Programming with WebGL
Part 1: Background

CS 432 Interactive Computer Graphics
Prof. David E. Breen
Department of Computer Science
Objectives

- Development of the OpenGL API
- OpenGL Architecture
  - OpenGL as a state machine
  - WebGL as a data flow machine
- Functions
  - Types
  - Formats
- Simple program
Retained vs. Immediate Mode Graphics

• Immediate
  - Geometry is drawn when CPU sends it to GPU
  - All data needs to be resent even if little changes
  - Once drawn, geometry on GPU is discarded
  - Requires major bandwidth between CPU and GPU
  - Minimizes memory requirements on GPU

• Retained
  - Geometry is sent to GPU and stored
  - It is displayed when directed by CPU
  - CPU may send transformations to move geometry
  - Minimizes data transfers, but GPU now needs enough memory to store geometry
Early History of APIs

• IFIPS (1973) formed 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)
  - 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 graphics 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
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
- Focused on rendering
- Omitted windowing and input to avoid window system dependencies
- An immediate mode system, that later added retained mode functionality
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
OpenGL 3.1 (2009)

• 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
  - deprecate in CS - To mark (a component of a software standard) as obsolete to warn against its use in the future, so that it may be phased out.

• Backward compatibility not required
Other Versions

• OpenGL ES
  - Embedded systems
  - Version 1.0 simplified OpenGL 2.1
  - Version 2.0 simplified OpenGL 3.1
    • Shader based
  - Version 3.0 simplified OpenGL 4.3

• WebGL 1.0
  - Javascript implementation of ES 2.0
  - Supported on newer browsers

• OpenGL 4.1 ➔ 4.5
  - Added geometry & compute shaders and tessellator
What About Other Low-Level Graphics Libraries?

• Direct3D
  - Part of DirectX, Windows-only

• Mantle (discontinued)
  - Developed by AMD

• Metal
  - Developed by Apple

• Vulkan
  - “next-gen” OpenGL
  - Derived from Mantle
OpenGL Architecture

Application program → Graphics library (API) → Drivers → Display

Keyboard
Mouse
A Simple Program (?)

