CSE 5542 - Real Time Rendering
Week 2
Graphics Processing
Physical Approaches
Projection-Based
Projection

3D objects -> 2D image
- Perspective
- Parallel/Orthographic

Orthographic Projection: Camera positioned infinitely far away at $z = \infty$

Perspective Projection: The camera's view frustum is specified via 4 view parameters: fovy, aspect, zNear and zFar.
The Hardware

Diagram showing the flow of data from inputs to the CPU, then to the GPU, and finally to the display. The components include:

- **Inputs**
- **CPU** (Central Processing Unit)
- **Main Memory (RAM)**
- **Graphics Memory (VRAM)**
- **Frame Buffer**
- **Graphics Subsystem**
- **Display**
The API/System

Application program → Graphics library (API) → Hardware

Connected to:
- Keyboard
- Mouse
- CRT
The Graphics Pipeline
Object & Primitive & Vertex

http://www.3dcadbrowser.com/download.aspx?3dmodel=27814
Object & Triangles & Vertices

N=100  
U=8156.7

N=200  
U=33917.1

N=300  
U=77600.6

N=400  
U=139322.3

N=500  
U=230354.4

N=600  
U=317118.3

Primitives

GL_POINTS  GL_LINES  GL_LINE_STRIP  GL_LINE_LOOP

GL_POLYGON  GL_TRIANGLES  GL_TRIANGLE_STRIP

GL_TRIANGLE_FAN  GL_QUADS  GL_QUAD_STRIP

OpenGL Primitives
Example (old style)

```c
glBegin(GL_POLYGON)
    glVertex3f(0.0, 0.0, 0.0);
    glVertex3f(0.0, 1.0, 0.0);
    glVertex3f(0.0, 0.0, 1.0);
    glVertex3f(0.0, 0.0, 1.0);
    glVertex3f(0.0, 0.0, 1.0);
    glVertex3f(0.0, 1.0, 0.0);
    glVertex3f(0.0, 0.0, 0.0);
    glEnd();
```
Example (GPU based)

