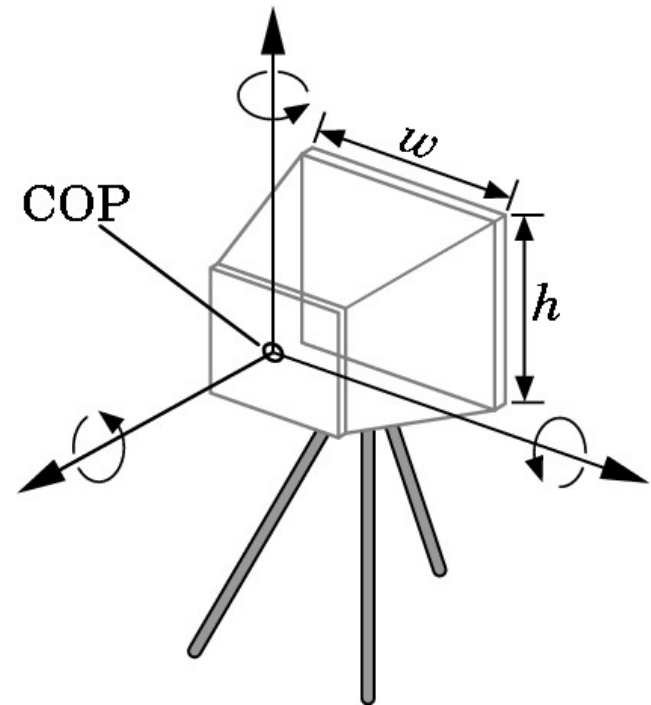
- Specify material properties for geometry
  - Color properties
  - Maybe advanced shading information
    - Is it shiny like metal or dull like paper?
  - We'll see how in future lectures
- Your code will then send array to GPU
  - And tell GPU to render as triangle





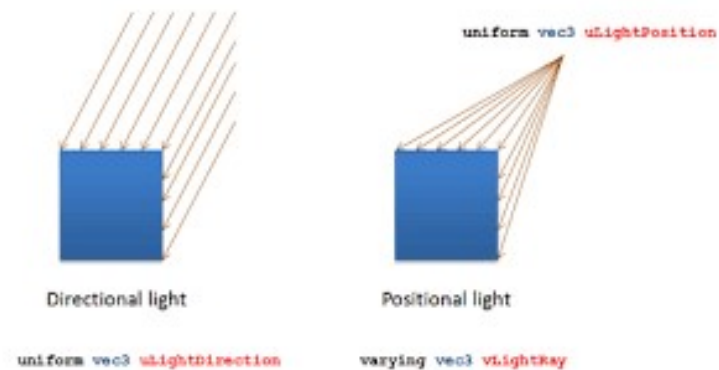
## Step 2: Specify Camera Parameters

- Specify an eye position
  - ...and a direction to look in
  - ...and a viewing volume for clipping
- Some APIs let you do more
  - What else might you want to do?



# Step 3: Specify Lights

- Types of lights
  - Near and far sources – physically, what is the major difference?
  - Color properties
  - WebGL easily supports ambient, directional, and point lights
    - But you can implement other types....