Generate a square on a solid background
It used to be easy

```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
• Made heavy use of state variable default values that no longer exist
  - Viewing
  - Colors
  - Window parameters
• Current version makes the defaults more explicit
• However, processing loop is the same
Execution in Browser

URL

Browser

Web Server

HTML
JS files

JS Engine

CPU/GPU

Framebuffer

Canvas

Angel and Shreiner: Interactive Computer Graphics 7E © Addison-Wesley 2015
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
  - `glUniform3f`
  - `glUniform2i`
  - `glUniform3dv`

• Underlying storage mode is the same
WebGL function format

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

- `x, y, z` are float variables
- `gl.uniform3f` belongs to WebGL canvas

`gl.uniform3fv(p)`

- `p` is an array
- `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
Programming with OpenGL
Part 2: 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
Coding in WebGL

• Example: Draw a square
  - Each application consists of three types of files

• HTML (index.html)
  - describes canvas, i.e. page layout
  - includes utility scripts
  - includes application scripts

• JavaScript
  - contains the actual graphics code

• GLSL
  - contains shader code
Coding in WebGL

- Can run WebGL on any recent browser
  - Chrome
  - Firefox
  - Safari
  - Edge
- Code written in JavaScript
- JS runs within browser
  - Use local resources
Square Program
WebGL

- Five steps
  - Describe page (HTML file)
    - request WebGL Canvas
    - read in necessary files
  - Define shaders (GLSL file)
  - Compute or specify data (JS file)
  - Send data to GPU (JS file)
  - Render data (JS file)
<!DOCTYPE HTML>
<html>
<head>
<script src="https://greggman.github.io/webgl-lint/webgl-lint.js" crossorigin></script>
<script type="text/javascript" src="./Common/initShaders2.js"></script>
<script type="text/javascript" src="./Common/MVnew.js"></script>
<script type="text/javascript" src="./square.js"></script>
<script type="text/javascript" src="./app.js"></script>
</head>
<body>
<canvas id="gl-canvas" width="512" height="512">
Not supported
</canvas>
</body>
</html>
Utility Files

- **webgl-lint.js**: checks for common WebGL errors
  - See https://github.com/greggman/webgl-lint
- **../Common/initShaders2.js**: contains JS and WebGL code for reading, compiling and linking the shaders
- **../Common/MVnew.js**: matrix-vector package
Application Files

- **square.js**: contains JS & WebGL code for
  - defining square geometry
  - setting up shader programs
  - initializing buffers and pointers
  - drawing square

- **app.js**: contains JS & WebGL code for
  - initializing canvas and WebGL
  - render() function
  - instantiating square and rendering it
Shaders

• We access shaders through their filenames in the JS file
• These are trivial pass-through (do nothing) shaders which set the
  - one required built-in variable (gl_Position) in the vertex shader
  - assign an output color for the fragment
• Note both shaders are full programs
• Note vector types vec2 and vec4
• Must set precision in fragment shader
Notes

- **onload**: determines where to start execution when all code is loaded
- `canvas` gets WebGL context from HTML file
- `vertices` use `vec2` type in `MVnew.js`
- JS array is not the same as a C or Java array
  - object with methods
  - `vertices.length` // 4
- Values in clip coordinates
• `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
function render() {
    gl.clear( gl.COLOR_BUFFER_BIT );
    sq.draw();
}

draw() {
    ... 
    gl.drawArrays( gl.TRIANGLE_FAN, 0, 4 );
    ... 
}
Triangles, Fans or Strips

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

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

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

Angel and Shreiner: Interactive Computer Graphics 7E © Addison-Wesley 2015
JavaScript Notes

• JavaScript (JS) is the language of the Web
  - All browsers will execute JS code
  - JavaScript is an interpreted object-oriented language

• References
  - Flanagan, JavaScript: The Definitive Guide, O’Reilly
  - Crockford, JavaScript, The Good Parts, O’Reilly
  - Many Web tutorials
JS Notes

• Is JS slow?
  - JS engines in browsers are getting much faster
  - Not a key issues for graphics since once we get the data to the GPU it doesn’t matter how we got the data there

• JS is a (too) big language
  - We don’t need to use it all
  - Choose parts we want to use
  - Don’t try to make your code look like C or Java
• Very few native types:
  - numbers
  - strings
  - booleans

• Only one numerical type: 64 bit float
  - var x = 1;
  - var x = 1.0; // same
  - potential issue in loops
  - two operators for equality == and ===

• Dynamic typing
Scoping

• Different from other languages
• Function scope
  • variables are *hoisted* within a function
    - can use a variable before it is declared
  • Note functions are first class objects in JS
JS Arrays

• JS arrays are objects
  - inherit methods
  - var a = [1, 2, 3];
    is not the same as in C++ or Java
  - a.length  // 3
  - a.push(4); // length now 4
  - a.pop();  // 4
- avoids use of many loops and indexing
- Problem for WebGL which expects C-style arrays
Typed Arrays

JS has typed arrays that are like C arrays

```javascript
var a = new Float32Array(3)
var b = new Uint8Array(3)
```

Generally, we prefer to work with standard JS arrays and convert to typed arrays only when we need to send data to the GPU with the flatten function in MVnew.js
A Minimalist Approach

• We will use only core JS and HTML
  - no extras or variants

• No additional packages
  - CSS
  - JQuery

• Focus on graphics
  - examples may lack beauty
Buffer Object

• Buffers objects allow us to transfer large amounts of data to the GPU
• Need to create, bind (make current) and identify/specify data

```javascript
var buffer_id;
buffer_id = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, buffer_id);
gl.bufferData(gl.ARRAY_BUFFER, data,
              gl.STATIC_DRAW);
```

• Data in current buffer is sent to GPU
Why use Buffer Objects?

Only Advantages

• The memory manager in the buffer object will put the data into the best memory locations based on user's hints

• Memory manager can optimize the buffers by balancing between 3 kinds of memory:
  - system, GPU and video memory
gl.createBuffer()

- creates a buffer object and returns the buffer object

WebGLBuffer gl.createBuffer()

- Returns a WebGLBuffer for storing data such as vertices or colors.
gl.bindBuffer()

- Once the buffer object has been created, we need to bind it to a target.
- Also makes the buffer “current”

void gl.bindBuffer(GLenum target, WebGLBuffer buffer)

- Target can be
  • gl.ARRAY_BUFFER: Any vertex attribute, such as vertex coordinates, texture coordinates, normals and color component arrays
  • gl.ELEMENT_ARRAY_BUFFER: Index array which is used for glDraw[Range]Elements()

- Once first called, the buffer is initialized with a zero-sized memory buffer and sets the initial states
gl.bufferData()

- You can initialize and copy the data into the buffer object with gl.bufferData().