- Put geometric data in an array
  ```
  vec3 points[3];
  points[0] = vec3(0.0, 0.0, 0.0);
  points[1] = vec3(0.0, 1.0, 0.0);
  points[2] = 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
Materials

Optical properties

– Absorption/Reflection: color Scattering
  • Diffuse
  • Specular
  • Transparent

– Texture

– …
Lights

Types

– Point sources vs distributed sources
– Spot lights
– Near and far sources
– Color properties
Vertex Processing

Polygon Soup

Raw Vertices & Primitives

Transformed Vertices & Primitives

Fragment Processor (Programmable)

Rasterizer

Fragments

Fragment Processor (Programmable)

Processed Fragments

Output Merging

3D

2D array of color-values

Display
Vertex Processing

- Define object representations from one coordinate system to another
  - Object coordinates
  - World Coordinates
  - Camera (eye) coordinates
  - Screen coordinates

- Enter Linear algebra – Transformations
- Material properties
World
Primitive Assembly

Vertices collected into geometric objects

- Line segments
- Polygons
- Curves and surfaces
Clipping

Bounding Sphere

Bounding Box

View volume

Front clipping plane

Back clipping plane

COP
Rasterization

- Output are fragments
- Fragments == potential pixels
  - Location in frame buffer
  - Color and depth attributes at vertices
  - Hidden surface removal?
- Vertex attributes are interpolated over objects
Fragment Processing

A *primitive* is formed by one or more *vertices*. Vertices are not grid-aligned.

Grid-aligned *fragments* are interpolated from vertices.

All primitives are merged to produce 2D *pixels* on the display.
The Graphics Pipeline

Application Stage → 3D Triangles → Geometry Stage → 2D Triangles → Rasterization Stage → Pixels

For each triangle vertex:
- Transform 3D position into screen position
- Compute attributes

For each triangle:
- Rasterize triangle
- Interpolate vertex attributes across triangle
- Shade pixels
- Resolve visibility
What is Missing?
Not Quite?
Next ?
My Desktop
# My Desktop

**Chipset Model:** AMD Radeon HD 6770M  
**Type:** GPU  
**Bus:** PCIe  
**PCIe Lane Width:** x16  
**VRAM (Total):** 512 MB  
**Vendor:** ATI (0x1002)  
**Device ID:** 0x6740  
**Revision ID:** 0x0000  
**ROM Revision:** 113-C0170F-170  
**EFI Driver Version:** 01.00.544  

### Displays:

**iMac:**  
**Display Type:** LCD  
**Resolution:** 2560 x 1440  
**Pixel Depth:** 32-Bit Color (ARGB8888)  
**Main Display:** Yes  
**Mirror:** Off  
**Online:** Yes  
**Built-In:** Yes
AMD RADEON™

HD 6770 GPU
ENGINE CLOCK Up to 850MHz
MEMORY 512MB or 1GB DDR3 or GDDR5
MEMORY CLOCK 1200MHz
MEMORY BANDWIDTH 76.8 GB/s (maximum)
SINGLE PRECISION COMPUTE POWER 1.36 TFLOPs
TERASCALE 2 UNIFIED PROCESSING ARCHITECTURE
800 Stream Processors
40 Texture Units
64 Z/Stencil ROP Units
16 Color ROP Units
BUS INTERFACE PCI Express 2.1 x16
OPENGL 4.1 SUPPORT Yes

IMAGE QUALITY ENHANCEMENT TECHNOLOGY
Up to 24x multi-sample and super-sample anti-aliasing modes
Adaptive anti-aliasing
16x angle independent anisotropic texture filtering
128-bit floating point HDR rendering

CUTTING-EDGE INTEGRATED DISPLAY SUPPORT
Integrated DisplayPort Output
Max resolution: 2560x1600 per display
HDMI® (With 3D, Deep Color and x.v.Color™)
Max resolution: 1920x1200
Integrated Dual-link DVI with HDCP
Max resolution: 2560x1600
Integrated VGA
Max resolution: 2048x1536

INTEGRATED HD AUDIO CONTROLLER
Output protected high bit rate 7.1 channel surround sound over HDMI or DisplayPort with no additional cables required
Supports AC-3, AAC, Dolby TrueHD and DTS Master Audio formats

AMD TECHNOLOGIES
AMD Eyefinity multiview technology
Native support for up to 5 simultaneous displays
Independent resolutions, refresh rates, color controls and video overlays
Display grouping
Combine multiple displays to behave like a single large display
AMD App Acceleration
OpenCL 1.1 Support
DirectCompute 11
Accelerated video encoding, transcoding and upscaling
UVD 2 dedicated video playback accelerator
H.264
VC-1
MPEG-2
H.264 MVC (Blu-ray 3D)
Adobe Flash
Enhanced Video Quality features
Advanced post-processing and scaling
Dynamic contrast enhancement and color correction
Brighter whites processing (Blue Stretch)
Independent video gamma control
Dynamic video range control
Dual-stream HD (1080p) playback support
DXVA 1.0 & 2.0 support

AMD HD3D technology
Steroscopic 3D display/glasses support