# WebGL Application Structure

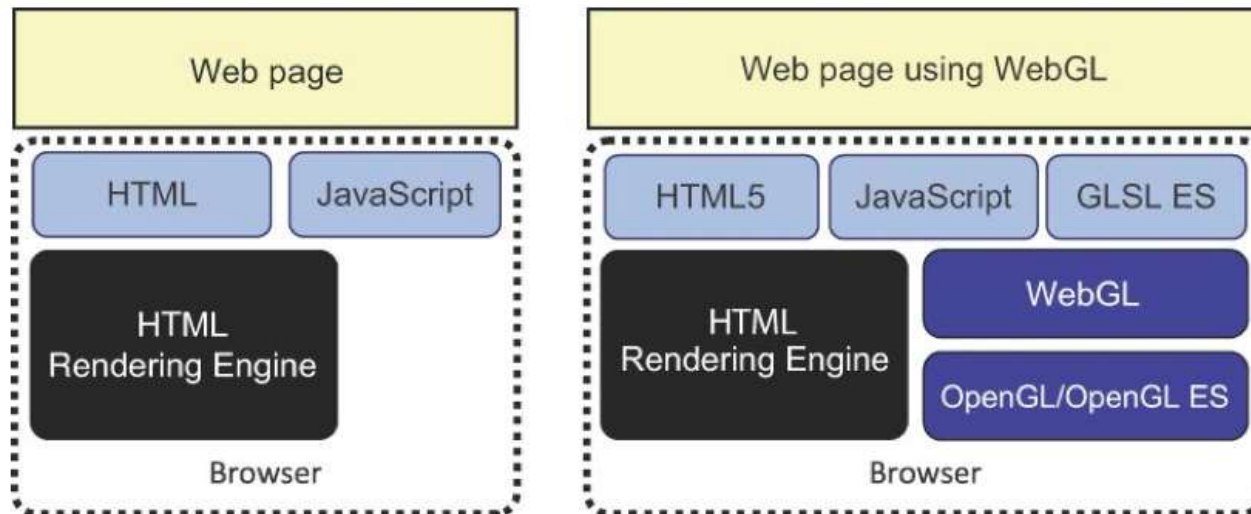


Figure from *WebGL Programming Guide: Interactive 3D Graphics Programming with WebGL* by Matsuda and Lea

Your application will generally just have HTML and JavaScript files

---

# What you will learn...later today...or Friday

- ▣ Create a basic but complete WebGL application
  - ▣ Create a WebGL context
  - ▣ Write a simple vertex shader and a fragment shader
  - ▣ Load your shader source code through the WebGL API
  - ▣ Compile and link your shaders
  - ▣ Load your vertex data into the WebGL buffers
  - ▣ Use the buffers to draw your scene
-

# WebGL function naming conventions

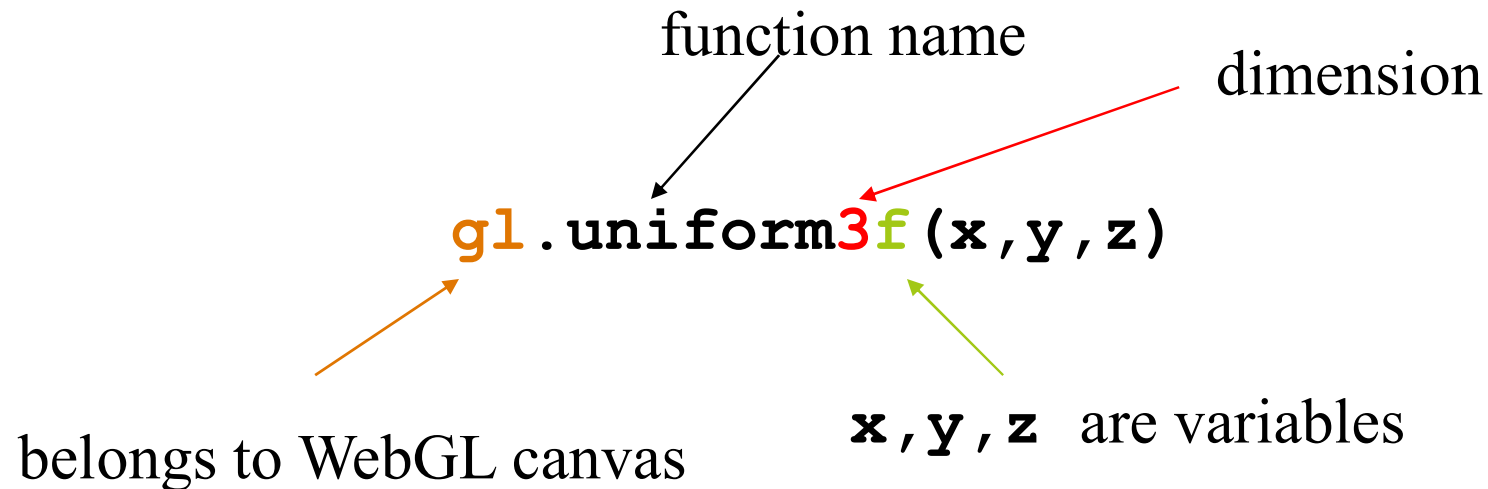
function name

dimension

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

belongs to WebGL canvas

`x, y, z` are variables



`gl.uniform3fv(p)`

`p` is an array



Slide adapted from  
Angel and Shreiner: Interactive Computer Graphics  
7E © Addison-Wesley 2015

# 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)`

Slide adapted from  
Angel and Shreiner: Interactive Computer Graphics  
7E © Addison-Wesley 2015

# WebGL and GLSL

- ▣ WebGL requires shaders
- ▣ 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

---

# Shaders

- ▣ Shader source code will be in the HTML file or a JS file...usually
  - ▣ At a minimum shaders must set the two required built-in variables
    - ▣ `gl_Position`
    - ▣ `gl_FragColor`
  - ▣ Vertex Shaders generally move vertices around
    - ▣ Projection, animation, etc.
  - ▣ Fragment Shaders generally determine a fragment color
-



