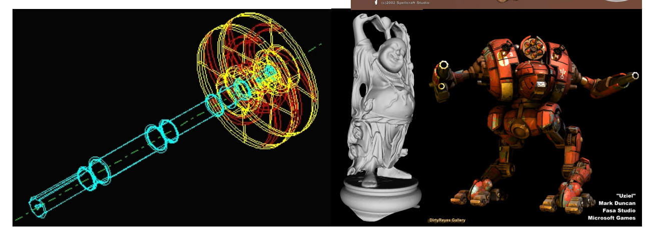
EECS 487: Interactive Computer Graphics

Lecture 20:
- Brief History of Graphics Hardware
- OpenGL ES and OpenGL 3.1+
- WebGL and HTML5 template(s)
- Javascript: a quick tutorial
- Sample code (same one as last lecture) at http://web.eecs.umich.edu/~sugih/courses/eecs487/common/notes/gl3+webgl.tgz

Graphics Hardware

Five generations
[Akeley & Hanrahan]
- wireframe
- shaded solids
- texture mapping
- programmability
- global illumination?



1st Gen: Wireframe (50's-70's)

Hardware: mainframe, vector graphics

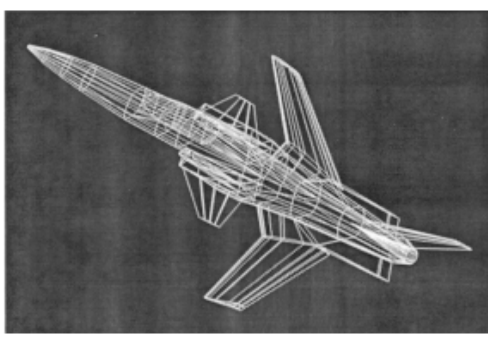
Tech: line drawing, hidden lines, parametric surfaces, solids, ray tracing

Vertex: transform, clip, project

Rasterization: color interpolation (points, lines)

Fragment: overwrite

API: GKS



Akeley&Hanrahan07

2nd Gen: Shaded Solids (80's)

Hardware: graphics workstation (SGI $50-100K)

Tech: framebuffers, z-buffer, flat and smooth shading

Vertex: lighting calculation

Rasterization: depth interpolation (triangle)

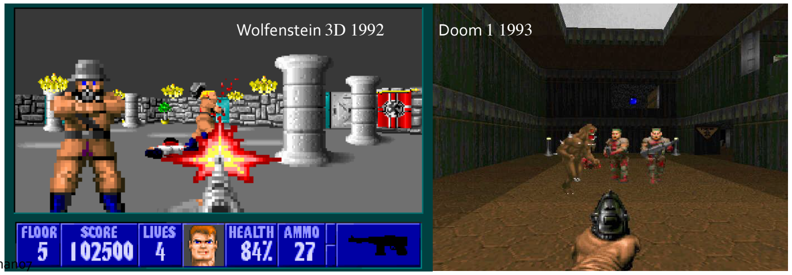
Fragment: depth buffer, color blending

APIs: PHIGS, Renderman

Akeley&Hanrahan07

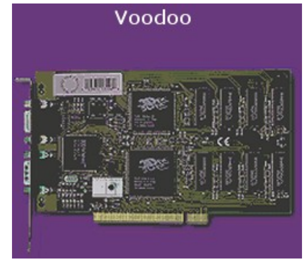
3rd Gen: Texture Mapping (90's)

Hardware: PC graphics board → GPUs ($10K-$200)
Tech: tex. mapping, transformation and lighting on GPU
Vertex: texture coordinate transform
Rasterization: texture coordinate interpolation
Fragment: texture evaluation, anti-aliasing
APIs: OpenGL (1.1 – 1.4), Direct3D (5.2 – 7)