```c
void gl.bufferData(GLenum target, GLsizeiptr size,
                   GLenum usage)
void gl.bufferData(GLenum target, ArrayBuffer data,
                   GLenum usage)
```

- target is either GL_ARRAY_BUFFER or GL_ELEMENT_ARRAY_BUFFER.
- size is the number of bytes of data to transfer.
- Data is the array holding the data to be copied.
- "usage" flag is a performance hint to provide how the buffer object is going to be used: static, dynamic or stream, and read, copy or draw.
Usage Flags

- `gl.STATIC_DRAW`
  - Contents of the buffer are likely to be used often and not change often.

- `gl.DYNAMIC_DRAW`
  - Contents of the buffer are likely to be used often and change often.

- `gl.STREAM_DRAW`
  - Contents of the buffer are likely to not be used often.

- All contents are written to the buffer, but not read.
void gl.bufferSubData(GLenum target, GLintptr offset, ArrayBuffer data)

- Like gl.bufferData(),
  - used to copy data into BO
- It only replaces a range of data into the existing buffer, starting from the given offset.
- The total size of the buffer must be set by gl.bufferData() before using gl.bufferSubData().
gl.deleteBuffer()

void gl.deleteBuffers(WebGLBuffer buffer)

- You can delete a BO with gl.deleteBuffer(), if it is no longer needed. After a buffer object is deleted, its contents will be lost.
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 *vertices* are determined by the application and are called *object, world, model* or *problem coordinates*
• Viewing specifications usually are also in object coordinates
• `GL_Positions` are passed to clipping volume
  - Most important is *clip 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
Coordinate Systems and Shaders

• Vertex shader must output vertices in clip coordinates
• Input to fragment shader from rasterizer is in window coordinates (pixels)
• Application can provide vertex data in any coordinate system, but vertex shader must eventually produce `gl_Position` in clip coordinates
• Simple example uses clip coordinates
WebGL Camera

• WebGL places a camera at the origin in camera space pointing in the negative $z$ direction
• The view/clipping volume is a box centered at the origin with sides of length 2
• $(-1, -1, -1) \Rightarrow (1, 1, 1)$
In the default orthographic (parallel) view, all points in the view volume are projected along the \( z \) axis onto the plane \( z = 0 \).
WebGL View Volume

• Only geometry (gl_Positions) inside of view volume will be rendered!
• Doesn’t matter if they are in front of or behind camera (origin)
• The viewing/clipping volume is a box centered at the origin with sides of length 2
• \((-1,-1,-1) \Rightarrow (1,1,1)\)
Viewing Rectangle Maps to Viewport

Objects in the Viewing Rectangle are mapped into the Viewport
Note the window’s coordinate frame!
Viewports

- Do not have to use the entire canvas for the image: `gl.viewport(x, y, w, h)`
- Values in pixels (window coordinates)
- `w` and `h` should be and `x` and `y` are recommended to be non-negative
- Specified in `square.js`
Transformations and Viewing

- In WebGL, projection is carried out by a projection matrix (transformation)
- Transformation functions are also used for changes in coordinate systems
- Pre 3.0 OpenGL had a set of transformation functions which have been deprecated
- Three choices
  - Application code
  - GLSL functions
  - MVnew.js
First Programming Assignment

• Get example code from HW1 web page
• Get test code running
• Make minor modifications to it
• Draw red pentagon, instead of a white square
First Programming Assignment

- Change viewport (app.js)
- Add vertex/vertices to define another triangle (square.js)
- Modify gl.drawArrays() (square.js)
- Change output color (vshader.glsl)