# Shading

- Shading: The process of generating a color using lighting and material information
- You can do this in either shader
- Why is the per-fragment shading tighter?



per vertex shading



per fragment shading

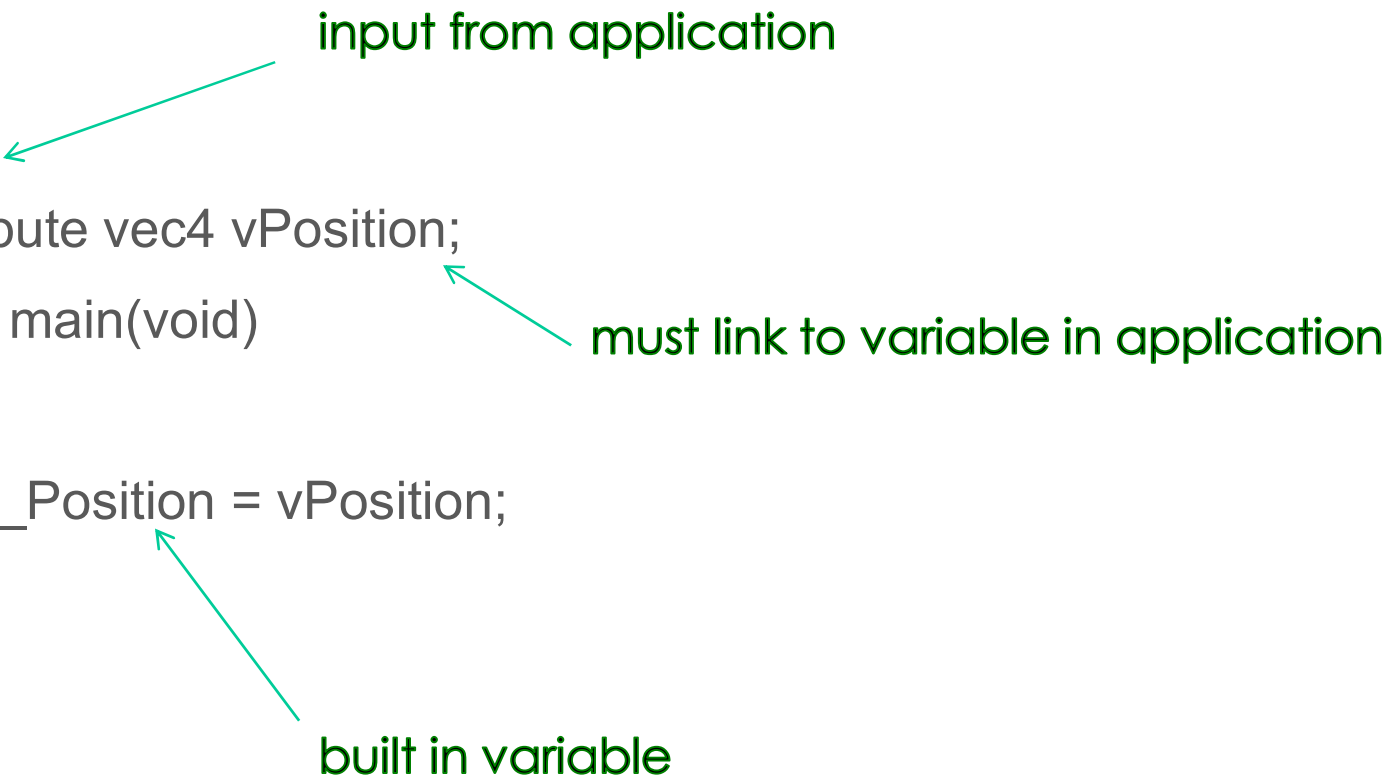
# Simple Vertex Shader

input from application

```
attribute vec4 vPosition;  
void main(void)  
{  
    gl_Position = vPosition;  
}
```