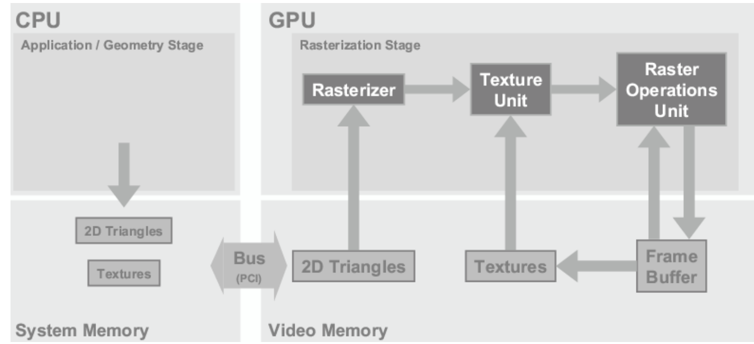
Akeley&Hanrahan07

1st Gen GPU (1995)



1st popular card: Voodoo by 3dfx

- only texture mapping and z-buffer were implemented on the card
- vertex transformation still done on CPU
- mostly "graphics accelerator"



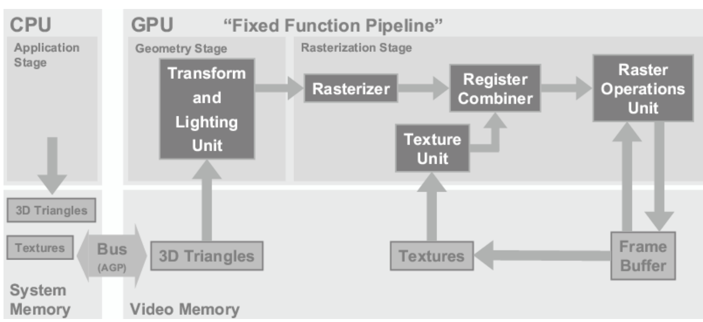
Zhu

2nd Gen GPU (1999)

1st consumer GPU: nvidia GeForce256

- transformation and lighting on card
- register combiner: fragment color from texture and interpolated color values
- complete hardware 3D pipeline on card
- Direct3D 7 usable

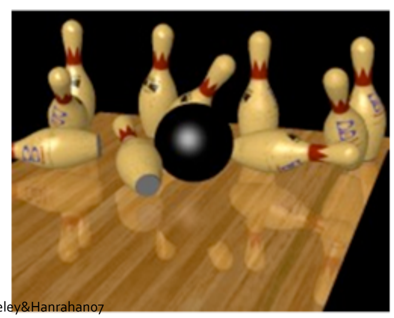
GeForce 256



Zhu

4th Gen: Shaders (2000's – now)

Hardware: GPU, mobile GPU
Tech: shaders, high-level shading languages, GPGPU
Vertex: programmable transform (2001)
Rasterization: user attributes interpolation
Fragment: programmable color computation (2002)



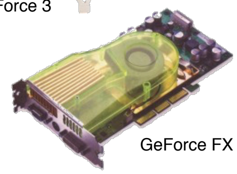
Akeley&Hanrahan07

4th Gen: Shaders (2001 – 2004)

3rd gen GPU ('01):

- nvidia GeForce3: 1st vertex shader
- ATI Radeon 8500, Xbox
- Direct3D 8.1
  - shader programming assembly like

GeForce 3



GeForce FX

4th gen GPU ('02-'04):

- GeForceFX, 1st fragment shader
- ATI Radeon 9700, floating point in fragment shader
- high-level shading languages: Direct3D 9.0/HLSL, OpenGL 2.0/GLSL 1.1
- OpenGL ES 1.1 (fixed function)

4th Gen: Shaders (2006 – 2008)

2006:

- Unified Shader Architecture (Shader Model 4.0): ATI Xenos in Xbox360
- tile-based deferred rendering (TBDR) mobile GPU: PowerVR MBX (to be used in original iPhone, Nokia N95)
- geometry shader: Direct3D 10, Windows Vista, GeForce 8800
- OpenGL 2.1/GLSL 1.2, OpenGL ES 2.0/GLSL ES 1.0 (programmable, PowerVR SGX535)

GeForce 8800

2007-08:

- OpenGL ES 1.1 on iPhone (iOS 1.0)
- OpenGL 3.0/GLSL 1.3 (deprecation mechanism, demise of Longs Peak)

