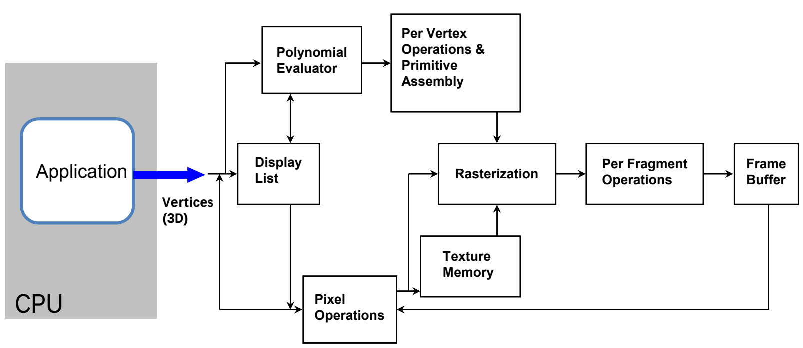
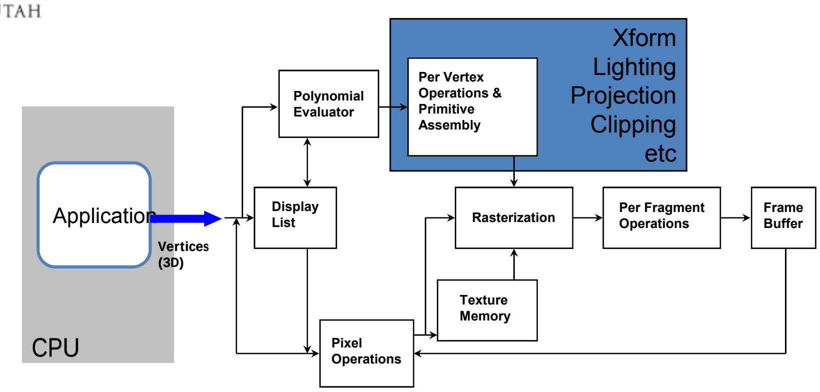
Introduction to WebGL

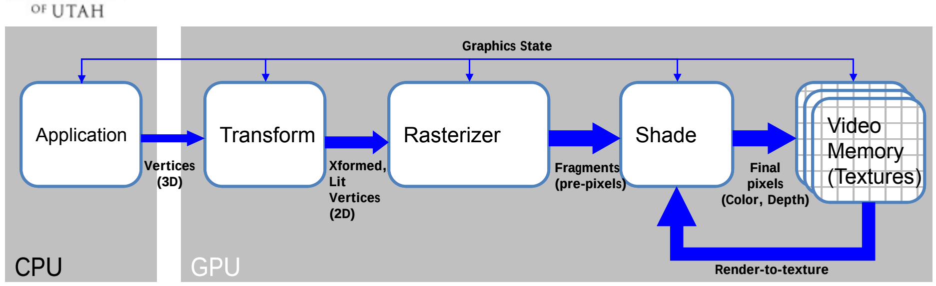
Traditional Graphics Pipeline (Fixed Function OpenGL)



Traditional Graphics Pipeline



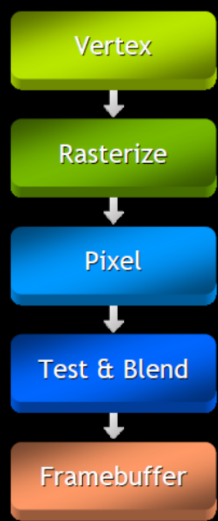
Traditional Graphics Pipeline



A simplified graphics pipeline

- Note that pipe widths vary
- Many caches, FIFOs, and so on not shown

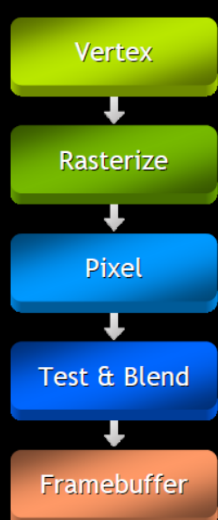
The Graphics Pipeline

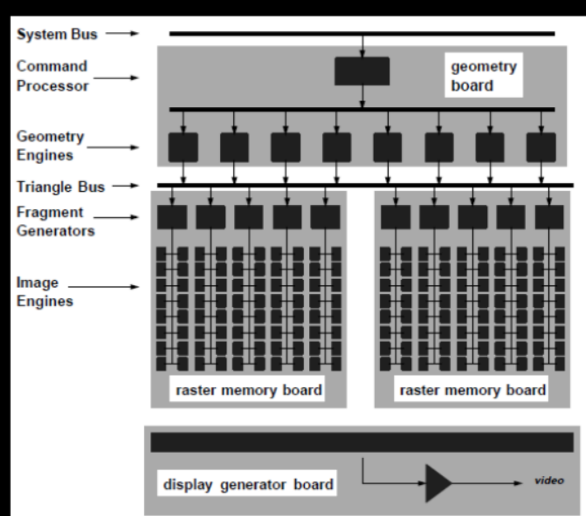


- Key abstraction of real-time graphics
- Hardware used to look like this
- Distinct chips/boards per stage
- Fixed data flow through pipeline

Beyond Programmable Shading: In Action

SGI RealityEngine (1993)

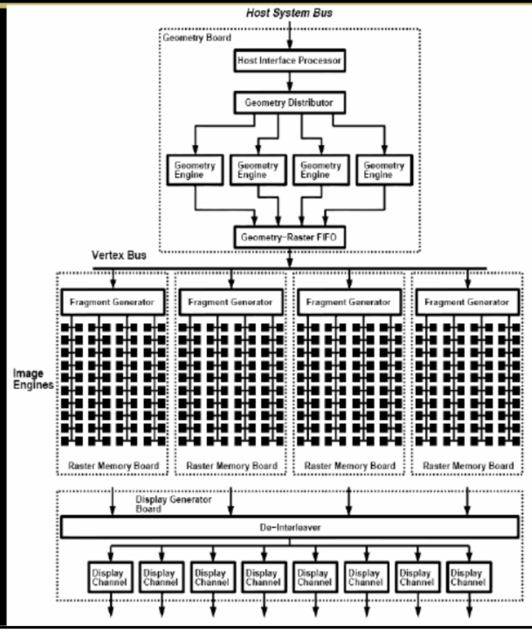




Kurt Akeley. RealityEngine Graphics. In *Proc. SIGGRAPH '93*. ACM Press, 1993.

SGI InfiniteReality (1997)

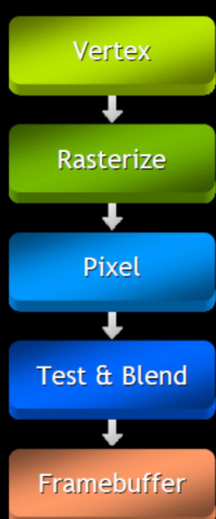
SIGGRAPH2008



Montrym, Baum, Dignam, & Migdal. InfiniteReality: A real-time graphics system. In *Proc. SIGGRAPH '97*. ACM Press, 1997.