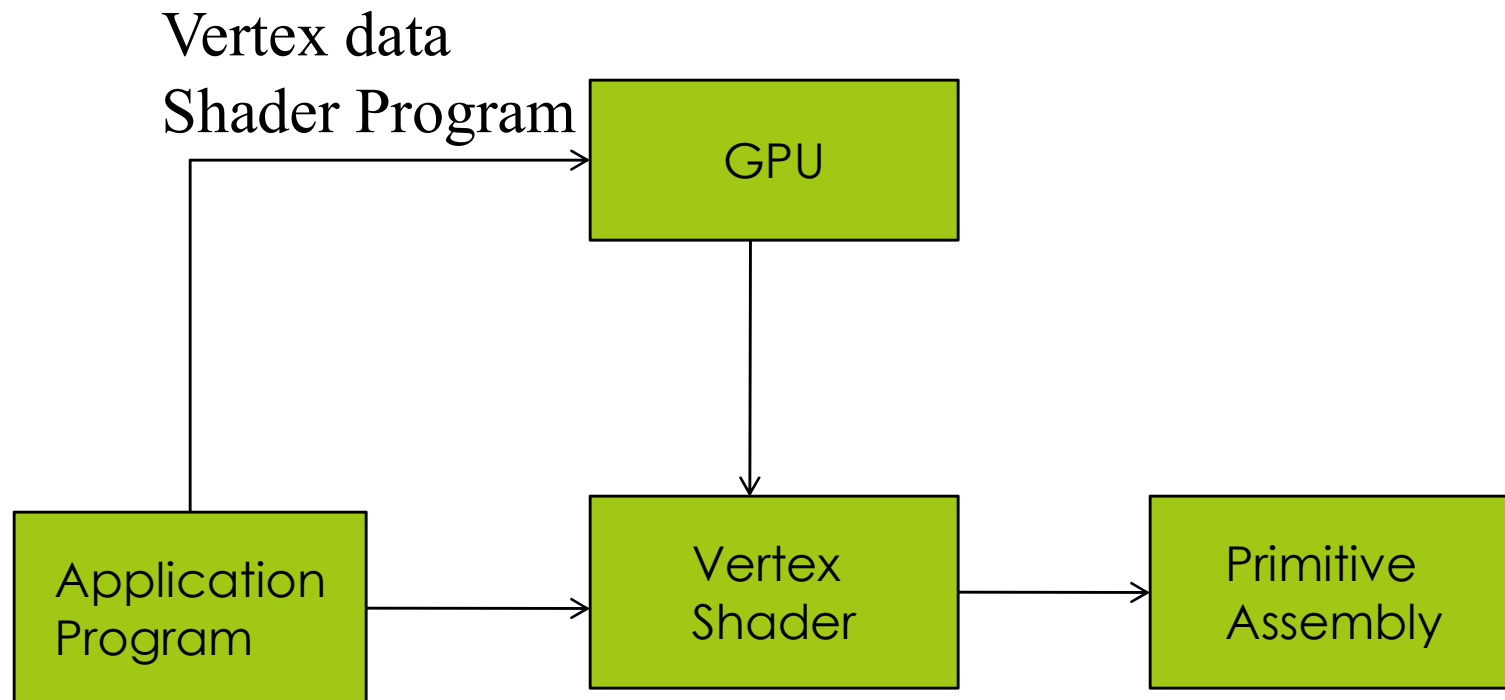
must link to variable in application

built in variable



Slide adapted from  
Angel and Shreiner: Interactive Computer Graphics  
7E © Addison-Wesley 2015

# Execution Model

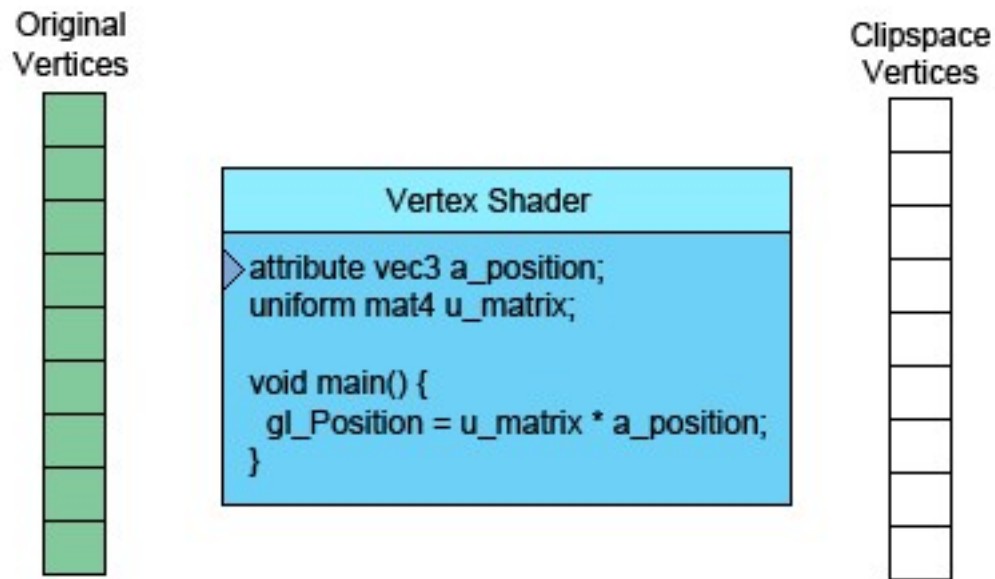


`gl.drawArrays`

Vertex

Slide adapted from  
Angel and Shreiner: Interactive Computer Graphics  
7E © Addison-Wesley 2015