4th Gen: Shaders (2009 – 2011)

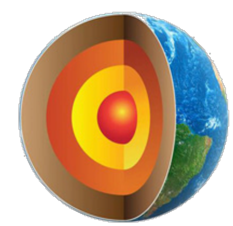
2009:

- Direct3D 11: tessellation and compute shaders, Windows 7, AMD HD 5870
- OpenGL 3.2/GLSL 1.5: geometry shader, GeForce 300
- mobile: ARM Mali 400 (TBDR, multi-core)
- Android 1.6 supports OpenGL ES 1.1
- iPad, iOS 3.0 (OpenGL ES 2.0, programmable pipeline)

2010-11:

- OpenGL 4.0/GLSL 4.0: tessellation shader
- WebGL 1.0
- Android 2.2 (OpenGL ES 2.0, programmable pipeline)

4th Gen: Shaders (2012 – 2013)

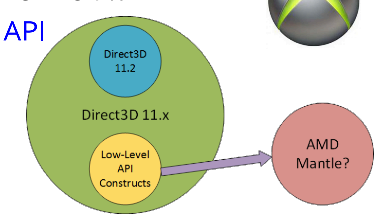


2012:

- OpenGL 4.0: compute shader, GeForce 6xx
- OpenGL ES 3.0: "modern" OpenGL

2013:

- Chrome 25, 1st browser to support WebGL, requires Android 4.0
- Android 4.3 and iOS 7 support OpenGL ES 3.0
- AMD Mantle: 1st low-level graphics API
- Direct3D 11.x: superset of 11.2, contains low-level API, runs on Xbox One



4th Gen: Shaders (2014)

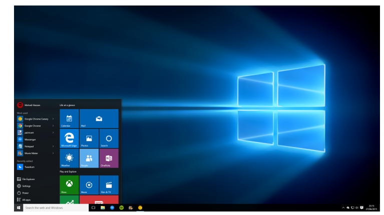
2014:

- OpenGL ES 3.1: compute shader, in Android 5.0 (Lollipop)
- OpenGL 4.5/GLSL 4.5: latest OpenGL standard
- March: Direct3D 12: low-level graphics API announced
- June: Metal: Apple's low-level graphics API released, released with iOS 8, requires A7 or later
- iOS 8 also supports WebGL
- August: glNext: Next Generation OpenGL Initiative
- Sept.: Direct3D 11.3 (latest high-level API, subset of D3D 12?)

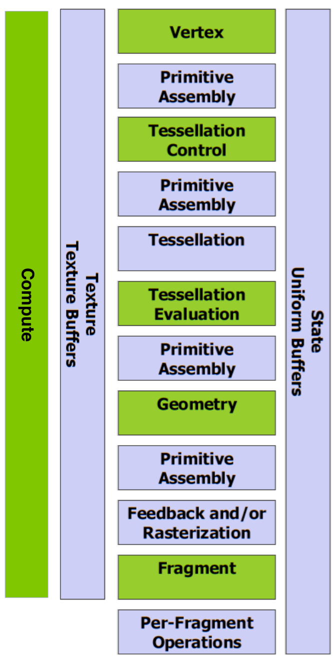
4th Gen: Shaders (2015)

2015:

- March: glNext → Vulkan (with AMD Mantle as core)
- June: Metal on OS X 10.11 (El Capitan)
- July: Windows 10 with Direct3D 11.3 and 12 released
- August: OpenGL ES 3.2: geometry and tessellation shaders, latest OpenGL ES standard



OpenGL 4.0 (3/11/10)
4.1 (7/26/10)
4.2 (8/8/11)
4.3 (8/6/12)
4.4 (7/22/13)
4.5 (8/11/14)