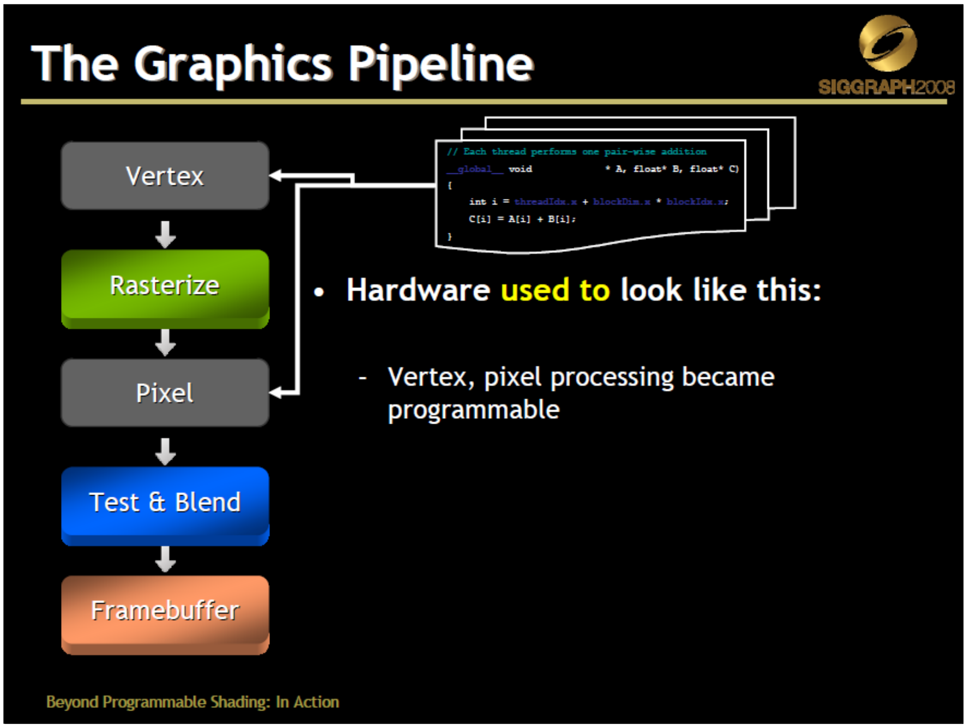
The Graphics Pipeline

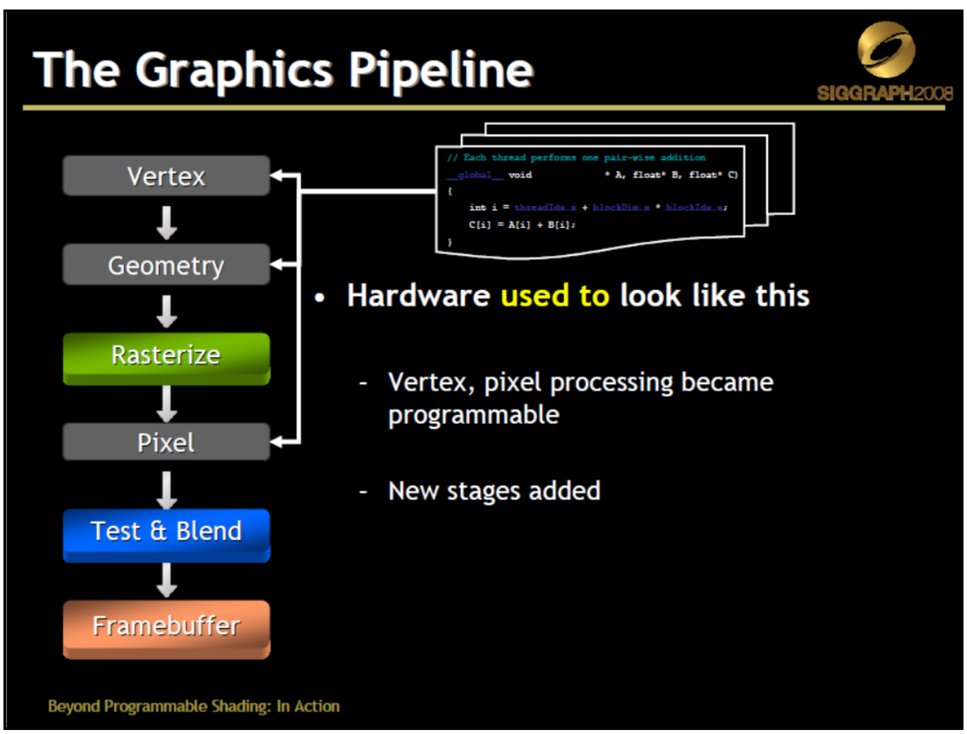
SIGGRAPH2008

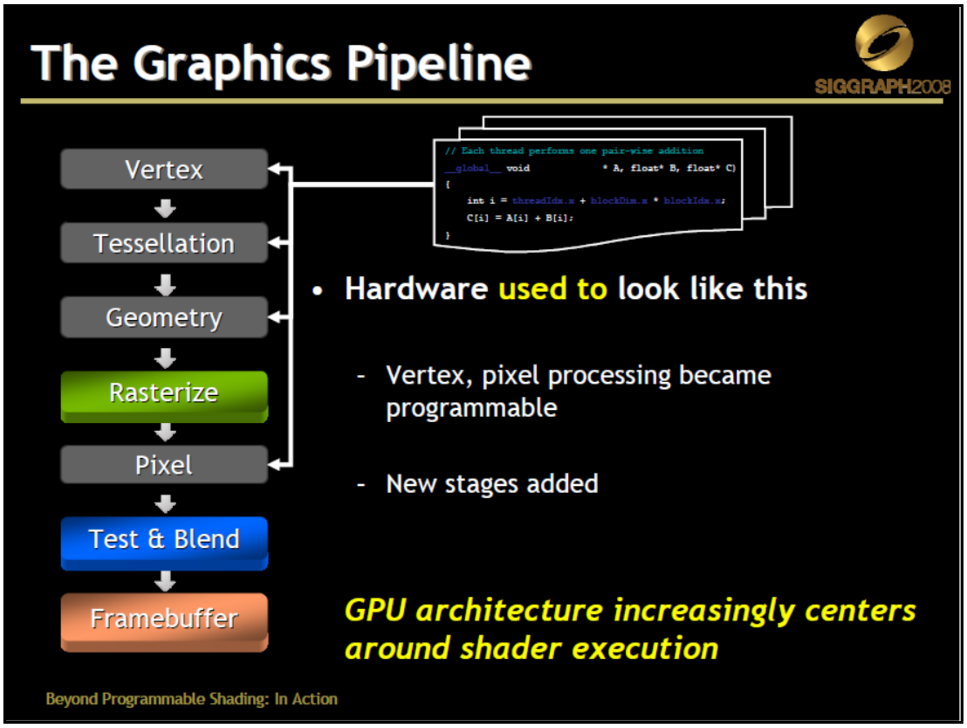


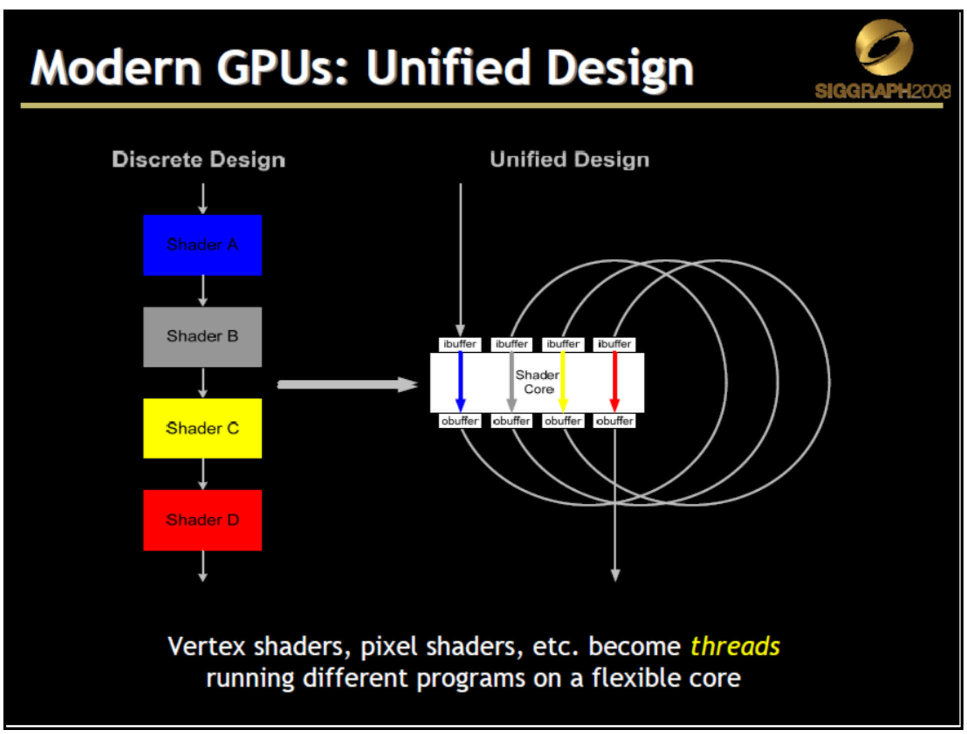
- **Remains a useful abstraction**
- **Hardware used to look like this**

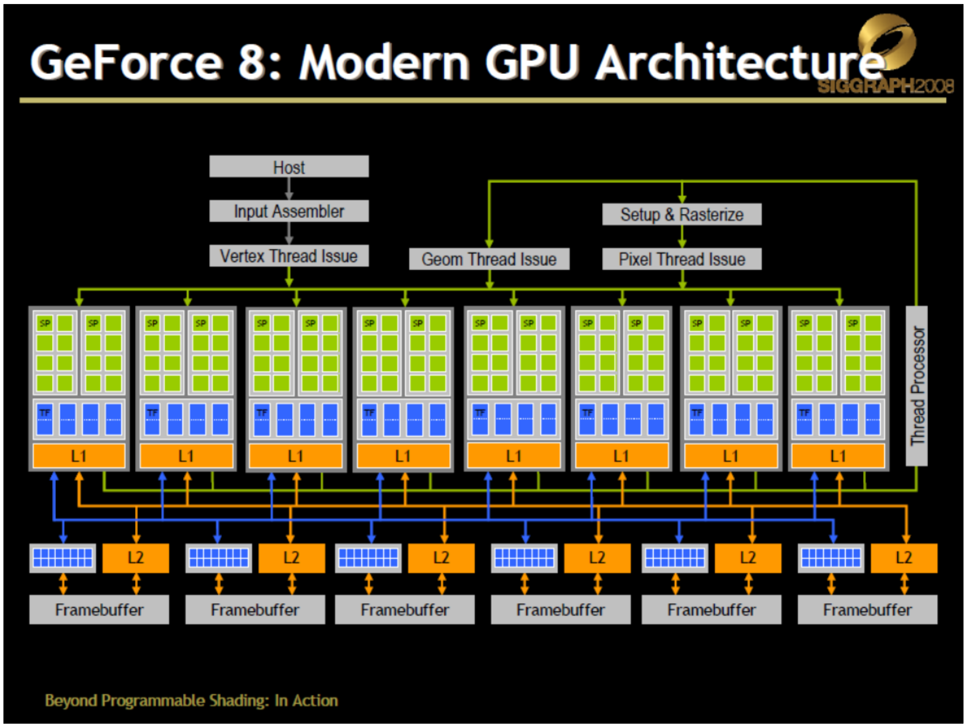
Beyond Programmable Shading: In Action

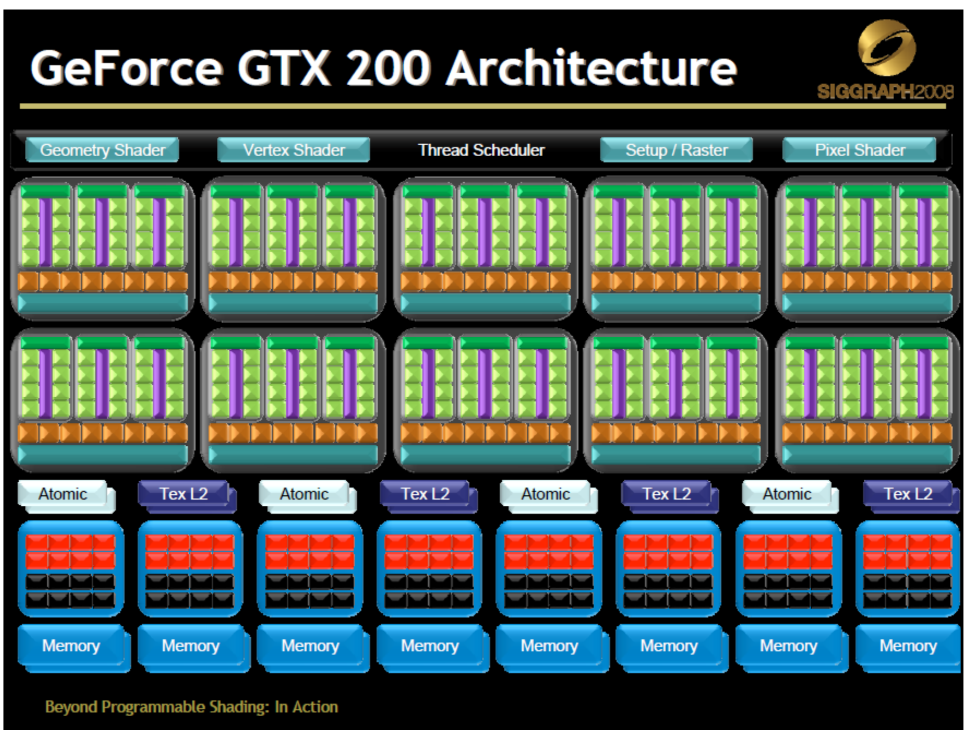

The Graphics Pipeline
SIGGRAPH2008
Vertex
Rasterize
Pixel
Test & Blend
Framebuffer
// Each thread performs one pair-wise addition
__global__ void * A, float* B, float* C)
{
int i = threadIdx.x + blockDim.x * blockIdx.x;
C[i] = A[i] + B[i];
}
• Hardware used to look like this:
- Vertex, pixel processing became programmable
Beyond Programmable Shading: In Action

