WebGL: Hands On

DevCon5 NYC 2011

Kenneth Russell
Software Engineer, Google, Inc.
Chair, WebGL Working Group

Today's Agenda

Introduce WebGL and its programming model.

Show code for a complete example.

Demonstrate techniques for achieving best performance.

Introduction

WebGL brings OpenGL ES 2.0 to the web.

It is a proven rendering model, but very different than other web APIs.

The triangle is the basic drawing primitive.

Data for many triangles is prepared once, drawn many times.
Vertex positions, colors, textures, ...

Shaders – small programs that execute on the graphics card – determine position of each triangle and color of each pixel.

Programming Model

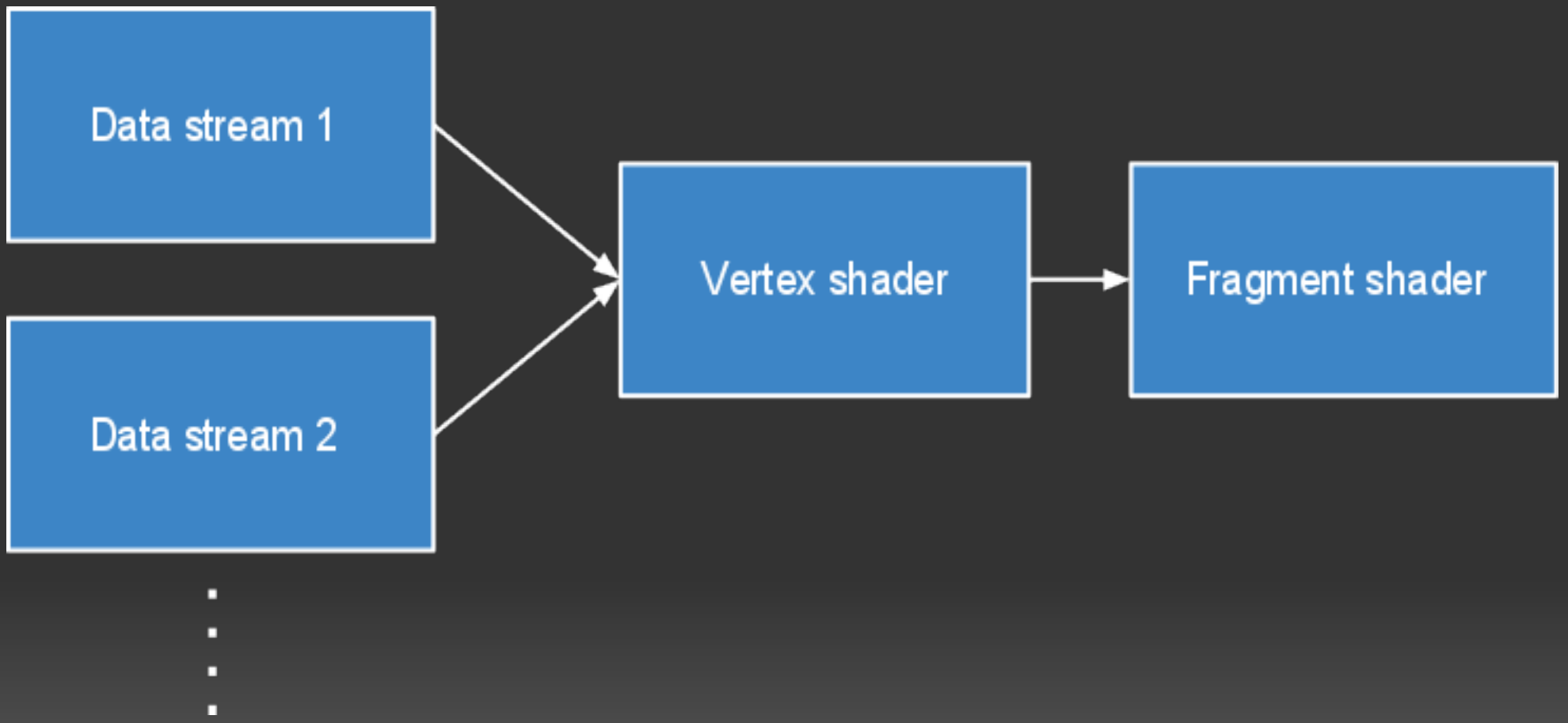
The GPU is a stream processor.