# What a Vertex Shader Does...



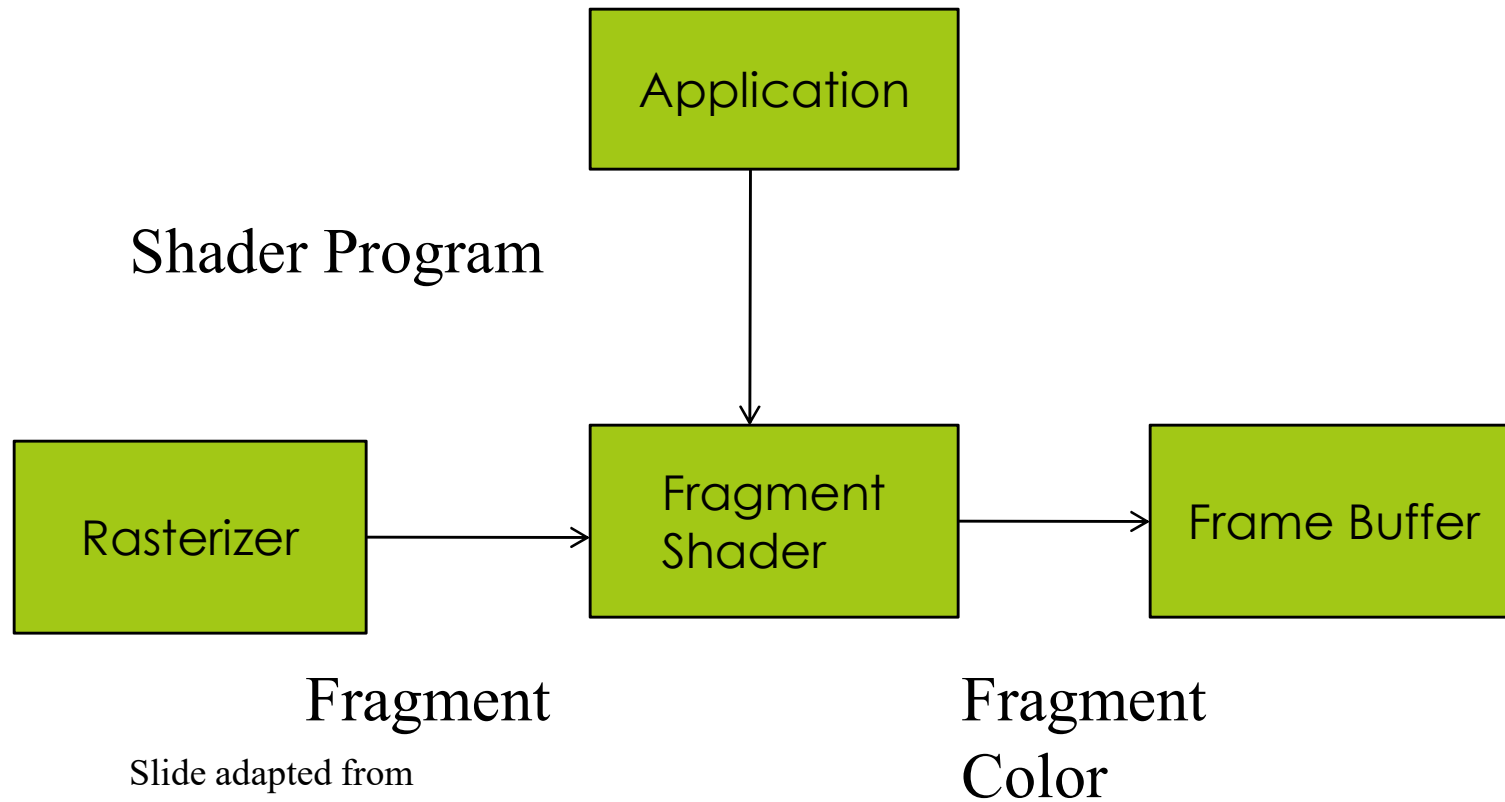
Taken from [webglfundamentals.org](http://webglfundamentals.org)

What is slightly incorrect about this animation?