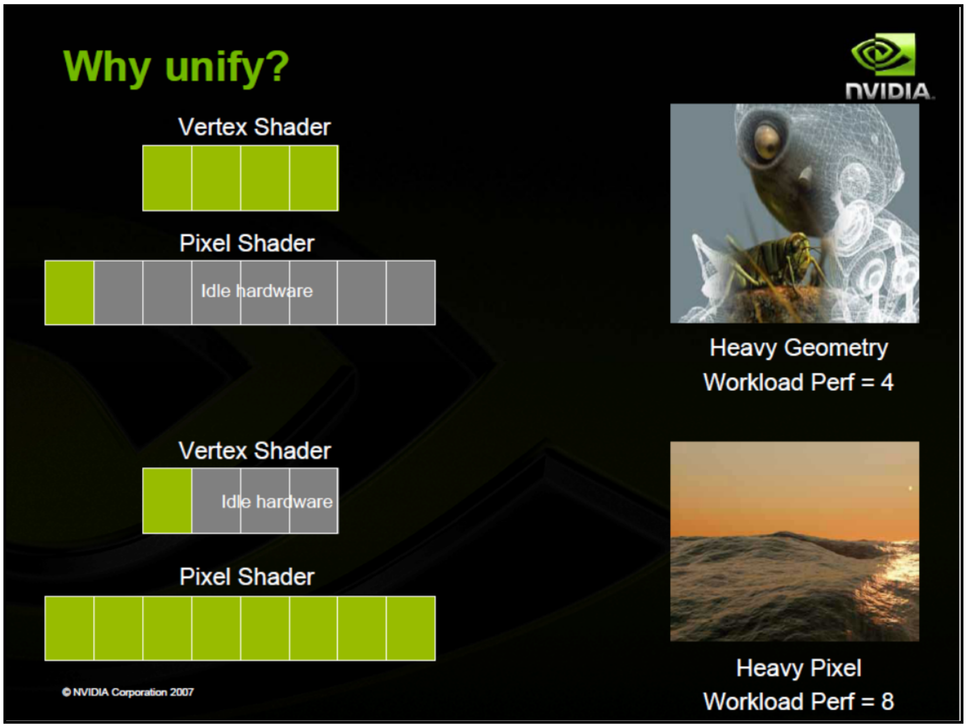

The Graphics Pipeline
SIGGRAPH2008
Vertex
Geometry
Rasterize
Pixel
Test & Blend
Framebuffer
// Each thread performs one pair-wise addition
__global__ void * A, float* B, float* C)
{
int i = threadIdx.x + blockDim.x * blockIdx.x;
C[i] = A[i] + B[i];
}
• Hardware used to look like this
- Vertex, pixel processing became programmable
- New stages added
Beyond Programmable Shading: In Action

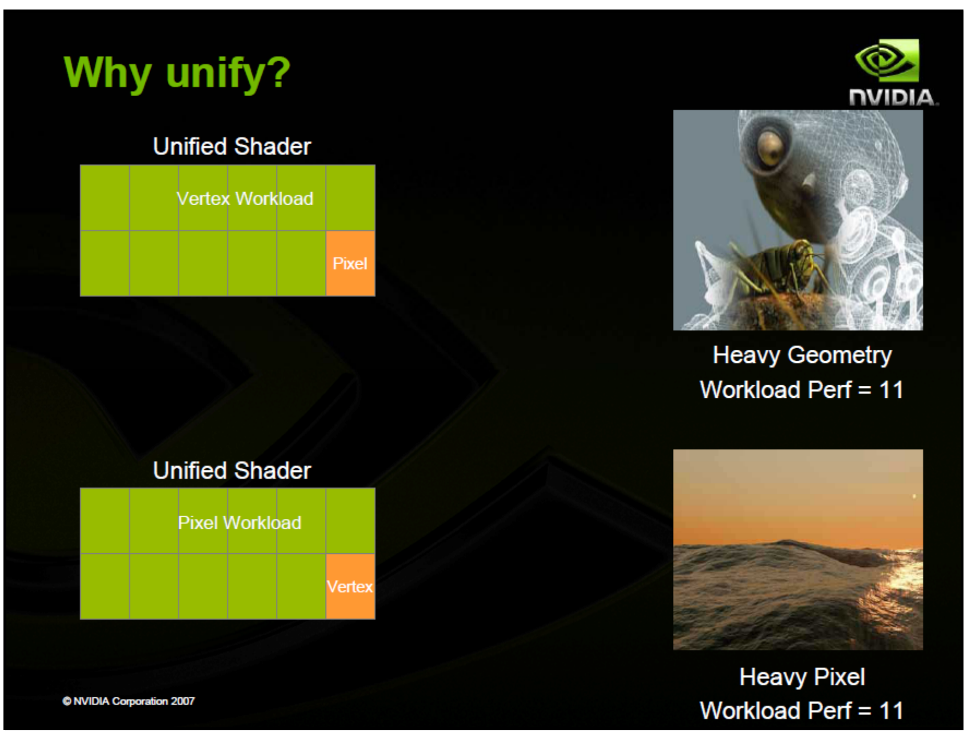

The Graphics Pipeline
SIGGRAPH2008
Vertex
Tessellation
Geometry
Rasterize
Pixel
Test & Blend
Framebuffer
// Each thread performs one pair-wise addition
__global__ void * A, float* B, float* C)
{
int i = threadIdx.x + blockDim.x * blockIdx.x;
C[i] = A[i] + B[i];
}
• Hardware used to look like this
- Vertex, pixel processing became programmable
- New stages added
GPU architecture increasingly centers around shader execution
Beyond Programmable Shading: In Action

