WebGL

Hardware accelerated graphics in the browser

This chapter will give a brief introduction to WebGL programming and assumes that the reader is already familiar with OpenGL and C/C++ programming.

In the first part the basic browser technologies are explained briefly; HTML, CSS and JavaScript. Afterwards WebGL is explained with a focus on how it differs from OpenGL and OpenGL ES. Finally some of the best and most popular WebGL tools are described. Experience with modern OpenGL (3.1+) or OpenGL ES (2.0+) is assumed, so you should be familiar with the shader-based pipeline including GLSL, buffer objects and shader bindings.

One of the major benefits of using WebGL for graphics in the browser is its availability and dynamic behavior. For this reason the chapter is also available in a web-based version, which allow you to change the code examples and see the effect instantly. The web-version of the chapter is available here: **http://www.imm.dtu.dk/ mono/webgl_v100/index.html** (if prompted for username/password use webgl / webgl).

The examples are written as small as possible, which means that they don't use best practices of HTML. However the examples are guaranteed to work in an up-to-date version of Safari, Chrome and Firefox.

HTML

Hyper-Text Markup Language is still the most fundamental component of the world wide web. HTML describes the structure and content of a web page in a hierarchical structure.

```
<html>
        <body>
                <a href="/home">
                        Link
```

```
                </a>
                Hello
                <br>
                World
        </body>
</html>
```

A HTML document is a text-based document, which consists of html tags and text that provides structure and content. Due to its simple syntax an html-document is easy to read and write for both human and computers. A HTML tag is surrounded by < and >. A tag can be a single standalone tag (like the
 above - sometimes written as
) or it can be a pair consisting of a start tag and an end tag (such as <a>link (note the backslash in front of the end tag).

A html tag consists of a name and a number of attributes. In the example above the a-tag has the attribute href, which is a hyperlink. For a complete description of html-tags and their attributes, consult one of the HTML online references. One of the most important attributes is the id-attribute, which can be applied all html-tags. The id-attribute allows referencing to a specific html tag using a unique name.

Text elements usually contain visible text, but may also contain other data such as style sheets (<style>) or source code (<script>).

CSS

Another important component used in web browsers is Cascading Style Sheets (CSS). Where HTML provides the content and structure, the style sheet contains the styles of each element. Styles can be applied based on tag names, ids and classes of tags (a HTML tag can have one or more classes listed in the attribute class).

Figure 1: Example of HTML using CSS.

```
<html>
        <body>
                <style> /* style sheet */
                body {
                        color: blue;
                }
                a {
                        color: #FFoooo; /* Hex value */
                }
                </style>
                <a href="/home">
                        Link
```

```
                </a>
                Hello
                <br>
                World
        </body>
</html>
```

Style sheets are a very fundamental component when creating web pages, but are not used much in relation to WebGL.

JavaScript fundamentals

JavaScript is dynamically typed and its syntax and naming conventions was heavily influenced by C and Java (but other than that they are very different languages). JavaScript used to be interpreted but are now just-in-time compiled to machine code by JavaScript engines.

Web-browsers provides access to browser functionality and HTML document using the Document Object Model API (Aka. DOM API). WebGL is another JavaScript API available in modern web browsers.

The basic control structures are the same as used in C and Java, which makes it easy for C programmers to read and write simple JavaScript. The following gives a short example of the commonly used control structures:

Listing 1: if-statement

```
if (true) {
    var hw = 'Hello world';
    document.write(hw);
} else {
    var b = 'Bye!';
    document.write(b);
}
```

Listing 2: for-loop

```
for (var i=0;i<3;i++){
    var hw = 'Hello world '+i;
    document.write(hw);
}
```

Listing 3: while-loop

```
var i=0;
while (i<3){
    var hw = 'Hello world '+i;
```

```
    document.write(hw);
    i++;
}
```

JavaScript variables and types

In JavaScript types are associated with the values and not the variable. This means that a single variable can contains values of different types in different places in the program. In the example below the variable i first refers to the string "Hello world " and later refers to the value 0. The printed result is "Hello world 0".

```
var i = "Hello world ";
document.write(i);
i = 0;
document.write(i);
```

The following lists the different types of variables in JavaScript.