Each point in 3D space has one or more streams of data associated with it.
Position, surface normal, color, texture coordinate, ...

In OpenGL (and WebGL) these streams are called vertex attributes.

These streams of data flow through the vertex and fragment shaders.

Programming Model



Vertex and Fragment Shaders

Small, stateless programs which run on the GPU with a high degree of parallelism.

The vertex shader is applied to each vertex of each triangle.

It may output values to the fragment shader.

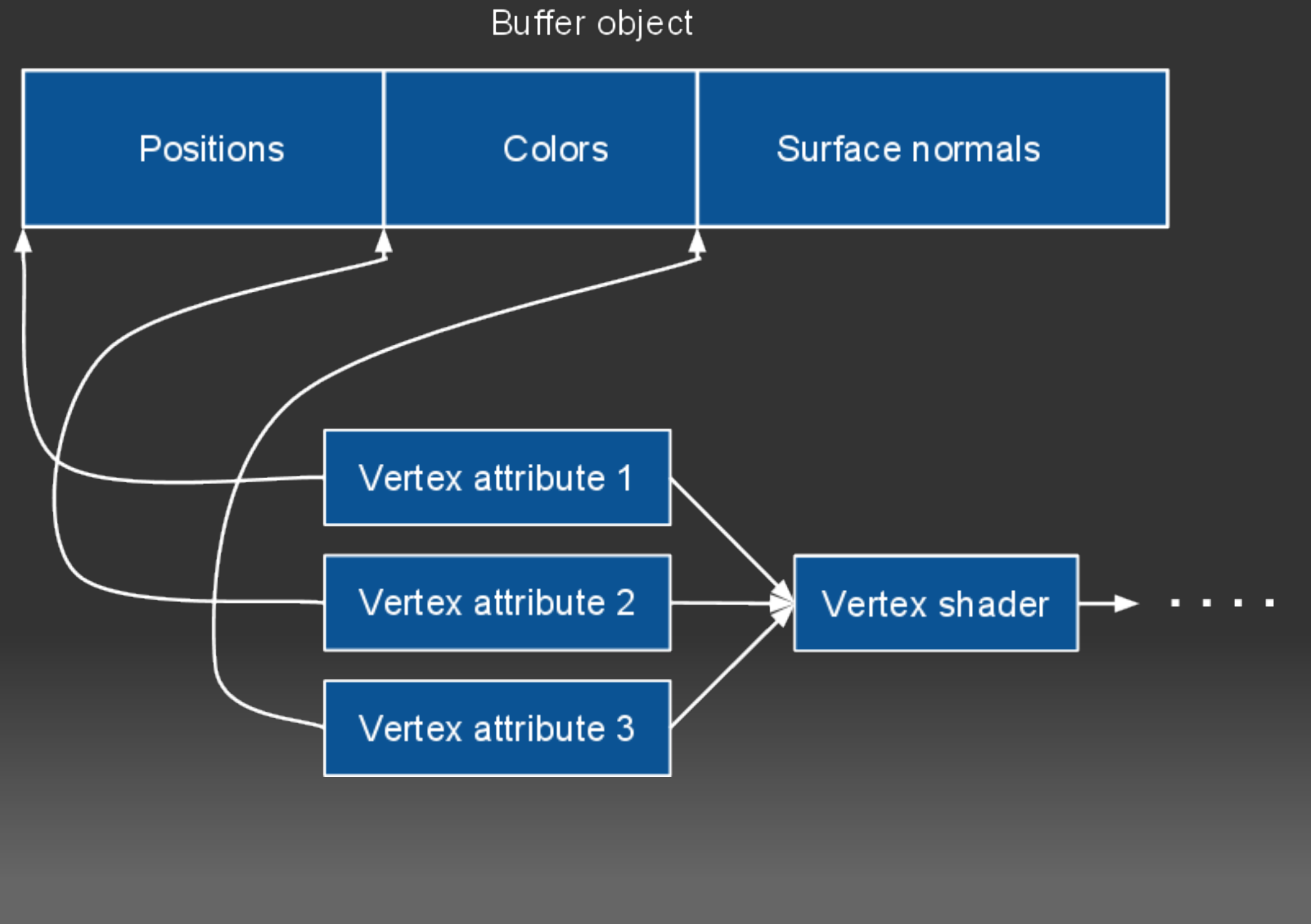
The GPU figures out which pixels on the screen are covered by the triangle.