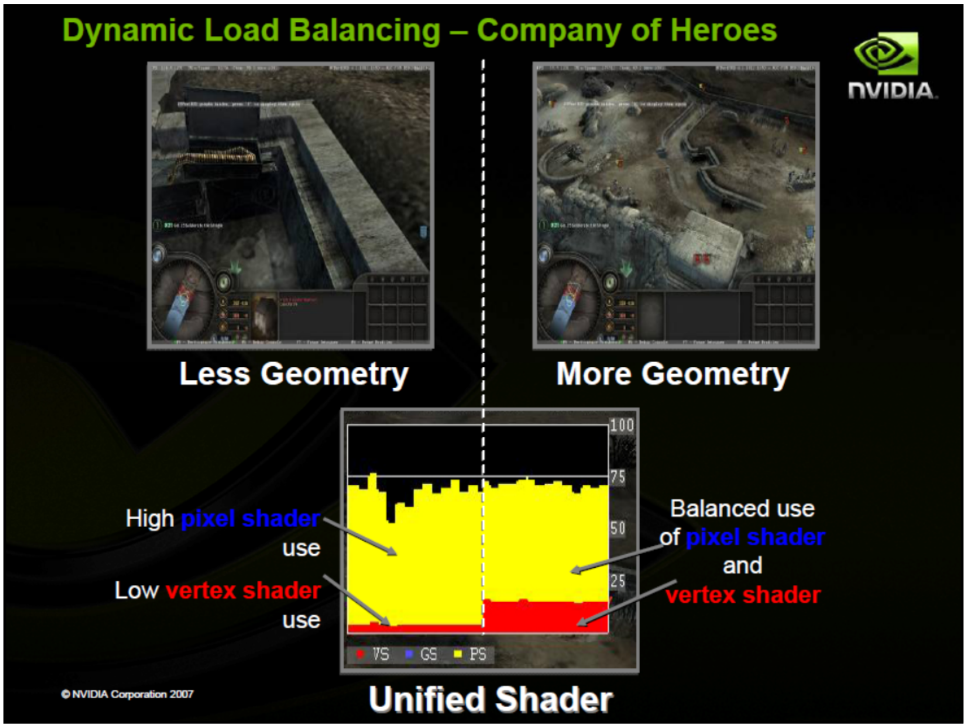

Modern GPUs: Unified Design
SIGGRAPH2008
Discrete Design
Unified Design
Shader A
Shader B
Shader C
Shader D
ibuffer
ibuffer
ibuffer
ibuffer
Shader Core
obuffer
obuffer
obuffer
obuffer
Vertex shaders, pixel shaders, etc. become threads running different programs on a flexible core

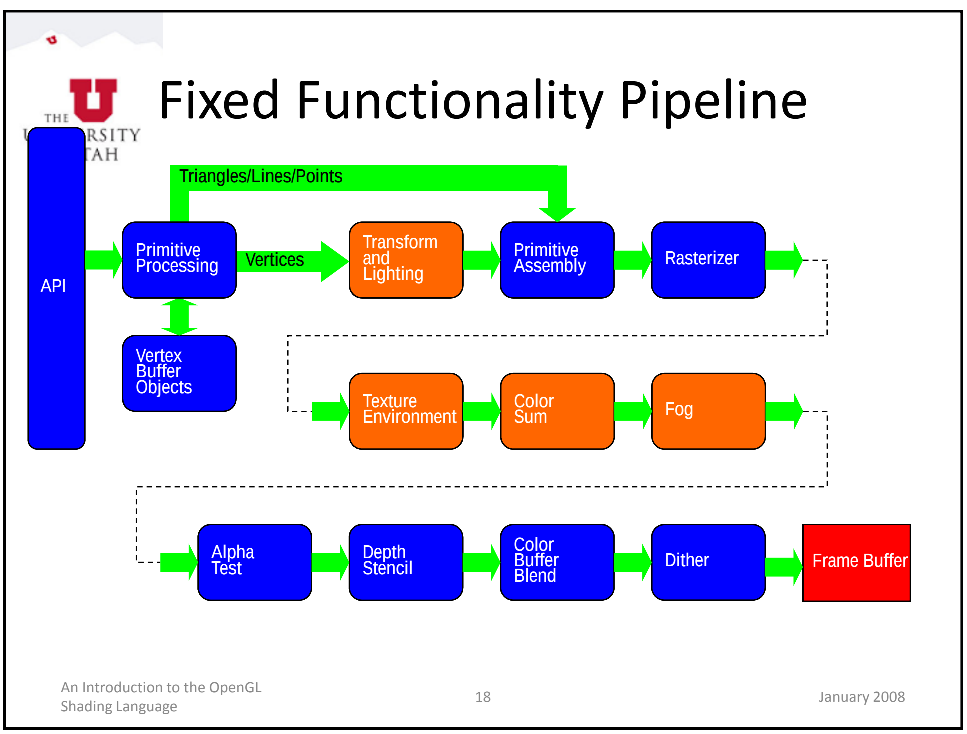

GeForce 8: Modern GPU Architecture
SIGGRAPH2008
Host
Input Assembler
Vertex Thread Issue
Setup & Rasterize
Geom Thread Issue
Pixel Thread Issue
Thread Processor
L1
L2
Framebuffer
Beyond Programmable Shading: In Action


GeForce GTX 200 Architecture
SIGGRAPH2008
Geometry Shader
Vertex Shader
Thread Scheduler
Setup / Raster
Pixel Shader
Atomic
Tex L2
Memory
Beyond Programmable Shading: In Action

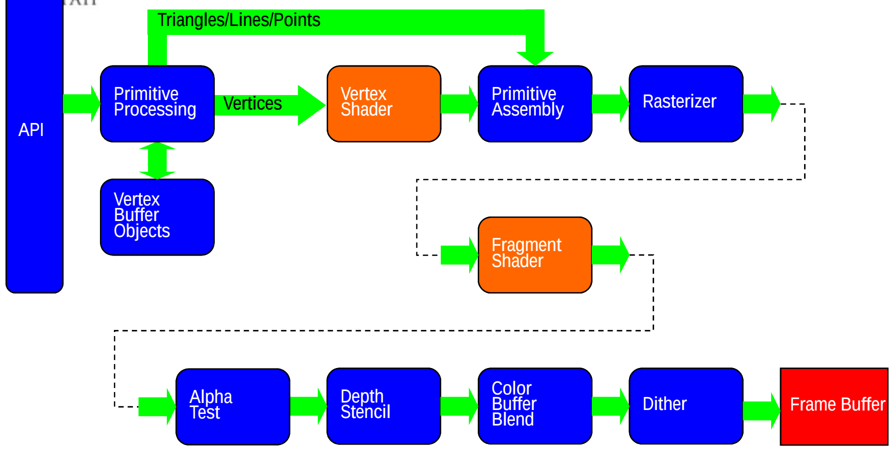

Why unify?
NVIDIA
Vertex Shader
Pixel Shader
Idle hardware
Heavy Geometry
Workload Perf = 4
Vertex Shader
Idle hardware
Pixel Shader
Heavy Pixel
Workload Perf = 8
© NVIDIA Corporation 2007