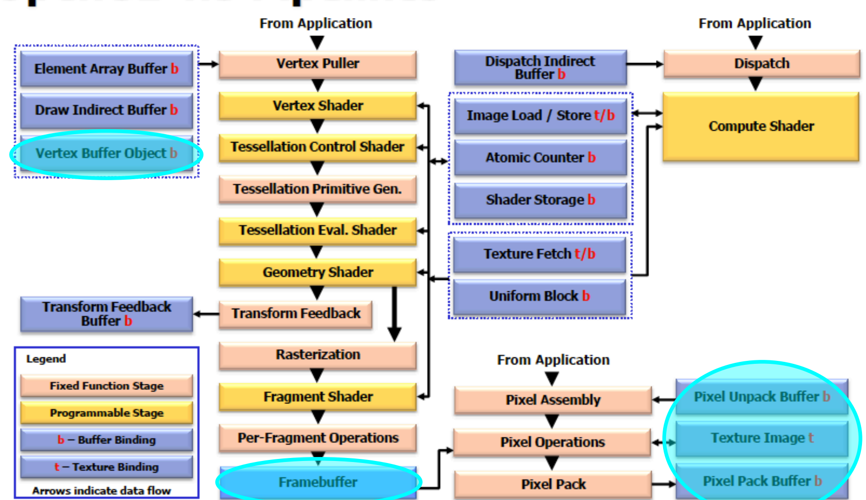
fixed function
programmable

Compute shader:

- image processing
- AI simulation
- global illumination
- physics processing
- etc.

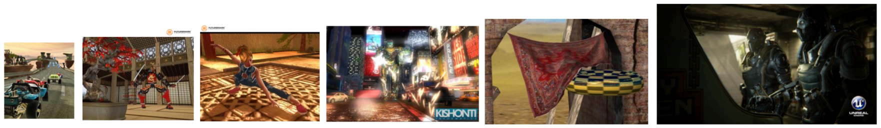
[Khronos]

OpenGL 4.3 Pipelines



[Khronos]

Subset of OpenGL for Embedded Systems (phones, games consoles, cars, etc.)
- convenience functions removed



2003 —Driver Update→ 2004 —Silicon Update→ 2007 —Silicon Update→ 2012 —Driver Update→ 2014 (GDC) —Silicon Update→ 2015

1.0 · 1.1 · 2.0 · 3.0 · 3.1 · 3.2

1.0/1.1:
- based on OpenGL 1.3/1.5
- fixed-function pipeline
- 1.0+extensions used in PS3 (PSGL)

2.0:
- based on OpenGL 2.0
- programmable pipeline, GLSL ES
- not 1.x compatible
- no fixed-function pipeline
- buffers, textures, shaders

3.0:
- based on OpenGL 3.3/4.x
- no geometry shader
- 2.0 compatible
- 32-bit float in GLSL ES

3.1:
- independent shaders
- compute shader

3.2:
- geometry shader
- tessellation shader

[Khronos]

OpenGL ES Ecosystem

What has been removed?
See the OpenGL ES specs for a complete list
Some highlights:

"Newbie friendliness" that makes OpenGL suitable for learning computer graphics is gone

Primitives more complicated than the triangle are also gone: `GL_QUADS`, `GL_QUAD_STRIP`, and `GL_POLYGON`, also `glMap` and `glEval`

OpenGL ES further doesn't support `double`

Vertex transformation and lighting are gone:
- transformation: `glMatrixMode`, `glLoadIdentity`, `glRotate`, `glTranslate`, `glScale`, `glOrtho`, `glFrustum` etc.
- lighting: `glLight`, etc.

GLU and GLUT, including `gluLookAt`, `gluPerspective`, `gluSphere`, and `glut*Teapot`, `glut*Torus`, etc.

You need to provide these in your app or shaders

The following vertex passing modes are also gone:

- immediate mode rendering (`glBegin`, `glEnd`), along with streaming vertex attributes `glVertex`, `glColor`, `glNormal`, `glTexCoords`, etc.
- client-side buffering: vertex array, color array, etc.
- client-side vertex attribute locations: `GL_VERTEX_ARRAY`, `GL_COLOR_ARRAY`, `GL_NORMAL_ARRAY`, etc.
- compiled retained mode rendering with display list: `glGenList`, `glNewList`, `glEndList`, `glCallList`, etc.

Use VBO and VAO with custom vertex attributes

What are Left?