Type	**Examples**	**Copy semantic**	**Comment**
Boolean	true / false	Copy by value	
Number	3.14	Copy by value	64-bit precision
String	"xyz"	Copy by reference	immutable
Function	function foo(){ return 0; }	Copy by reference	immutable
Object	{pi: 3.14}; new Object()	Copy by reference	mutable
Arrays	[1,2,3] ;new Array(2)	Copy by reference	mutable

A variable that has not been assigned a value has the value undefined, which is different from the value null. The null value is also different from the numeric value 0. To read the type of a variable you can use the typeof function which returns a string. Example "document.write(typeof({}));" which writes the string "object".

JavaScript functions

JavaScript functions have a list of parameters and can optionally return a value. A function is defined using the function keyword. A function can be invoked using the name of the function and a comma-separated list of values as parameters wrapped in parenthesis.

Listing 4: Functions example

```
function f(a,b) {
    return a+b;
```

```
}
var res = f(1,2);
document.write(res)
var txt = f('a','b');
document.write(txt);
```

A JavaScript function is just a JavaScript object that can be invoked using the parenthesis operator. This means that you pass around function objects as any other object. Another detail is that function declaration returns the function object, which gives an alternative way of creating functions (see function g in the example below).

Listing 5: Functions as objects example

```
function f(a,b){
    return a+b;
}
var g = function(a,b){
    return a-b;
};
g = f;
document.write(g(1,2));
```

Functions are often used as callback handlers for instance when loading images or when using timers.

Listing 6: Callback functions example

```
function onReady(){
    var txtEl = document.createTextNode("ready");
    document.body.appendChild(txtEl);
}
setTimeout(onReady, 500); // call onReady after 0.5s
document.write("init ");
```

Another detail is that functions are the only way to define scope of variables (in other words curly-brackets does not define scope). This means that from a function you can access local variables in the function and variables defined in outer functions and global variables defined outside any function.

JavaScript objects

Objects are unordered sets of key-value pair. You can add or remove a key-value pair to and from an object. Keys must be strings but values can be any JavaScript type. Objects are created using new Object(), new Fn() (where fn is a function which acts a constructor) and which is the short notation for object construction.

The this variable refers to the current object (or the global context when used outside any object). In the code below pay special attention to how the same function f works differently depending on whether it is used as a method on an object or a global function.

Listing 7: Objects example. Result is "global 12"

```
var o = new Object();
// var o = {}; // alternative
o.x = 12;
var f = function(){
    document.write(this.x);
};
o.y = f;
var x = "global ";
f();
o.y();
```

To delete a property from an object you use the delete operator (example delete o.x;).

JavaScript doesn't have classes but instead uses a concept called prototypes to define archetypes of objects. Prototypes are useful when you want to create inheritance and can be used to mimic traditional object-oriented class hierarchies.

JavaScript arrays and typed arrays

An array in JavaScript is a special type of object, which contains an indexed set of objects. The values in an array, which may be of different types, can be accessed using the []-operator. An array has a length property, which returns the value of the largest used index plus one. Arrays in JavaScript is much more dynamic than arrays in C++ or Java; Arrays automatically grows to your needs and arrays can even be sparse (unused elements in an array will have the value undefined). An array also has a number of useful methods such as push(), pop() and sort().

Listing 8: Arrays example. Result is "12y2"

```
var a = new Array();
// var a = []; // alternative
a[0] = 12;
document.write(a[0]);
a[1] = 'y';
document.write(a[1]);
document.write(a.length);
```

When working with data used for WebGL you need to be able to known the data type (such as 32-bit floating point number) and that the data is located continuous in memory to ensure efficiency. Neither of these two requirements is guaranteed by JavaScript arrays.

When WebGL was introduced a new type of array was also introduced; Typed arrays. Typed arrays works much more like arrays in C++ or Java. The array has a given type (32 bit float or unsigned 16 bit integer) and data are located continuous in memory and has fixed length. Examples of typed arrays are Float32Array and Uint16Array and a typed array object is created using the new operator with the array length as parameter.