Why unify?
NVIDIA
Unified Shader
Vertex Workload
Pixel
Heavy Geometry
Workload Perf = 11
Unified Shader
Pixel Workload
Vertex
Heavy Pixel
Workload Perf = 11
© NVIDIA Corporation 2007


Dynamic Load Balancing – Company of Heroes
NVIDIA
Less Geometry
More Geometry
100
75
50
25
VS GS PS
High pixel shader use
Low vertex shader use
Balanced use of pixel shader and vertex shader
© NVIDIA Corporation 2007
Unified Shader

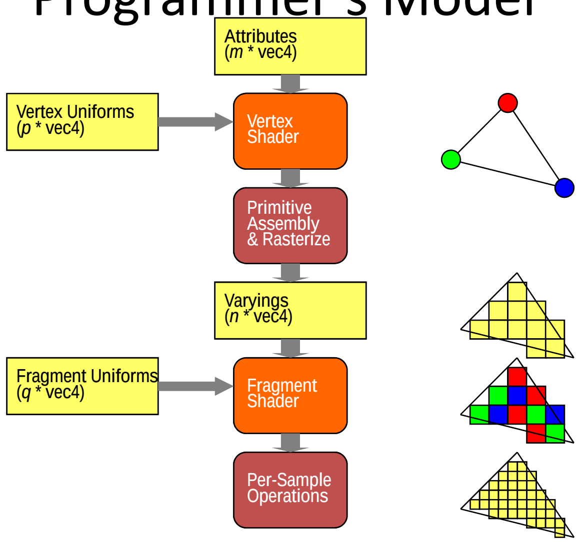

Fixed Functionality Pipeline
API
Triangles/Lines/Points
Primitive Processing
Vertices
Transform and Lighting
Primitive Assembly
Rasterizer
Vertex Buffer Objects
Texture Environment
Color Sum
Fog
Alpha Test
Depth Stencil
Color Buffer Blend
Dither
Frame Buffer
An Introduction to the OpenGL Shading Language
18
January 2008

Programmable Shader Pipeline



An Introduction to the OpenGL Shading Language 19 January 2008

Programmer's Model



An Introduction to the OpenGL Shading Language 20 January 2008

Simple Vertex Shader

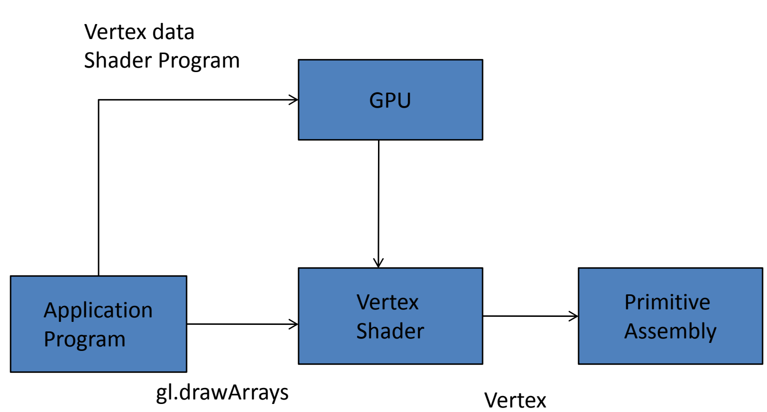
input from application

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

must link to variable in application

built in variable

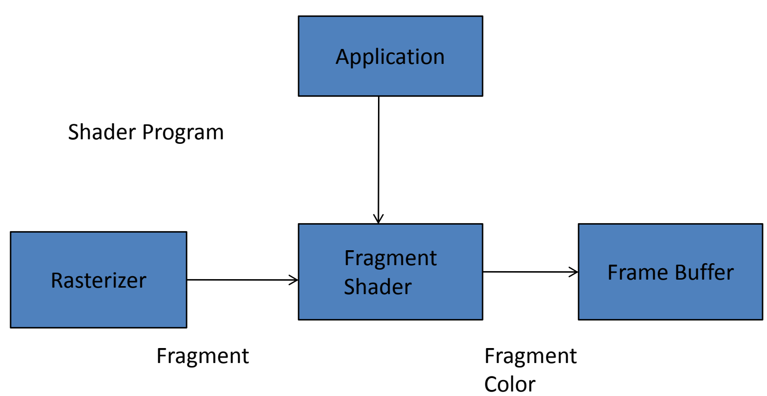
Vertex Execution Model



Simple Fragment Program

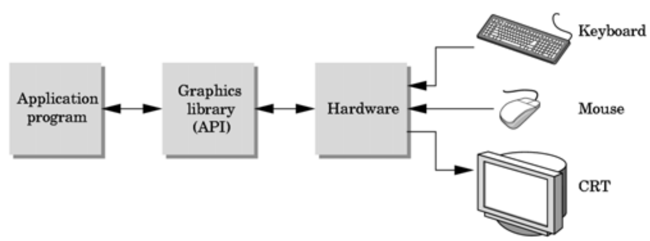
```
precision mediump float;
void main(void)
{
  gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```


Fragment Execution Model



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 (openGL state)
  – 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) Immediate Mode

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

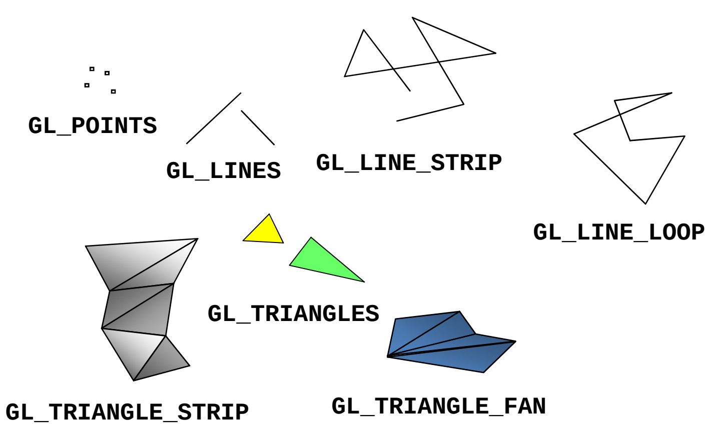
Example (GPU based) vertex arrays

- 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),
];
```

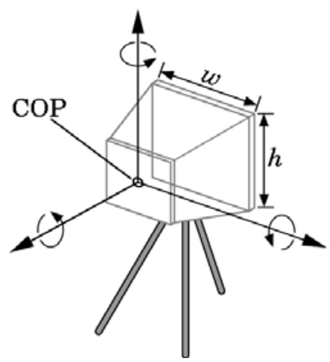
- Send array to GPU
- Tell GPU to render as triangle