Buffers, textures, and shaders

BUT all is not lost:

- most 3.0+ drivers support the (2.1) Compatibility Profile
- most 2.1 drivers have been updated to provide most of the new functions as extensions

For a complete list of deprecated API calls and supported extensions, see the OpenGL APIs Table
(http://web.eecs.umich.edu/~sugih/courses/eecs487/common/notes/APITables.xml)

Why OpenGL 3.1+?

Three reasons to migrate to OpenGL 3.1+:

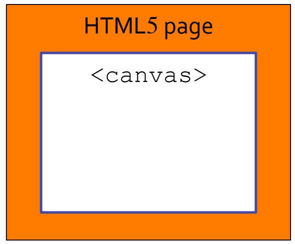
1. easier porting to OpenGL ES
2. write very high-performance animated graphics
3. vertex attributes can be `int` (for array indexing, for example)

GPU-accelerated 3D graphics in web browsers

- OpenGL for JavaScript
- based on OpenGL ES 2.0 and GLSL ES 1.0
  - APIs mostly the same as OpenGL ES 2.0, minus the `gl` prefix, and that all object generation APIs (`glGen*()`) have been renamed `create*()`, which generate only one object at a time, not an array
- no fixed-function APIs, 3D scene must be programmed using "buffers, textures, and shaders"

GPU-accelerated 3D graphics in web browsers

- requires HTML5 Canvas element as render target
  - a canvas element is a rectangular drawing area within an HTML5 page that can be dynamically updated using JavaScript
  - WebGL is a rendering context for HTML5 canvas element (the other is "2d")



OS/driver must support OpenGL ES 2.0

- on Windows, Chrome and Firefox use ANGLE (Almost Native Graphics Layer Engine) to translate OpenGL ES 2.0 (3.0 to come) to Direct3D 9.0c/11
- IE 11 (Windows 8.1), concern about WebGL security
- Chrome on Android 4.0+
- iOS 8.0+, iPhone 4S or later

WebGL demoes (not all may run on your platform):

- http://www.bongiovi.tw/experiments/webgl/blossom/
- http://madebyevan.com/webgl-water/ (requires extension)
- http://www.playmapscube.com

Advantages

Using HTML5 to create "Web Apps" has many advantages:

- portable to any browser-enabled system (`html5test.com`)
- minimal efforts required to port app to web page
- web app is searchable and discoverable through the web
- not a closed app store – no app store "tax"

Template Alternative 1

JavaScript as an HTML object event handler

```
<html>
<head>
<title>Alternative 1</title>
<script id="minvs" type="x-shader/x-vertex">
 attribute vec4 vPos;                                  // plain GLSL
 uniform mat4 PMVmat;
 void main() { gl_Position = PMVmat*vPos; }
</script>
<script id="minfs" type="x-shader/x-fragment">
  precision mediump float;                             // plain GLSL
  void main(void) { gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0); }
</script>
<script type="text/javascript" src="sample.js"></script>
</head>

<body onload="sample_main('sample_canvas', 'minvs', 'minfs')">
       <!-- limited to 1 per html page -->
<canvas id="sample_canvas" width="300" height="300"></canvas>
</body>
</html>
```

Template Alternative 2

```
<html>
<head>
<title>Alternative 2</title>
</head>
<body>
<canvas id="sample_canvas" width="300" height="300"></canvas>
<script id="minvs" type="x-shader/x-vertex">
 attribute vec4 vPos;                          // plain GLSL
 uniform mat4 PMVmat;
 void main() { gl_Position = PMVmat*vPos; }
</script>
<script id="minfs" type="x-shader/x-fragment">
  precision mediump float;                     // plain GLSL
  void main(void) { gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0); }
</script>
<script type="text/javascript">
  /* put the content of sample.js inline here */
  sample_main("sample_canvas", "minvs", "minfs");  // call
main(), all <script> must be after <canvas>
</script>
</body>
</html>
```

`<script>` in `<body>`, after `<canvas>` (was that `<script>` must be in `<head>`)

