

Graphics Programming for the Web

Single sentence summary:

We introduce and demonstrate HTML5 web technologies that enable graphics developers to produce complex, general-purpose graphics applications for the web.

Abstract

With the advent of HTML5 and the ever-improving browser performance, the world wide web has emerged as an ideal platform for showcasing graphics applications. Several graphics applications that earlier may have been too slow to be written in anything but native code may now be fast enough to run as web apps. This course hopes to help those developers who wish to develop graphics applications for the web. We introduce the dominant graphics technologies that are accessible via web programming on most modern browsers.

We start with a quick primer on general-purpose web programming. We introduce HTML parsing, CSS, DOM- and render-tree construction, and the use of Javascript for generating dynamic web content. The bulk of this course describes the web technologies specific to graphics:

1. HTML5 Canvas: path API, image editing, animation, comparison with SVG
2. CSS3: transitions, animations, 3D transforms and the new css-shaders
3. WebGL: getting started, achieving high performance, advanced 3D techniques
4. WebCL: the formal specifications, what's implemented, and what's to come

For each topic, we provide a significant number of code examples that illustrate the relevant graphics capabilities. The course participants will see several interactive demos during the course, and will also be able to copy and paste our code snippets and execute them easily inside any modern web browser

We do not require the course participants to have any knowledge of web programming. Graphics expertise is not required or assumed. However, we expect this course will be most useful to people already familiar with graphics concepts.

Online Course Repository:

The latest version of these notes and other helpful resources will be hosted at:

<http://www.khronos.org/developers/library/2012-siggraph-course-graphics-programming-for-the-web>

Sample Course Schedule

9:00 am: Introduction

9:05 am: Canvas &SVG

9:30 am: CSS

10:00 am: WebGL Part 1

10:30 am: break

10:45 am: WebGL Part 2

11:30 am: WebGL

12:15 pm Course End

(we will allot time for questions at the end of each section talk)

Instructors:

Pushkar Joshi is a graphics research engineer at Motorola Mobility. His research focuses on geometric modeling, with an emphasis on casual modeling for novice users. Prior to Motorola Mobility, he was a computer scientist at the Advanced Technology Labs at Adobe, where he developed the core geometry engine for the Repoussé feature of Adobe Photoshop CS5. Pushkar has a Ph.D. in computer science from the University of California, Berkeley.

Mikael Bourges-Sevenier is a multimedia software architect at Motorola Mobility focusing on user experience and multicore applications on mobile devices. He is co-editor of WebGL specification. Prior to Motorola, he was editor of various standards such as MPEG-4, X3D, U3D, and their implementation in products of Adobe, Sun, iVAST, France Telecom. Mikael has an MS. in mechanical and electrical engineering from ECAM Lyon, France and a MS. in signal and image processing from University Rennes I, France.

Ken Russell is a software engineer on the Chrome web browser team at Google, Inc, and is currently serving as the chair of the WebGL working group at Khronos. Ken has over fifteen years of 3D graphics programming experience. He holds a Bachelor of Science in Electrical Engineering and Computer Science from the Massachusetts Institute of Technology and a Master of Science in Media Arts and Sciences from the MIT Media Lab.

Zhenyao Mo is a software engineer on the Chrome web browser team at Google, Inc. Zhenyao earned his Ph.D. from University of Southern California, during which his research focused on computer graphics. After graduation, he continued his enthusiasm in 3D graphics; for the past 2.5 years working at Google, his main effort is implementing and improving WebGL in webkit and Chrome.

Introduction

Web programming has evolved from small tasks for dynamic web page content to full applications for accomplishing complex tasks. Vast improvements in Javascript performance in all major browsers, including those for mobile platforms, have made it possible for the web to emerge as a common computing platform for most hardware and software configurations. With the advent of HTML5 and CSS3, all dominant browsers provide a drawing API, complete with manipulation of web page elements in 3D space. Through WebGL, most browsers also provide a direct interface to the graphics hardware on the client computer. Upcoming WebCL technologies make it even easier for people to convert their multithreaded native applications into web applications. These factors make the web an especially ideal platform for showcasing advanced graphics applications.