# Simple Fragment Program

```
precision mediump float;  
void main(void)  
{  
    gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);  
}
```

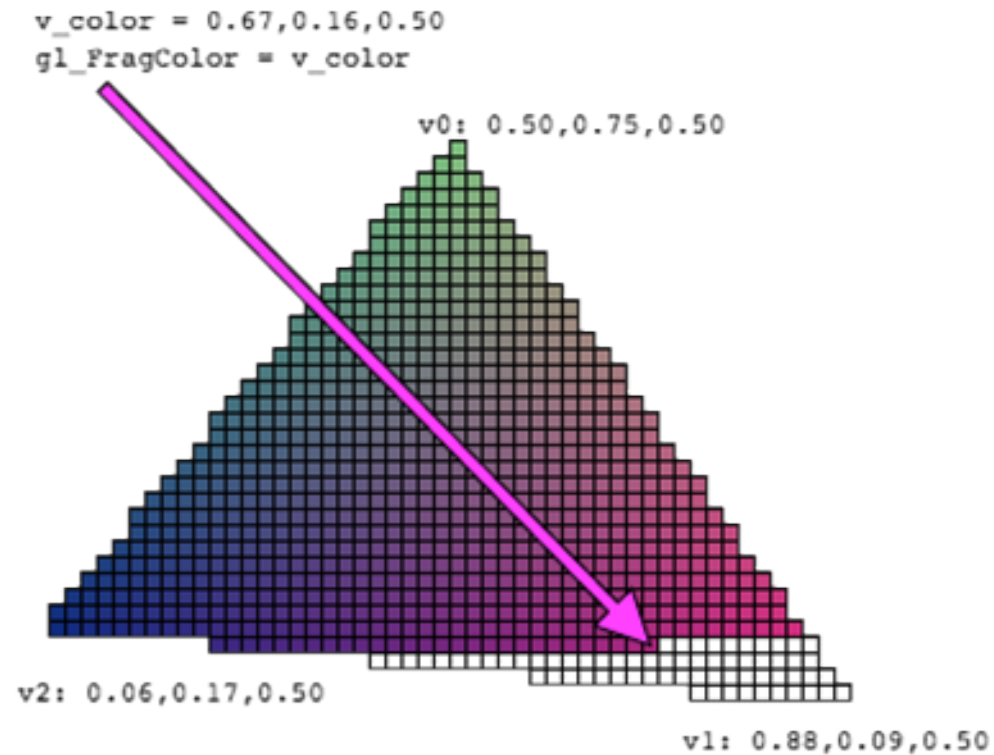
# Execution Model



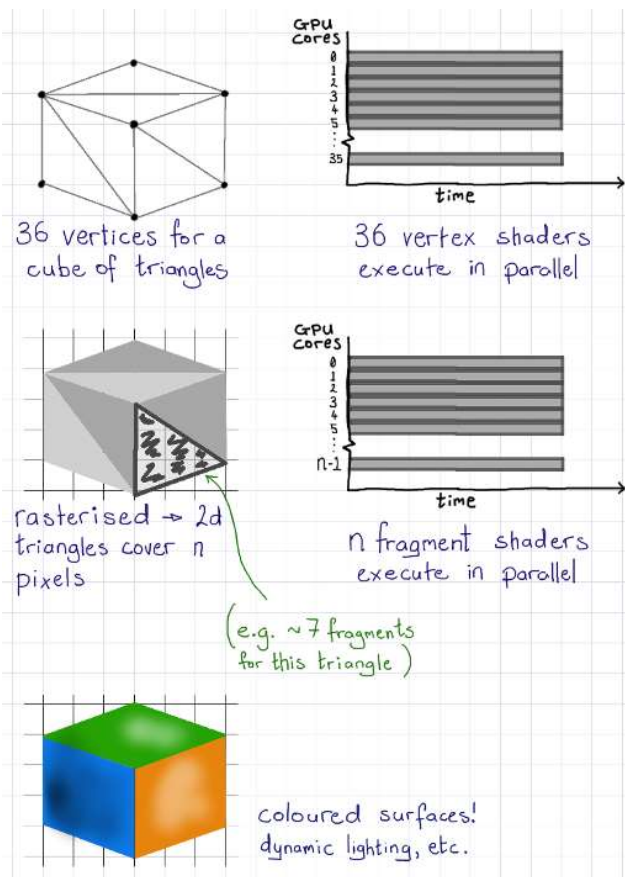
Slide adapted from  
Angel and Shreiner: Interactive Computer Graphics  
7E © Addison-Wesley 2015

Fragment  
Color

# What a Fragment Shader Does...



# Processing on a GPU



The Graphics Processing Unit (GPU) will have a large number of cores.