The fragment shader is run at each pixel, automatically blending the outputs of the vertex shader.

Getting Data On To the GPU

Vertex data is uploaded in to one or more buffer objects.

The vertex attributes in the vertex shader are bound to the data in these buffer objects.


Buffer object
Positions
Colors
Surface normals
Vertex attribute 1
Vertex attribute 2
Vertex attribute 3
Vertex shader

A Concrete Example

Adapted from Giles Thomas' Learning WebGL Lesson 2

Code is checked in to http://webglsamples.googlecode.com/ under hello-webgl/

Vertex Shader

```
attribute vec3 positionAttr;
attribute vec4 colorAttr;

varying vec4 vColor;

void main(void) {
    gl_Position = vec4(positionAttr, 1.0);
    vColor = colorAttr;
}
```

Fragment Shader

```
precision mediump float;

varying vec4 vColor;
void main(void) {
    gl_FragColor = vColor;
}
```

Initializing WebGL

```
var gl;
function initGL(canvas) {
  try {
    gl = canvas.getContext("experimental-webgl");
    gl.viewportWidth = canvas.width;
    gl.viewportHeight = canvas.height;
  } catch (e) {
  }
  if (!gl) {
    alert("Could not initialise WebGL, sorry :-(");
  }
}
```

Not the best detection and error reporting code.
Link to http://get.webgl.org/ if initialization fails.

Shader Text

The shader text for this sample is embedded in the web page using script elements.

```
<script id="shader-vs" type="x-shader/x-vertex">
  attribute vec3 positionAttr;
  attribute vec4 colorAttr;
  ...
</script>
<script id="shader-fs" type="x-shader/x-fragment">
  precision mediump float;
  varying vec4 vColor;
  ...
</script>
```

Loading a Shader

Create the shader object – vertex or fragment.

Specify its source code.

Compile it.

Check whether compilation succeeded.

Complete code follows. Some error checking elided.

```
function getShader(gl, id) {
  var script = document.getElementById(id);
  var shader;
  if (script.type == "x-shader/x-vertex") {
    shader = gl.createShader(gl.VERTEX_SHADER);
  } else if (script.type == "x-shader/x-fragment") {
    shader = gl.createShader(gl.FRAGMENT_SHADER);
  }
  gl.shaderSource(shader, script.text);
  gl.compileShader(shader);
  if (!gl.getShaderParameter(
      shader, gl.COMPILE_STATUS)) {
    alert(gl.getShaderInfoLog(shader));
    return null;
  }

  return shader;
}
```

Loading the Shader Program

A program object combines the vertex and fragment shaders.

Load each shader separately.

Attach each to the program.

Link the program.

Check whether linking succeeded.

Prepare vertex attributes for later assignment.

Complete code follows.

```
var program;

function initShaders() {
  var fragmentShader = getShader(gl, "shader-fs");
  var vertexShader = getShader(gl, "shader-vs");
  program = gl.createProgram();
  gl.attachShader(program, vertexShader);
  gl.attachShader(program, fragmentShader);
  gl.linkProgram(program);
  if (!gl.getProgramParameter(program, gl.LINK_STATUS))
    alert("Could not initialise shaders");
  gl.useProgram(program);
  program.positionAttr =
    gl.getAttribLocation(program, "positionAttr");
  gl.enableVertexAttribArray(program.positionAttr);
  program.colorAttr =
    gl.getAttribLocation(program, "colorAttr");
  gl.enableVertexAttribArray(program.colorAttr);
}
```

Setting Up Geometry

Allocate buffer object on the GPU.