Listing 9: Typed array example. Result is "65535 3"

```
var a = new Uint16Array(2);
a[0] = -1;
document.write(a[0]);
a[1] = 3.14;
document.write('<br>'+a[1]);
```

The array types like Float32Array and Uint16Array actually works as view on a ArrayBuffer (which is essentially is a block of memory). You can have different views on the same ArrayBuffer, which can be useful when working with raw data.

WebGL

The WebGL 1.0 API and feature-set is almost identically to the OpenGL ES 2.0 API.

All WebGL functions and enums are accessed through a WebGL context object. A WebGL context is obtained by calling getContext("webgl") on a canvas object and store it in a variable (usually named gl). Some browsers only have experimental WebGL support, which uses another context name. A cross-platform way to get a context is getContext("experimental-webgl") || getContext("webgl").

The WebGL context object methods are named similar to OpenGL functions but without the gl-prefix (E.g. glClear(int flag) becomes gl.clear). The same is true of OpenGL enums (E.g. GL_COLOR_BUFFER_BIT becomes gl.COLOR_BUFFER_BIT).

Listing 10: WebGL Context creation example

```
<canvas id="n" width="50" height="50"></canvas>
<script>
var canvas = document.getElementById("n");
var gl = canvas.getContext("experimental-webgl") ||
```

```
        canvas.getContext("webgl");
gl.clearColor(1, 0, 0, 1);
gl.clear(gl.COLOR_BUFFER_BIT);
</script>
```

After a WebGL context has been created it is ready for WebGL. In order to setup an animation-loop, you can use the javascript function setInterval(fn, millis), which invokes the function fn every millis milliseconds. (A more sophisticated method for animation is requestAnimationFrame, which takes screen repaint events into account).

Listing 11: WebGL Context creation example - Interactive

```
<canvas id="n" width="50" height="50"></canvas>
<script>
var canvas = document.getElementById("n");
var gl = canvas.getContext("experimental-webgl") || canvas.getContext("webgl");
function updateClearColor(){
    gl.clearColor(Math.random(), Math.random(), Math.random(), 1);
    gl.clear(gl.COLOR_BUFFER_BIT);
}
setInterval(updateClearColor, 16);
</script>
```

A full WebGL reference can be obtained here:

- http://www.khronos.org/files/webgl/webgl-reference-card-1_0.pdf
- https://www.khronos.org/registry/webgl/specs/1.0/

WebGL geometry data

Just like any other OpenGL program you would need to provide some geometry-data and upload the data to the GPU before any rendering can take place.

In WebGL geometry data is often written using JSON notation. JSON is actually just a subset of JavaScript and provides a simple way to describe objects {}, objects with attributes {"someAttribute":"someValue", "anotherA": 1} and arrays [1, 2, 3]. A JSON object can only consist of objects, object-attributes, array, strings and numbers. Other JavaScript-code is not allowed in JSON.

Listing 12: Geometry data using JSON notation example

```
<script>
    var modelData = {
    "vertex":[0,0,0,
```

```
                1,0,0,
                0,1,0],
    "color":[0,0,1,
                0,1,0,
                1,0,0]
};
</script>
```

The modelData-variable contains two arrays with values for vertex positions and colors. Note that the arrays in modelData are JavaScript arrays. To use the data we need to first concatenate (or interleave) the vertex data and then transform the array to a float array. Afterwards the float array can be uploaded to a vertex buffer object.

Listing 13: Building vertex buffer objects example

```
// var modelData =  ...
var canvas =  document.getElementById("n");
var gl = canvas.getContext("experimental-webgl") || canvas.getContext("webgl");

// prepare data
var data = modelData.vertex.concat(modelData.color);
var dataFloat = new Float32Array(data);

// create and upload vertex buffer
var vertexBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
gl.bufferData(gl.ARRAY_BUFFER, dataFloat,
                                gl.STATIC_DRAW);
```

WebGL shaders

Another important element is shaders. A simple way to store shaders in a website is to add them to the html-document using a script element with a unknown type (such as <script type="shader">) which makes the browser ignore the element. The content of the element data can still be read using JavaScript.