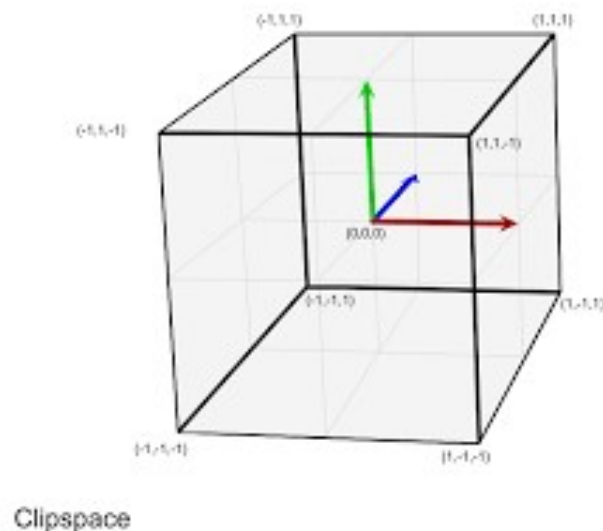
This architecture supports a massively-threaded environment for processing vertices and fragments (think of fragments as pixels for now)

Image from  
<http://antongerdelan.net/opengl/shaders.html>



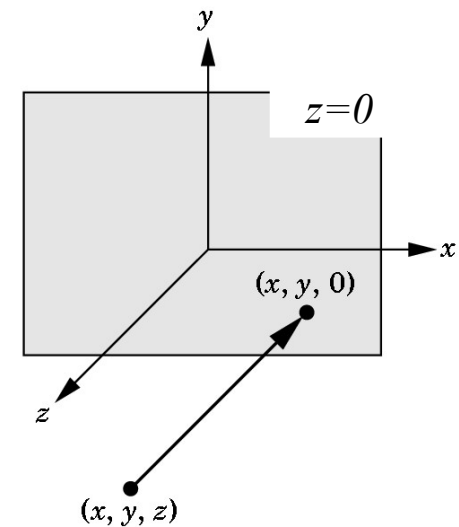
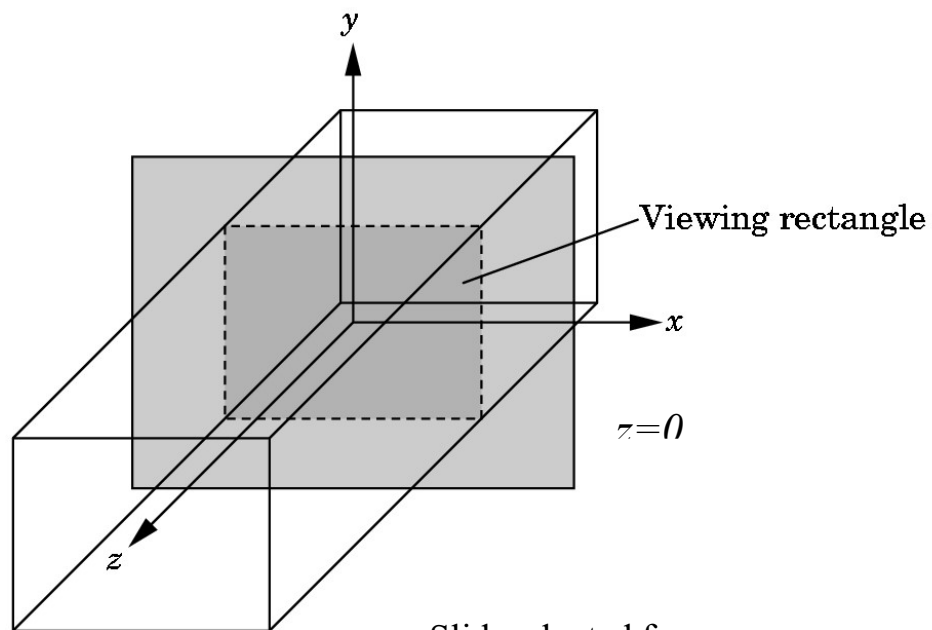
# WebGL Default View

- The default viewing volume is a box centered at the origin with sides of length 2
- This coordinate system is sometimes referred to as clip coordinates