Template Alternative 3

```
<html>
<head>
<title>Alternative 3</title>
<script id="minvs" type="x-shader/x-vertex">
 // plain GLSL, cannot be in include file
 attribute vec4 vPos; uniform mat4 PMVmat;
 void main() { gl_Position = PMVmat*vPos; }
</script>
<script id="minfs" type="x-shader/x-fragment">
  precision mediump float;      // plain GLSL, must be inline
  void main(void) { gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0); }
</script>
<script type="text/javascript" src="sample.js"></script>
</head>

<body>
<canvas id="sample_canvas" width="300" height="300"></canvas>
<script>sample_main("sample_canvas", "minvs", "minfs");</script>
</body>
</html>
```

Template Alternative 3

Can have multiple WebGL canvases on the same page, but beware of name conflict in global variables, including functions

```
<body>

<!-- first canvas -->
<canvas id="sample" width="300" height="300"></canvas>
<script>main();</script>

<!-- second canvas -->
<canvas id="sample2" width="100" height="100"></canvas>
<script>main2();</script>

</body>
```

See `sample*.html`, `sample.js` (and `glum.js`) and `vao-cva.{html,js}` in http://web.eecs.umich.edu/~sugih/courses/eecs487/common/notes/gl3+webgl.tgz

JavaScript Variables

Dynamic typing:
variables are of the same type as the values assigned

```
a = 32;  // 'a' is an int
a = "thirty two";
    /* type changed, not an error */
a = 32 + "is thirty two";
    // 'a' is a string "32 is thirty two"
```

See how comments are marked?

Variables

Declaration and scoping:

- either just assign a value to a variable, scope is global:

```
a = 32;  // not recommended
```

- or declare with keyword "`var`", scope can be local or global

```
var a = 1;
function f() {
  var a = "a string";
  alert(a); /* prints "a string" */
}
alert(a); // prints 1
```

- no block scoping (declarations inside a loop are visible throughout function)
- hoisting/lifting: declarations automatically moved to top of function

See how functions are declared?

Variables

Arrays, objects, and functions are passed by reference

Everything else is passed by value

```
stack.push(arr);
```

pushes a reference to array onto the stack:
if `arr` element is later modified,
`arr` at the top of the stack is also modified; use

```
stack.push(new ArrayBuffer(arr));
```

to push a copy of `arr` onto stack

See: https://developer.mozilla.org/en-US/docs/JavaScript/Guide/Values,_variables,_and_literals

Array

Arrays are dynamically sized by usage:

```
var arr = []; // empty array
arr[99] = 1;  // array now has 100 elements
```

Arrays also come with `push()` and `pop()`, handy for constructing array of arrays and array of objects (remember to make a copy of elements pushed onto the array)

See https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Array

Object

Objects are associative arrays;
members and methods are created by assignment:

```
var obj = { id1: 1, f: null };
obj["id2"] = ' is one';  // member
obj.f = function () {    // method
  alert(obj["id1"]+obj.id2);
};
```

See https://developer.mozilla.org/en-US/docs/JavaScript/Guide/Working_with_Objects

Typed Array

WebGL needs to specify precise byte offset for buffer and texture operations

JavaScript adds Typed Array, a fixed-length buffer type, and provides View Types for type casting

```
var b = new ArrayBuffer(8);
var f = new Float32Array(b);
```

same as:

```
var f = new Float32Array(2);
```

var	index							
	bytes (not indexable)							
b=	0	1	2	3	4	5	6	7
	indices							
f=	0				1			

JavaScript has no `sizeof()` function,
but Typed Array has `BYTES_PER_ELEMENT` constant

See: http://www.khronos.org/registry/typedarray/specs/latest/

Typed Array

To implement:

```
struct record {
  ulong id; char uname[16]; float height;
};
```

do:

```
var record = new ArrayBuffer(24);
var id = new Uint32Array(record, 0, 1); // byte offset, count
var uname = new Uint8Array(record, 4, 16);
var height = new Float32Array(record, 20, 1);
```

height is accessed as `height[0];`

See https://developer.mozilla.org/en-US/docs/JavaScript_typed_arrays

Math etc.