Listing 14: Embedding shaders in HTML document example

```
<script type="shader" id="vshader">
attribute vec4 vertex;
attribute vec4 color;
varying vec4 vColor;
void main(void) {
```

```
        gl_Position = vertex;
    vColor = color;
}
</script>
<script type="shader" id="fshader">
precision highp float;
varying vec4 vColor;
void main(void) {
        gl_FragColor = vColor;
}
</script>
```

Shader source code is now read from JavaScript and then compiled and finally linked. The shader source is read by first getting a reference to the html-element using document.getElementById(id) and afterwards calling .textContent on the result. To simplify the code the error check after shader compilation and linking has been omitted.

Listing 15: Reading and compiling shaders example

```
// read shader source
var vshaderTag = document.getElementById('vshader');
var fshaderTag = document.getElementById('fshader');

// create, compile and link shader
var vShader = gl.createShader(gl.VERTEX_SHADER);
gl.shaderSource(vShader, vshaderTag.textContent);
gl.compileShader(vShader);
var fShader = gl.createShader(gl.FRAGMENT_SHADER);
gl.shaderSource(fShader, fshaderTag.textContent);
gl.compileShader(fShader);
shaderProgram = gl.createProgram();
gl.attachShader(shaderProgram, vShader);
gl.attachShader(shaderProgram, fShader);
gl.linkProgram(shaderProgram);
gl.useProgram(shaderProgram);
```

Finally we need to set the vertex attribute pointers and draw the geometry.

Listing 16: Setup vertex buffer object and draw geometry example

```
// setup vertex buffer
var vertexPos = gl.getAttribLocation(shaderProgram, "vertex");
var colorPos = gl.getAttribLocation(shaderProgram, "color");
var sizeOfFloat = 4;
var itemSize = 3;
```

```
var colorOffset = sizeOfFloat * modelData.vertex.length;
gl.enableVertexAttribArray(vertexPos);
gl.vertexAttribPointer(vertexPos, itemSize, gl.FLOAT, false, 0, 0);
gl.enableVertexAttribArray(colorPos);
gl.vertexAttribPointer(colorPos, itemSize, gl.FLOAT, false, 0, colorOffset);

// draw
gl.clear(gl.COLOR_BUFFER_BIT);
gl.drawArrays(gl.TRIANGLES, 0, 3);
```

Putting it all together we get a minimalistic WebGL example:

Figure 2: WebGL with simple geometry.

WebGL textures

So far we have only considered the case where data has been available when we want to use it. In many cases we first need to load and initialize data before they can be used. Loading an image used for WebGL textures is a good example of this. First we need to create an JavaScript image object, then register a callback function (called when image is loaded and ready) and finally set the image source, which will start an asynchronously image loading.

Listing 17: Image loading example

```
<p id="img_data"/>
<script>
var imgData = document.getElementById("img_data");
var img = new Image();
img.onload = function(){
    imgData.textContent = img.width+"x"+img.height;
};
img.src = "http://www.dtu.dk/favicon.ico";
```

```
</script>
```

For security reasons WebGL only accepts images hosted on the same server or on third party servers requested using a technique called CORS (https://developer.mozilla.org/en-US/docs/HTML/CORS_Enabled_Image).

A third technique involves base 64 encoding a image (simply turning the raw bytes into a text string). The cleaver thing about this technique is we can keep everything in a single-html file and it also works when a browser loads the file from a local file-system instead of a web-server.

First you need to encode an image (png or jpeg) to base 64 using one of the many free online tools (E.g. http://www.base64-image.de/). Then use the base 64 URL (starting with "data") as an image source.

Listing 18: Image loading (base 64) example

```
<p id="img_data"/>
<script>
var imgData = document.getElementById("img_data");
var img = new Image();
img.onload = function(){
    imgData.textContent = img.width+"x"+img.height;
};
img.src = ''+
'ABCAYAAAAfFcSJAAAACklEQVR4nGMAAQAABQABDQottAAAAABJRU5ErkJggg==';

</script>
```

In OpenGL you need to work with raw image data that will be uploaded to the GPU. In WebGL you would normally just use a Image object instead (you are still able to raw image data using a typed array - in case you want to create a texture procedurally on the CPU).