# Orthographic Projection

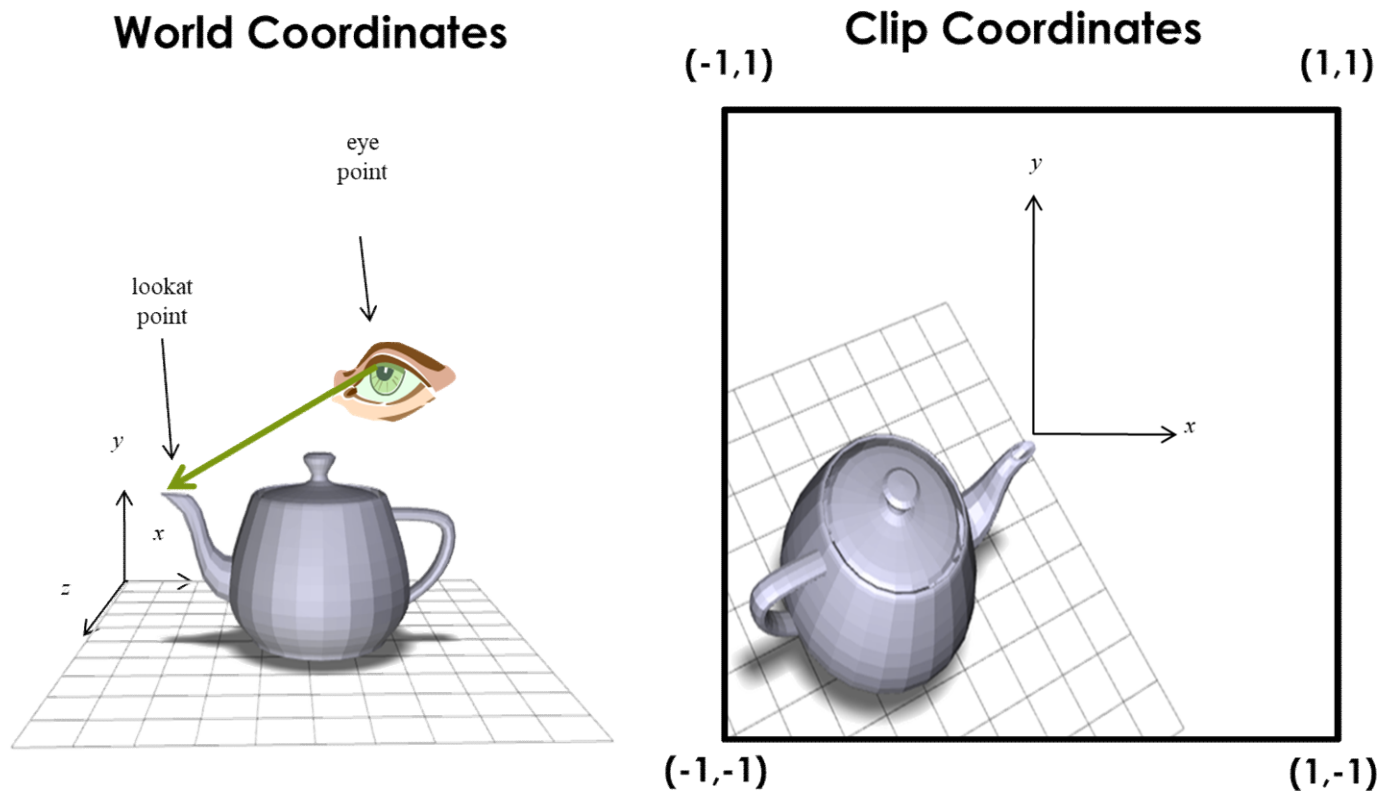
In the default orthographic view, points are projected forward along the  $z$  axis onto the plane  $z=0$



Slide adapted from  
Angel and Shreiner: Interactive Computer Graphics  
7E © Addison-Wesley 2015

# More Complicated Views Are Possible

- We'll see how to set up a virtual camera in future lectures
- Essentially just uses transformations to squish a general view volume into the WebGL default view volume



---

## Exercise: Where is the default eyepoint?

- ▣ Play around with this code:  
<http://jsfiddle.net/2x03hdc8/>
- ▣ Change the locations of the boxes to try to find the default eye (or camera) location for WebGL

---

# What Should You Know?

- ▣ General principles of rasterization
  - ▣ Pipeline model of a rasterization engine
  - ▣ What a vertex shader does
  - ▣ What a fragment shader does
  - ▣ Difference between rasterization and ray-tracing
-