For a list of math functions, see:
https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Math
note that trigonometric functions take angles in radian

For other JavaScript related topics see:
https://developer.mozilla.org/en-US/docs/JavaScript/Guide
http://www.quirksmode.org/js/contents.html
http://unixpapa.com/js/
https://developer.mozilla.org/en-US/docs/JavaScript/Reference

HTML JavaScript

JavaScript is most often used as an event handler

Event handlers can be assigned to HTML element attributes inside a JavaScript function, e.g.,:

```
function ReshapeFunc(reshapeFunc) {
  document.body.onresize=reshapeFunc;
}
function reshape() {
  gl.Viewport(0,0,width,height);
}
main() {
  ReshapeFunc(reshape);
}
```

Or event handlers can be specified as HTML element attribute at element definition, e.g.,:

```
<body onload="main()">
```

For a discussion on events and list of events, see:

http://www.quirksmode.org/js/events_properties.html

https://developer.mozilla.org/en-US/docs/DOM/event
https://developer.mozilla.org/en-US/docs/DOM/element
https://developer.mozilla.org/en-US/docs/DOM/window
https://developer.mozilla.org/en-US/docs/DOM/document

For keyboard and mouse event handling at a glance:
http://www.javascriptkit.com/jsref/eventkeyboardmouse.shtml

Further discussion on keyboard event and key codes:
http://www.quirksmode.org/js/keys.html
http://unixpapa.com/js/key.html

Further discussion on mouse event:
http://www.quirksmode.org/js/events_mouse.html
http://unixpapa.com/js/mouse.html
https://developer.mozilla.org/en-US/docs/XUL_Tutorial/More_Event_Handlers

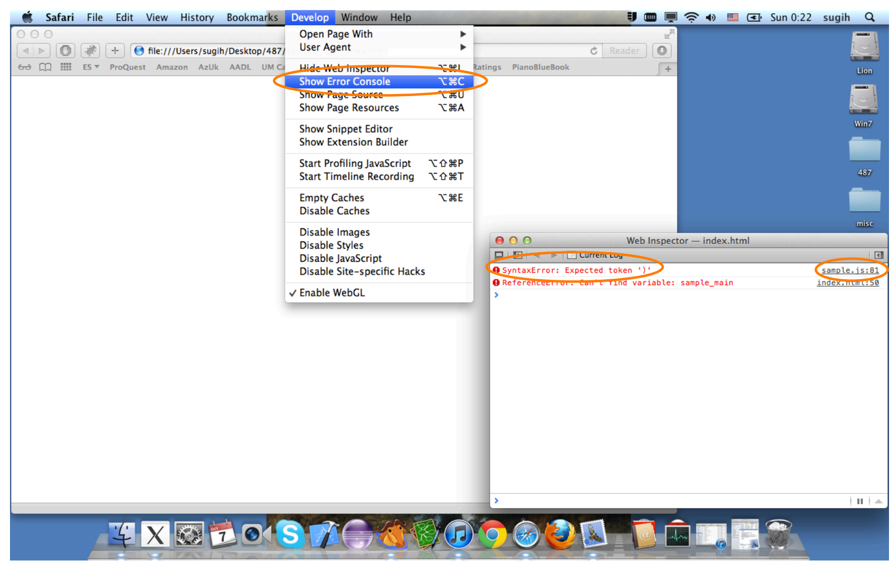
JavaScript Tools

The equivalent of getting compiler errors is viewing the "Developer/Error Console" of your browser (see next 3 slides)