Upload geometric data containing all vertex streams.

Many options: interleaved vs. non-interleaved data, using multiple buffer objects, etc.

Generally, want to use as few buffer objects as possible. Switching is expensive.

```
var buffer;
function initGeometry() {
  buffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, buffer);
  // Interleave vertex positions and colors
  var vertexData = [
    // Vertex 1 position
    0.0,  0.8,  0.0,
    // Vertex 1 Color
    1.0, 0.0, 0.0, 1.0,
    // Vertex 2 position
    -0.8, -0.8,  0.0,
    // Vertex 2 color
    0.0, 1.0, 0.0, 1.0,
    // Vertex 3 position
    0.8, -0.8,  0.0,
    // Vertex 3 color
    0.0, 0.0, 1.0, 1.0
  ];
  gl.bufferData(gl.ARRAY_BUFFER,
    new Float32Array(vertexData), gl.STATIC_DRAW);
}
```

Drawing the Scene

Clear the viewing area.

Set up vertex attribute streams.

Issue the draw call.

```
function drawScene() {
  gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);
  gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);

  gl.bindBuffer(gl.ARRAY_BUFFER, buffer);

  // There are 7 floating-point values per vertex
  var stride = 7 * Float32Array.BYTES_PER_ELEMENT;

  // Set up position stream
  gl.vertexAttribPointer(program.positionAttr,
    3, gl.FLOAT, false, stride, 0);
  // Set up color stream
  gl.vertexAttribPointer(program.colorAttr,
    4, gl.FLOAT, false, stride,
    3 * Float32Array.BYTES_PER_ELEMENT);

  gl.drawArrays(gl.TRIANGLES, 0, 3);
}
```

Libraries

Now that we've dragged you through a complete example...
Many libraries already exist to make it easier to use WebGL.
A list (not comprehensive):
http://www.khronos.org/webgl/wiki/User_Contributions#Frameworks

A few suggestions:
TDL (Aquarium, etc.)
Three.js (http://ro.me, mr. doob's demos)
CubicVR (Mozilla's WebGL demos like No Comply)
CopperLicht (same developer as Irrlicht)
PhiloGL (focus on data visualization)
SpiderGL (lots of interesting visual effects)
GLGE (used for early prototypes of Google Body)
SceneJS (Interesting declarative syntax)

Achieving High Performance

OpenGL "big rules" are to minimize:

Draw calls
Buffer, texture, and program binds
Uniform variable changes
State switches (enabling/disabling)

WebGL "big rule":

Offload as much JavaScript to the GPU as possible

Picking in Google Body

(Thanks to Body team for this information)