Blu-ray 3D support
Steroscopic 3D gaming
3rd party Stereoscopic 3D middleware software support
AMD CrossFire™ multi-GPU technology
Dual GPU scaling
AMD PowerPlay™ power management technology
Dynamic power management with low power idle state
Ultra-low power state support for multi-GPU configurations

AMD Catalyst™ software and HD video configuration software
Unified graphics display drivers
Certified for Windows 7, Windows Vista, and Windows XP
AMD Catalyst™ Control Center
Software application and user interface for setup, configuration and accessing special features of AMD Radeon products.
Computer Graphics Hardware: An Overview

Many Thanks to Prof. Han-wei Shen, CSE, OSU
Graphics System

- Input devices
- CPU/Memory
- GPU

Monitor
Raster Graphics System

Frame buffer

video controller

DAC

A
To Note

- Raster: An array of picture elements
- Based on raster-scan TV technology
- The screen & rendering consists of discrete pixels
- Each pixel has a small display area
The Frame Buffer
Frame Buffer

- Low-latency memory to hold pixel attributes
  - color, alpha, depth, stencil mask, who-knows-what
- Performance depends on
  - Size: screen resolution
  - Depth: color level
  - Speed: refresh speed
Depth

- 1 bit/pixel: black and white
- 8 bits/pixel: 256 levels of gray or color palette index
- 24 bits/pixel: 16 million colors
Image Digitization - Recap

**Sampling**: Resolution

**Quantization**: Measured Value
Image Digitization-Recap

Sampling

Quantization
The Architecture
(A way too) Simple Graphik System

Frame buffer is part of main memory

Problem?
Dedicated memory

Video memory: On-board frame buffer: much faster to access
Graphics Accelerator

A dedicated processor for graphics processing
Graphics Bus Interface

PCI based technology

- Graphics Memory/Frame buffer
- Graphics Processor
- Scan Controller
- Other Peripherals
- PCIe (8 GB/s)
- System Bus
- CPU
- Main Memory
Graphics Accelerators
(Massively) Parallel Processors
Stream Processing

[Diagram showing SISD and SIMD concepts with instruction and data flow]

[Diagram showing a stream processing model with kernels, input streams, and output streams]
The Main Drivers

I ♥ Gaming

entertainment UNLIMITED
GPU = General Purpose Units!
The Existentialist GPU

Tegra 2 GPU pipeline: Horsepower

- 8 shader cores
  - 4 pixel shader cores
  - 4 vertex shader cores
- 5x CSAA
- 16x anisotropic texture filtering

http://www.anandtech.com/show/4225/the-ipad-2-review/5
Multi-Core Galore

CPU
MULTIPLE CORES

GPU
THOUSANDS OF CORES
Largest Chip on Mother Board

Pentium Extreme Edition 840
- 3.2 GHz Dual Core
- 230M Transistors
- 90nm process
- 206 mm^2
- 2 x 1MB Cache
- 25.6 GFlops

GeForce 7800 GTX
- 430 MHz
- 302M Transistors
- 110nm process
- 326 mm^2
- 313 GFlops (shader)
- 1.3 TFlops (total)
Evolution of Performance

<table>
<thead>
<tr>
<th>Year</th>
<th>Mvertices/s</th>
<th>Mtransistors</th>
<th>Mpixels/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>1995</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>1996</td>
<td>100</td>
<td>100</td>
<td>1000</td>
</tr>
<tr>
<td>1997</td>
<td>1000</td>
<td>1000</td>
<td>10000</td>
</tr>
<tr>
<td>1998</td>
<td>10000</td>
<td>10000</td>
<td>100000</td>
</tr>
<tr>
<td>1999</td>
<td>100000</td>
<td>100000</td>
<td>1000000</td>
</tr>
<tr>
<td>2000</td>
<td>1000000</td>
<td>1000000</td>
<td>10000000</td>
</tr>
<tr>
<td>2001</td>
<td>10000000</td>
<td>10000000</td>
<td>100000000</td>
</tr>
<tr>
<td>2002</td>
<td>100000000</td>
<td>100000000</td>
<td>1000000000</td>
</tr>
<tr>
<td>2003</td>
<td>1000000000</td>
<td>1000000000</td>
<td>10000000000</td>
</tr>
<tr>
<td>2004</td>
<td>10000000000</td>
<td>10000000000</td>
<td>100000000000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Year</th>
<th>Video Memory</th>
<th>API</th>
</tr>
</thead>
<tbody>
<tr>
<td>1995</td>
<td>4 MB</td>
<td>DirectX 1, OpenGL 1.1</td>
</tr>
<tr>
<td>1996</td>
<td>32 MB</td>
<td>DirectX 2, DirectX 3, OpenGL 1.2</td>
</tr>
<tr>
<td>1997</td>
<td>64 MB</td>
<td>DirectX 5, OpenGL 1.3</td>
</tr>
<tr>
<td>1998</td>
<td>128 MB</td>
<td>DirectX 6, OpenGL 1.4</td>
</tr>
<tr>
<td>1999</td>
<td>256 MB</td>
<td>DirectX 7, OpenGL 1.5</td>
</tr>
<tr>
<td>2000</td>
<td>256 MB</td>
<td>DirectX 8, OpenGL 1.3</td>
</tr>
<tr>
<td>2001</td>
<td>512 MB</td>
<td>DirectX 9, OpenGL 1.5</td>
</tr>
<tr>
<td>2002</td>
<td>512 MB</td>