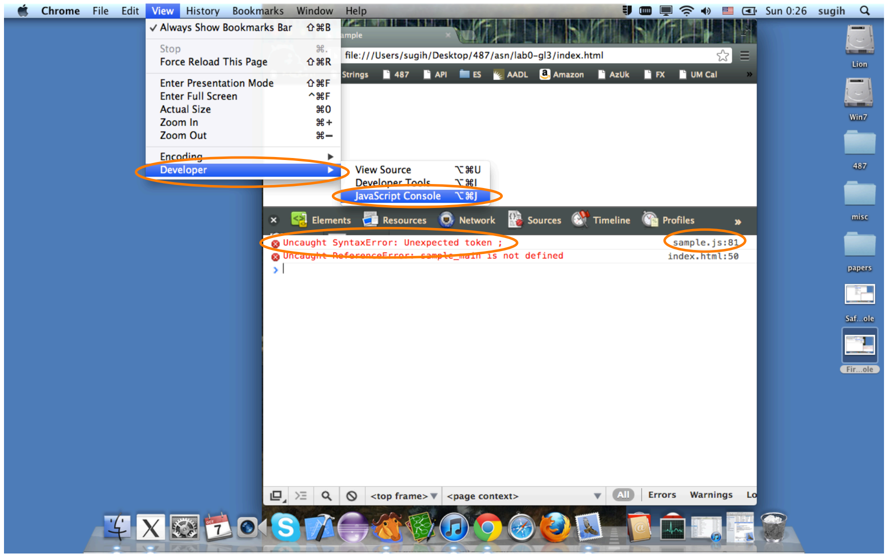
Leave the Console window up at all times!

- for more elaborate error messages, use Firebug (http://getfirebug.com), but it works well only with Firefox
- none of these tools will catch all of your typoes!

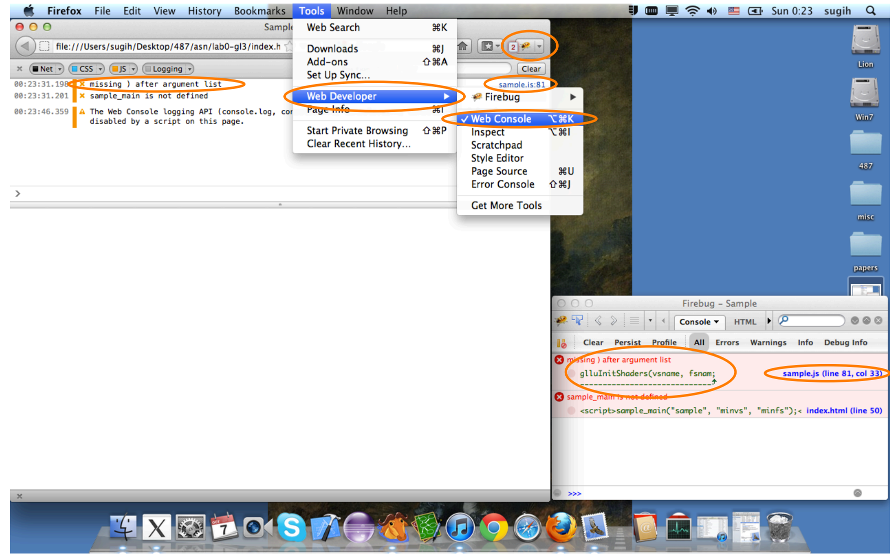
JavaScript Tools



JavaScript Tools



JavaScript Tools



JavaScript Tools

The equivalent of debugging by `cout/printf()` is to use `alert()` (there's also the as-yet non-standard `console.log()` (https://developer.mozilla.org/en-US/docs/Web/API/Console.log and http://jsconsole.com/remote-debugging.html))

jsFiddle (http://jsfiddle.net/vWx8V/) and other JS shells (https://developer.mozilla.org/en-US/docs/JavaScript/Shells) allow you to experiment with JS interactively

jsPerf (http://jsperf.com) compares the performance of different JS snippets

WebGL Tools

WebGL Playground (http://webglplayground.net) allows you to experiment with WebGL interactively

WebGL Inspector (http://benvanik.github.com/WebGL-Inspector/) allows you to view GL states similar to gDEBugger (http://www.gremedy.com) for OpenGL, but it hasn't seen development since 2012 and I'm having troubles using it

See also:

How you can lose context and how to handle it:
http://www.khronos.org/webgl/wiki/HandlingContextLost
http://www.khronos.org/registry/webgl/extensions/WEBGL_lose_context/

How to animate only when canvas is visible:
https://developer.mozilla.org/en-US/docs/DOM/window.requestAnimationFrame

For better performance, offload as much computing as possible from Javascript to GPU