WebGLPrimitives



Camera Specification

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



Coordinate Systems

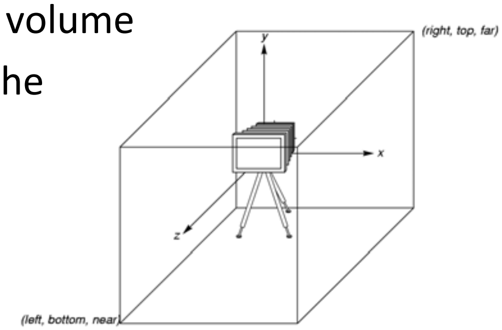
- The units in **`points`** are determined by the application and are called *model* or *problem coordinates*
- Viewing specifications usually are in world coordinates
- Eventually pixels will be produced in *window coordinates*
- WebGL also uses some internal representations that usually are not visible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

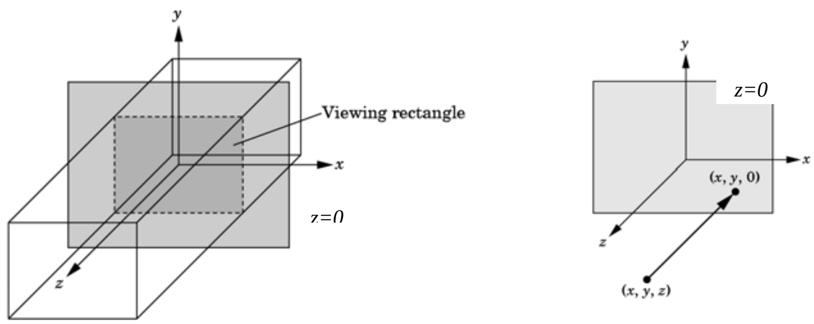
WebGL Camera

- 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



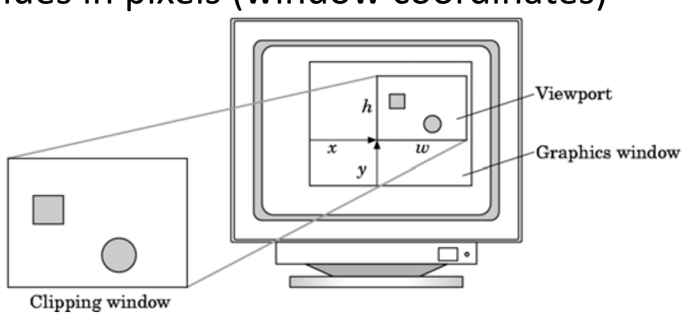
Orthographic Viewing

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)