<td>DirectX 8, OpenGL 1.4</td>
</tr>
<tr>
<td>2003</td>
<td>512 MB</td>
<td>DirectX 9, OpenGL 1.5</td>
</tr>
<tr>
<td>2004</td>
<td>512 MB</td>
<td>DirectX 10, OpenGL 2.0</td>
</tr>
</tbody>
</table>
nVidia G80 GPU (2006)

- 128 streaming floating point processors @1.5Ghz
- 1.5 Gb Shared RAM with 86Gb/s bandwidth
- 500 Gflop on one chip (single precision)
nVidia G80 GPU

Application
Data Assembler
Vtx Thread Issue
Prim Thread Issue
Frag Thread Issue
Setup / Rstr / ZCull

Thread Processor

Application

Vertex assembly

Vertex operations

Primitive assembly

Primitive operations

Rasterization

Fragment operations

Framebuffer
nVidia Fermi GPU (2009)
# nVidia Fermi GPU (2009)

<table>
<thead>
<tr>
<th>GPU Feature</th>
<th>G80</th>
<th>GT200</th>
<th>Fermi</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transistors</td>
<td>681 million</td>
<td>1.4 billion</td>
<td>3.0 billion</td>
</tr>
<tr>
<td>CUDA Cores</td>
<td>128</td>
<td>240</td>
<td>512</td>
</tr>
<tr>
<td>Double Precision Floating Point Capability</td>
<td>None</td>
<td>30 FMA ops / clock</td>
<td>256 FMA ops / clock</td>
</tr>
<tr>
<td>Single Precision Floating Point Capability</td>
<td>128 MAD ops / clock</td>
<td>240 MAD ops / clock</td>
<td>512 FMA ops / clock</td>
</tr>
<tr>
<td>Special Function Units (SFUs) / SM</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Warp schedulers (per SM)</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Shared Memory (per SM)</td>
<td>16 KB</td>
<td>16 KB</td>
<td>Configurable 48 KB or 16 KB</td>
</tr>
<tr>
<td>L1 Cache (per SM)</td>
<td>None</td>
<td>None</td>
<td>Configurable 16 KB or 48 KB</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>None</td>
<td>None</td>
<td>768 KB</td>
</tr>
<tr>
<td>ECC Memory Support</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Concurrent Kernels</td>
<td>No</td>
<td>No</td>
<td>Up to 16</td>
</tr>
<tr>
<td>Load/Store Address Width</td>
<td>32-bit</td>
<td>32-bit</td>
<td>64-bit</td>
</tr>
</tbody>
</table>
nVidia Kepler GK110 (2012)

Architecture
- 7.1B Transistors
- 15 SMX units
- > 1 TFLOP FP64
- 1.5 MB L2 Cache
- 384-bit GDDR5
- PCI Express Gen3
CPU/GPU Performance Gap
Why are GPUs Fast?
Moore’s Law ++

Microprocessor Transistor Counts 1971-2011 & Moore’s Law

curve shows transistor count doubling every five years

Date of introduction

Transistor count
Modern GPU has more ALU’s

Figure 1-2. The GPU Devotes More Transistors to Data Processing
Stream Processing
Single Chip Design
The Scourge

Starbucks
1 review
"This was the worst Starbucks ever. All they had was oranges."
Pros Und Cons!

• Very Efficient For
  – Fast Parallel Floating Point Processing
  – Single Instruction Multiple Data Operations
  – High Computation per Memory Access

• Not As Efficient For
  – Double Precision
  – Logical Operations on Integer Data
  – Branching-Intensive Operations
  – Random Access, Memory-Intensive Operations
The Rendering Pipeline

• Three conceptual stages
• A stage is pipeline & runs in parallel
• Performance set by slowest stage
• Modern graphics systems:
  – Software
  – hardware
Eine modern GPU

Input from CPU

Host interface

Vertex processing

Triangle setup

Pixel processing

Memory Interface

64bits to memory

64bits to memory

64bits to memory

64bits to memory
Hardware Rendering Pipeline

- Raw Vertices & Primitives
- Transformed Vertices & Primitives
- Fragments
- Processed Fragments
- Pixels

1. **Vertex Processor (Programmable)**
2. **Rasterizer**
3. **Fragment Processor (Programmable)**
4. **Output Merging**

- **Display**

- **3D** triangles
- **2D array of color-values**

**Host Interface**
- vertex processing
- triangle setup
- pixel processing
- memory interface
Host Interface

- Communication bridge between CPU & GPU
- Input: Commands from CPU; geometry information from memory
- Output: Stream of vertices in object space with associated information - normals, texture coordinates, per vertex color etc.
Transform Spaces

Object Space
Vertex Processing

- Input: Vertices from host interface in object space
- Output: Vertices in screen space - No new vertices; no vertices are discarded

- Operations: Simple linear transformation, or a complex operation morphing effects
- What: Normals, texcoords etc are also transformed
Transform Spaces

Object Space → Screen Space
Triangle Setup

- **Input**: Screen space geometry