The following shows how a image can be loaded as a WebGL texture.

Listing 19: Image to texture example

```
var img = new Image();
var textureId;
img.onload = function(){
    textureId = gl.createTexture();
    gl.bindTexture(gl.TEXTURE_2D, textureId);
    gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);
    gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE, img);
```

```
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.LINEAR);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR_MIPMAP_NEAREST);
    gl.generateMipmap(gl.TEXTURE_2D);
};
img.src = ''+
'ABCAYAAAAfFcSJAAAACklEQVR4nGMAAQAABQABDQottAAAAABJRU5ErkJggg==';
```

A complete example that uses texture can be seen here:
http://www.imm.dtu.dk/ mono/webgl_v100/11webgl-textures.html

WebGL libraries

You should now have a basic understanding of the basics of WebGL. But to do anything useful besides these limited examples you will need a few libraries.

glMatrix

First of all you will need a linear algebra library with vectors and matrices with dimensions up to 4. If you are a C++ programmer you will probably be a little disappointed finding out that JavaScript does not have operator overload, which makes the code rather ugly.

One of the most commonly used linear algebra libraries for JavaScript is called glMatrix (http://glmatrix.net/). The current version of glMatrix is 2.x. As many other JavaScript libraries it exist in two different versions, a normal version with human readable code and in a minified version, where the source code has been shortened to decrease the file-size and increase the download speed. An example of using the library can be seen below.

The API usually has the output variable as the first parameter. This may seem very odd at first but this allows programmers to use pre-allocated matrices, which avoids temporary matrix objects that eventually has to be cleaned up by the garbage collector.

Listing 20: glMatrix example

```
<script src="http://cdn.jsdelivr.net/glmatrix/2.2.0/gl-matrix.js"></script>
<script>
var persp = mat4.create();
mat4.perspective(persp, 45, 4/3, 1, 100);
document.write(mat4.str(persp));
document.write("<br>");

var mv = mat4.create();
mat4.identity(mv);
mat4.translate(mv, mv, [0, 0, -10]);
```

```
mat4.rotate(mv, mv, Math.PI/2, [0, 1, 0]);
mat4.scale(mv, mv, [2, 2, 2]);
document.write(mat4.str(mv));
</script>
```

To save network-bandwidth the code-example above downloads the library from a CDN-server (Content Delivery Network). The idea is similar to shared DLL files used in operating systems; here commonly used libraries are stored on public servers, which means that the browser only needs to download (and cache) the library once even through it might be used on multiple websites.

WebGL debugging

Debugging JavaScript in modern web browsers is very easy, since all browsers include a build-in debugger that can be started at any point in time. From the debugger you can inspect variables, modify variables and even run on-the-fly snippets of JavaScript code. Modern web browsers even include advanced profilers, which makes it easy to pinpoint slow code and resource leaks.

In contrast to normal JavaScript, WebGL can be fairly tricky to debug. The main reason is that WebGL is based on the C-API from OpenGL ES 2.0. This means that WebGL resources are accessed using resource handles (instead of JavaScript objects) which prevents the debugger to easily see the value of a variable. Another challenge is that WebGL state errors need to be tested using the gl.getError (in contrast to exceptions used in everywhere else in JavaScript). In WebGL both the type and the value of function parameters can be wrong.

One very useful tool for testing WebGL state errors is WebGL-Debug - a small script library developed by Khronos. The library replaces the actual WebGL context with a proxy object, which will then test for errors after each function call. The following gives a short example of how a WebGL state error is detected and printed to the browsers JavaScript console.