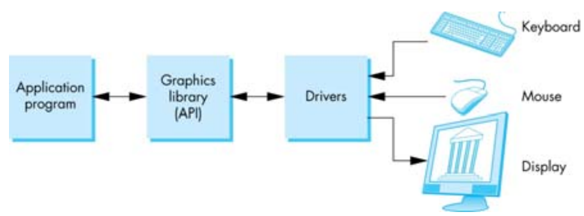
Transformations and Viewing

- 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

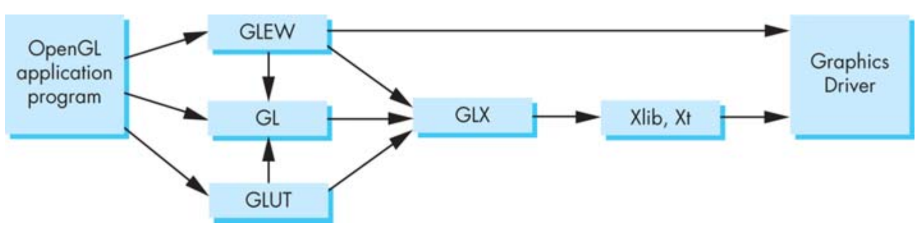
Lights and Materials

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

OpenGL Architecture



Software Organization



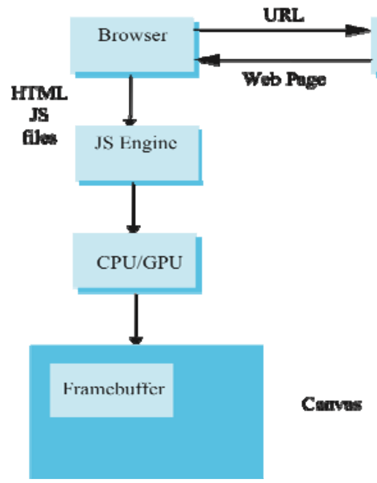
It used to be easy

```
#include <GL/glut.
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



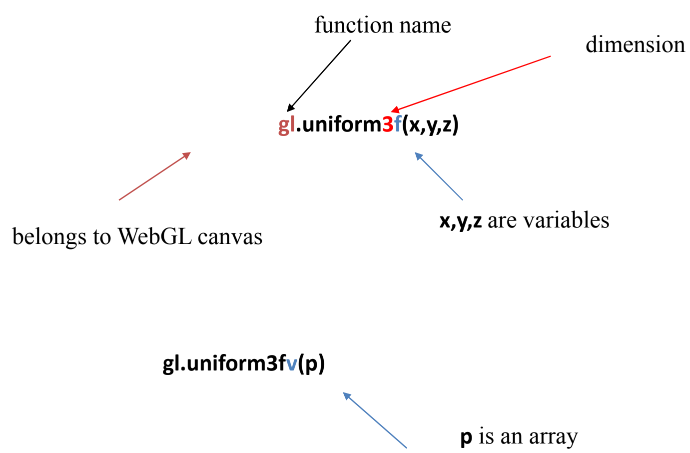
Event Loop

- 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



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)`**

Vertex Shader Applications

- Moving vertices
  - Morphing
  - Wave motion
  - Fractals
- Lighting
  - More realistic models
  - Cartoon shaders

Fragment Shader Applications

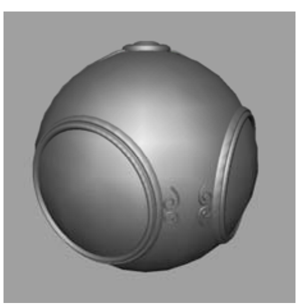
Per fragment lighting calculations

per vertex lighting

per fragment lighting

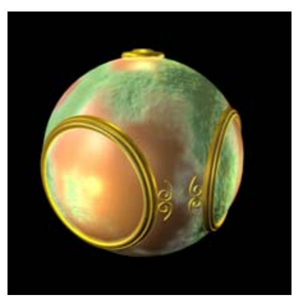
Fragment Shader Applications

Texture mapping



smooth shading

environment mapping



bump mapping

Writing Shaders

- First programmable shaders were programmed in an assembly-like manner
- OpenGL extensions added functions for vertex and fragment shaders
- Cg (C for graphics) C-like language for programming shaders
  - Works with both OpenGL and DirectX
  - Interface to OpenGL complex
- OpenGL Shading Language (GLSL)

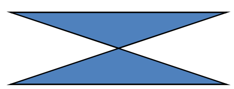
WebGL and GLSL

- 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
- Lots of action happens in shaders
- Job of application is to get data to GPU

52

Polygon Issues

- WebGL will only display triangles
  - Simple: edges cannot cross
  - Convex: All points on line segment between two points in a polygon are also in the polygon
  - Flat: all vertices are in the same plane
- Application program must tessellate a polygon into triangles (triangulation)
- OpenGL 4.1 contains a tessellator but not WebGL



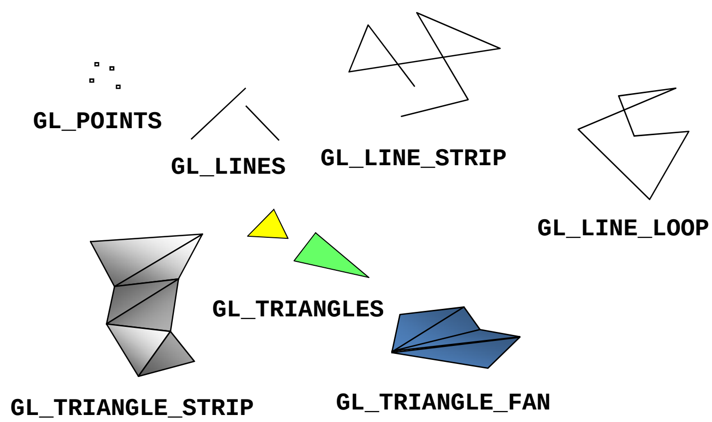
nonsimple polygon

nonconvex polygon

Polygon Testing

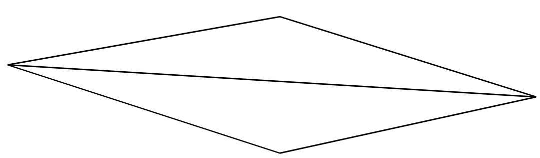
- Conceptually simple to test for simplicity and convexity
- Time consuming
- Earlier versions assumed both and left testing to the application
- Present version only renders triangles
- Need algorithm to triangulate an arbitrary polygon

WebGLPrimitives



Good and Bad Triangles

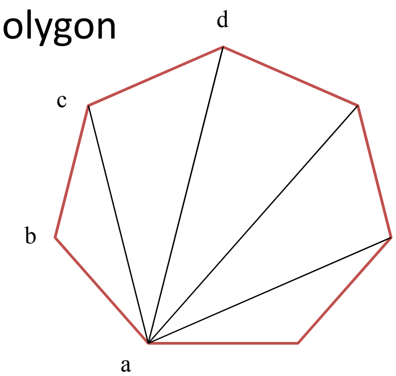
- Long thin triangles render poorly



- Equilateral triangles render well
- Maximize minimum angle
- Delaunay triangulation for unstructured points

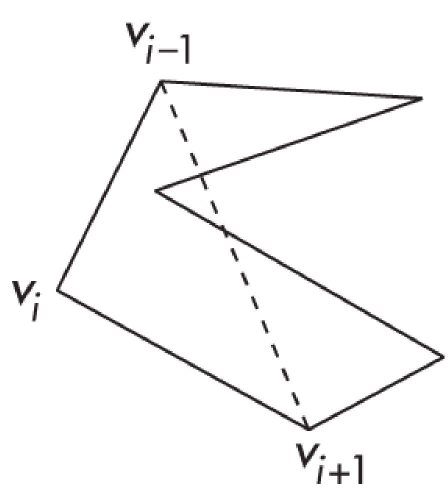
Triangularization

- Convex polygon



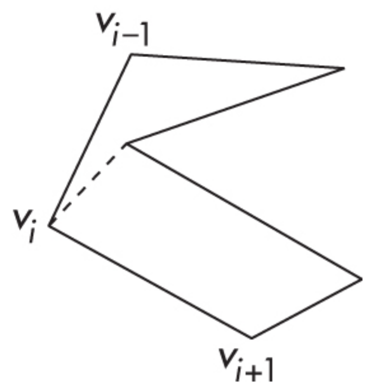
- Start with abc, remove b, then acd, ….

Non-convex (concave)



Recursive Division

- Find leftmost vertex and split

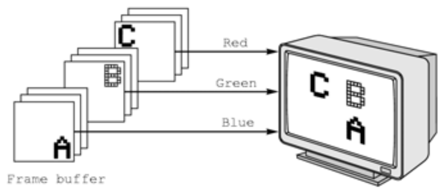


Attributes

- 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
    - Display vertices
- Only a few (gl_PointSize) are supported by WebGL functions

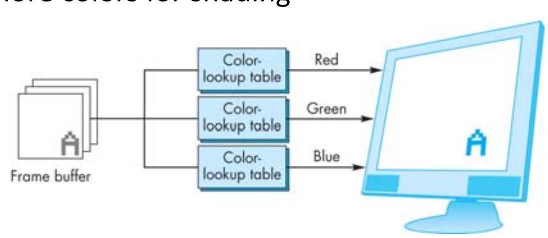
RGB color

- 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



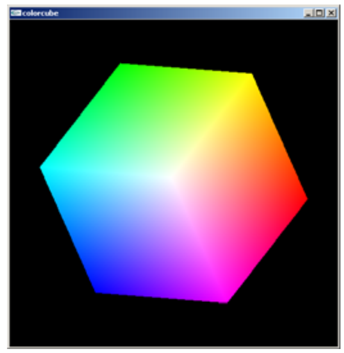
Indexed Color

- Colors are indices into tables of RGB values
- Requires less memory
  - indices usually 8 bits
  - not as important now
    - Memory inexpensive
    - Need more colors for shading



Smooth Color

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

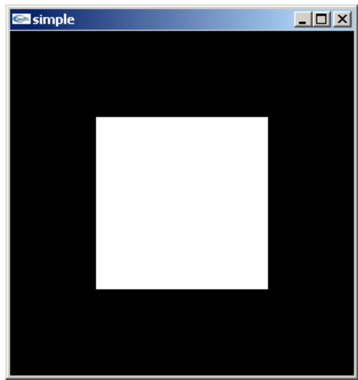


Setting Colors

- 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

A OpenGL Simple Program

Generate a square on a solid background



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)

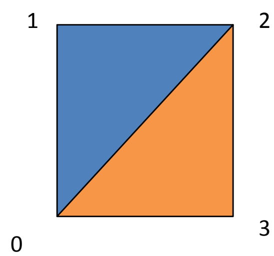
```
// Configure WebGL

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

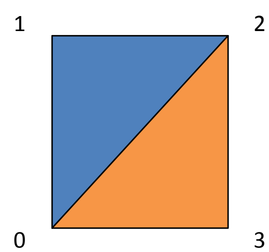
square.js (cont)

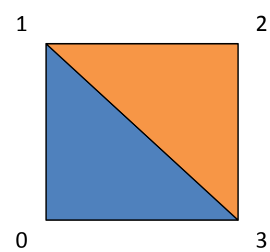
```
    render();
};

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



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
```