- **Output**: Raster/Pixels or Fragments
- **Operation**: Each fragment has attributes computed with perspective-correct interpolation of triangle vertices
Transform Spaces

Screen Space → Raster/Fragment
Triangle Setup - Optimizations

✓ O 1: Cull back-facing or outside viewing frustum

✓ O 2: Hidden Surface Removal

✓ O 3: Fragment is generated if and only if its center is inside the triangle
Fragment Processing

✓ Input: Fragments & attributes - position, normal, texcoord etc.

✓ Output: Final color for pixel

✓ Operations: Texture mapping & math operations

✓ Caveat: Bottleneck(s)
Memory Interface

✓ Input: Fragment

✓ Output: framebuffer operations
Programmability

✓ Vertex, fragment processing, triangle set-up programmable

✓ Programs executed for every vertex and every fragment

✓ Fully customizable geometry and shading effects
Advanced Musings
GPU Architecture Progression

1999
- Multi-texture, 32b rendering

2001
- Programmable vertex, 3D textures, shadow maps, multisampling

2003
- Fragment programs, color and depth compression

2005
- Transparency anti-aliasing

2007
- Double Precision

1998
- 16-bit depth, color, and textures

2000
- T&L, cube maps, texture compression, anisotropic filtering

2002
- Early 2-cull, dual-monitor

2004
- Flow control, FP textures, VTF

2006
- Unified shader, geometry shader, CUDA/C

Ballistics

Far Cry

© 2008 NVIDIA Corporation.

(courtesy: nvidia)
Modern GPU’s: Unified Architecture

Discrete Design

- Shader A
- Shader B
- Shader C
- Shader D

Unified Design

- Shader Core

Buffers

Vertex shaders, pixel shaders, etc. become *threads* running different programs on flexible cores.

(courtesy: nvidia)
Why unify?

Vertex Shader

Pixel Shader

Idle hardware

Heavy Geometry
Workload Perf = 4

Vertex Shader

Pixel Shader

Idle hardware

Heavy Pixel
Workload Perf = 8

(courtesy: nvidia)
Why unify?

- Unified Shader
  - Vertex Workload
  - Heavy Geometry
  - Workload Perf = 11

- Unified Shader
  - Pixel Workload
  - Heavy Pixel
  - Workload Perf = 11

(courtesy: nvidia)
The Holy Grail - Realism

Fracture
Soft Shadows
Indirect Lighting
Subsurface Scatter
Turbulence
Participating Media
Simulations
Fluids
Rich Environments
Detailed Characters
Ambient Occlusion

(courtesy: nvidia)
Software

Screen of 10x10 pixels.

most displays
are much bigger.

draw a cube
of 8 vertices

Surfaces cover
46 pixel-sized
"fragments"

8 vertex shaders run
def.

8/700 GPU slots used

Step 2:

vertices are now
discovered

"rasterized"

into 2D

(Step 2)

Step 3:

46 fragment
shaders are run
in parallel

46/700 GPU slots used.
*GL*
OpenGL
GLSL
WebGL
WebGL
Execution in Browser

1. **Browser**
   - Receives URL
   - Downloads HTML, JS files

2. **JS Engine**
   - Interprets HTML and JS files
   - Outputs Web Page

3. **CPU/GPU**
   - Processes Web Page
   - Sends Canvas data to Framebuffer

4. **Web Server**
   - Sends Web Page to Browser

5. **Framebuffer**
   - Draws Canvas

6. **Canvas**
   - Displays final output
A OpenGL Simple Program

Generate a square on a solid background
```c
#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
Event Loop

- Remember that 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
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

`gl.uniform3f(x,y,z)`
- **function name**: `gl.uniform3f`
- **dimension**: 3
- **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)
<!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>
</head>
</html>
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
<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
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 ),
        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
```javascript
render();
};

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

```javascript
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
```
BigBang of *GL*
OpenGL Architecture
Graphical APIs

• 1973 - two committees to propose 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)
  – 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 Khronos 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
Software Organization

![Diagram showing the software organization with nodes for OpenGL application program, GLEW, GL, GLUT, GLX, Xlib, Xt, and Graphics Driver. The diagram illustrates the flow and dependencies between these components.]
The End
Next – Sierpinski in GLSL