Listing 21: WebGL debug example

```
<canvas id="n" width="50" height="50"></canvas>
<script src="webgl-debug.js"></script>
<script>
var canvas = document.getElementById("n");
var gl = canvas.getContext("experimental-webgl") || canvas.getContext("webgl");
gl = WebGLDebugUtils.makeDebugContext(gl);
function updateClearColor(){
    gl.clearColor(Math.random(),
```

```
            Math.random(), Math.random(), 1);

    // error should be COLOR_BUFFER_BIT
    gl.clear(gl.COLOR_CLEAR_VALUE);
}
setInterval(updateClearColor, 16);
</script>
```

Another tool is WebGL-Inspector, which allows you to inspect WebGL including viewing textures, models and shaders. WebGL-Inspector can be installed as a browser plugin or be used as a JavaScript library.

Both tools are very useful for debugging, but should be disabled for production code, since they add a significant performance overhead.

WebGL frameworks

WebGL is a very low-level API and much more low-level than other APIs in the web-browser. So when starting a large project which includes WebGL an important question is if WebGL has the right level of abstraction for the project or you instead should use a WebGL framework, which has implemented many of the standard features (model-loading, scene graphics, etc). On the other hand a framework may have taken some design choices or may have other limitations which conflicts with your needs.

One of the most widely used WebGL frameworks is Three.js, which has a very simple API and is easy to get started. Eric Haines (one of the authors of Real-Time Rendering) has even created a free Audacity course about Computer Graphics taught in Three.js in the browser.

There exist a variety of alternative WebGL frameworks each with different strength and weaknesses.

Listing 22: Threejs example

```
<canvas id="n" width="300" height="300"/>
<script src="three.min.js"></script>
<script>
var scene = new THREE.Scene();
var camera = new THREE.PerspectiveCamera(75,
    window.innerWidth/window.innerHeight, 0.1, 1000);

var elem = document.getElementById('n');
var renderer = new THREE.WebGLRenderer({canvas:elem});
```

```
var geometry = new THREE.CubeGeometry(1,1,1);
var material = new THREE.MeshBasicMaterial(
    {color: 0x00ff00});
var cube = new THREE.Mesh(geometry, material);
scene.add(cube);

camera.position.z = 5;

var render = function () {
    requestAnimationFrame(render);

    cube.rotation.x += 0.1;
    cube.rotation.y += 0.1;

    renderer.render(scene, camera);
};

render();
</script>
```

Project: WebGL

Objectives

The objective with this exercise is to give you some experience with writing WebGL.

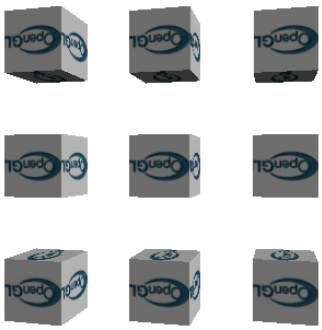

Figure 3: WebGL exercise.

Part 1

Try to run the example code located here (note that the code requires Firefox or Chrome since it uses a high-performance timer):

http://jsfiddle.net/mortennobel/29LMA/

Currently the code tries to benchmark the performance, however it needs to flush the pipeline before starting the timer and before ending the timer. Update the code to correctly flushing the pipeline.

Part 2

Currently the scene makes 9 draw-calls per frame which renders 9 cubes to the screen. Change the number of visible cubes to a much higher number of cubes (between 500 and 5000).

The code is written in a slightly inefficient way. Try to optimize the code. (Hint: the first thing you should do is to disable the WebGL-

debug).

For each optimization, list the performance improvement both in milliseconds and percent.

Hint: Mozilla Developer Network have a good article about WebGL best practices (https://developer.mozilla.org/en-US/docs/Web/WebGL/WebGL_best_practices).

Part 3

Create the same program in Qt and compare the runtime performance between WebGL and OpenGL (running in Qt). A sample project will be provided.

Try to see how the performance scales in terms of number of draw-calls for both WebGL and OpenGL.

Part 4 - Optional

A simple stateless particle demo, which emits particles when dragging mouse on the black canvas:

http://jsfiddle.net/mortennobel/YHMQZ/

Possible extensions:

- Change color, change particle size, change particle

Deliverables

The source code and answers for each part. Optionally a link to jsfiddle (you have to sign up to save your changes)