Highly detailed 3D model – over a million triangles

Selection is very fast

Picking in Google Body

Could consider doing ray-casting in JavaScript

- Attempt to do quick discards if ray doesn't intersect bbox

Still a lot of math to do in JavaScript

Picking in Google Body

When model is loaded, assign different color to each organ

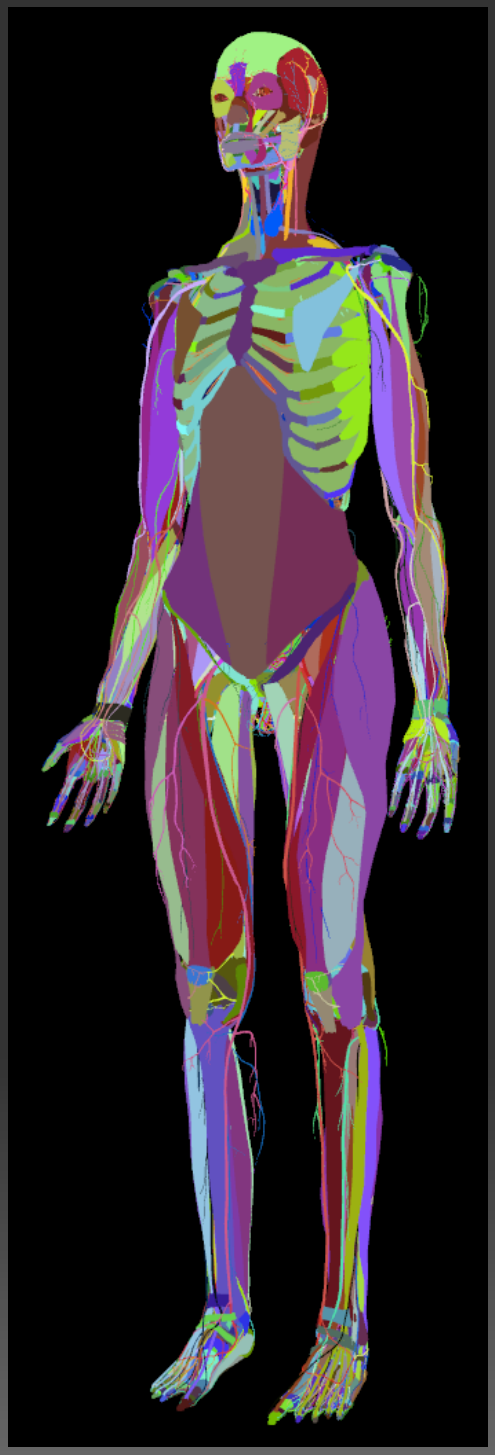
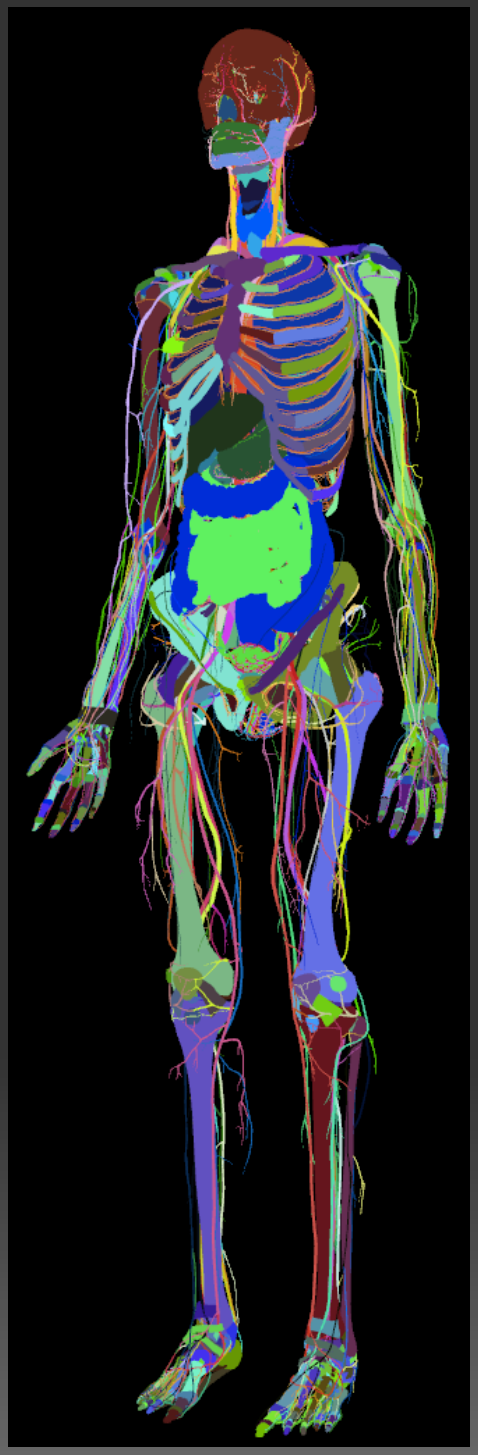
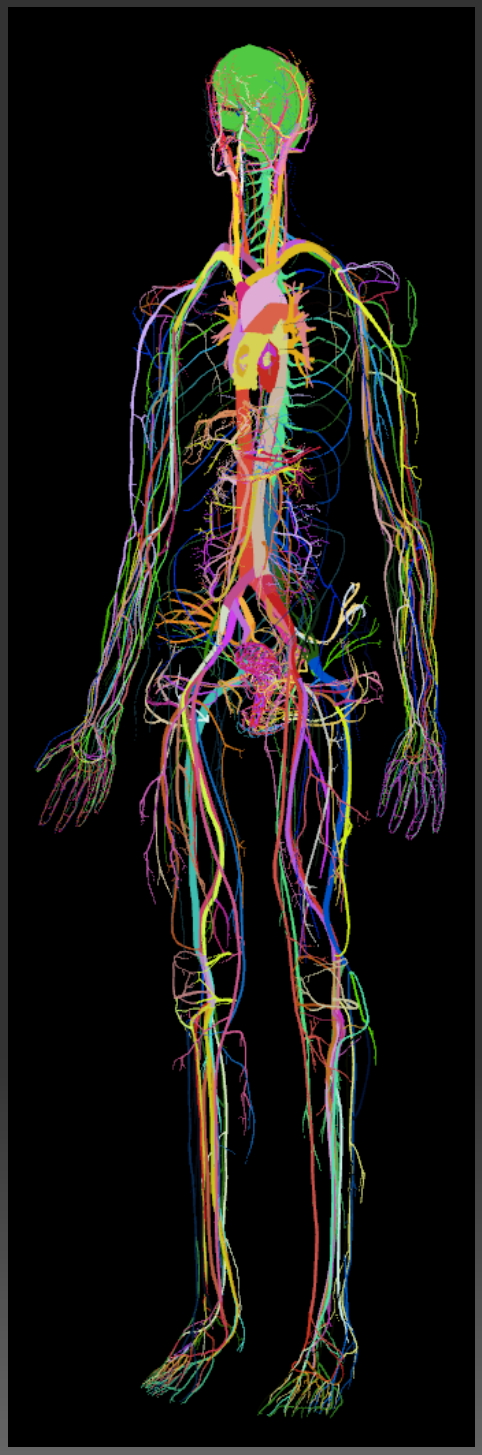
Upon mouse click:

Render body offscreen with different set of shaders

Use threshold to determine whether to draw translucent layers

Read back color of pixel under mouse pointer

Same technique works at different levels of granularity

Particle Systems

Particle demo from WebGL wiki (author: gman@)

http://www.khronos.org/webgl/wiki/Demo_Repository

Animates ~2000 particles at 60 FPS

Does all the animation math on the GPU

Particle Systems

Each particle's motion is defined by an equation

- Initial position, velocity, acceleration, spin, lifetime

Set up motion parameters when particle is created

Send down one parameter – time – each frame

Vertex shader evaluates equation of motion, moves particle

Absolute minimum amount of JavaScript work done per frame

Particle Systems

Worlds of WebGL demo shows another particle-based effect

http://www.nihilogic.dk/labs/worlds_of_webgl/

Uses point sprites rather than quads

Vertex shader computes perspective projection and sets gl_PointSize for each point

Particle Systems

Animation is done similarly to gman's particle demo

For each scene, random positions are chosen for particles at setup time

Time parameter interpolates between two position streams

Particles in air / particles on floor

Particles on floor / particle positions for next scene

JavaScript does almost no computation

Physical Simulation

WebGL supports floating point textures as an extension

Store the state of a physical simulation on the GPU

Any iterative computation where each step relies only on nearby neighbors is a good candidate for moving to GPU

Several examples (waves, interference patterns):

http://www.ibiblio.org/e-notes/webgl/gpu/contents.htm

More Resources

Gregg Tavares' Google I/O 2011 talk on WebGL Techniques and Performance

http://www.google.com/events/io/2011/sessions/webgl-techniques-and-performance.html

Giles Thomas' WebGL blog

http://learningwebgl.com/blog/

Mozilla's WebGL articles

http://hacks.mozilla.org/category/webgl/

WebGL Chrome Experiments

http://www.chromeexperiments.com/webgl/

More Resources

WebGL wiki
http://khronos.org/webgl/wiki

Public WebGL mailing list (spec discussion only)
https://www.khronos.org/webgl/public-mailing-list/

WebGL Developers' List
https://groups.google.com/group/webgl-dev-list

Q&A