This course is targeted towards programmers who wish to write general-purpose graphics applications for the web. By “general-purpose” we mean applications that may consume a large amount of computing resources and are typically written in native (C/C++) code by experienced graphics programmers. We wish to help those programmers write similar applications for the web by introducing graphics web programming.

We do not require the course participants to have any knowledge of web programming. Graphics expertise is not required or assumed; however, we expect this course will be most useful to people familiar with graphics concepts.

After a quick primer on web programming, we introduce the HTML5 canvas element and describe how it can be used to enable interactive drawing tools, image editing, and interactive 2D animations. Next we introduce CSS3 and show how it can be used to produce dynamic 3D user interfaces, 2D image filters, and general-purpose CSS shaders. Next we show how 3D graphical content can be displayed and manipulated via WebGL. Along with the initial setup necessary for all WebGL programs, we also give commonly applicable hints for improved graphics performance. Finally, we introduce the new WebCL specification that brings the parallel computation of OpenCL to web programs.

Web Programming Primer

In this section we introduce the technologies that are essential for web programming. Note that we describe web programming at a very high level, and this section is intended for someone completely new to web programming. Readers familiar with these topics can skip ahead to the next section where we will start discussing web programming topics specific to graphics.

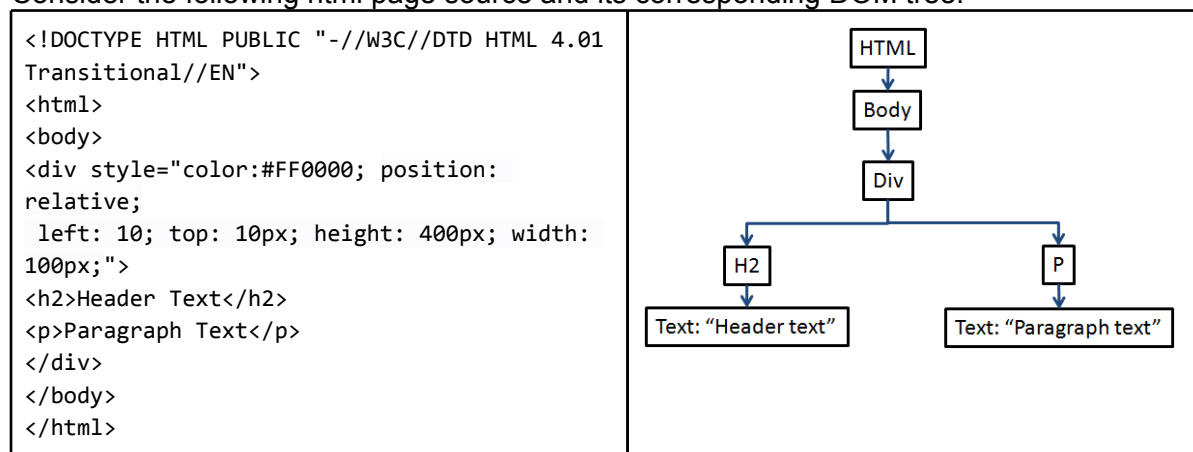
HTML and CSS

Most web content is in the form of plain text HTML (Hyper Text Markup Language). HTML is the language used to mark-up content to be displayed on the web. The web content is mostly text but can also include images or general binary objects (such as video). The mark-up is done by a content identifier (also known as a tag) and usually instructs a web browser how to display the content.

For example, an HTML page may contain tags like this: `<p style="bold">sample text</p>`. The "sample text" in between the start (e.g. `<p style="bold">`) and stop (e.g. `</p>`) tags will be displayed using the presentation style specified for the tag (bold, in this case). The display style for each tag of the document is given to the browser by the CSS (Cascading Style Sheet) for that document. The exact rules for writing HTML and CSS are defined in the specifications published by the World Wide Web Consortium (W3C).

A browser parses the HTML page and adds content corresponding to the tag as a new element of the page's document object model (DOM). The DOM is the programming interface for accessing and manipulating the contents of an HTML page. The DOM allows us to access things like the properties of a specific element, the number of a certain type of a tag, etc. The DOM elements form a DOM tree that gives us a hierarchical overview of the contents of the page.

Consider the following html page source and its corresponding DOM tree:



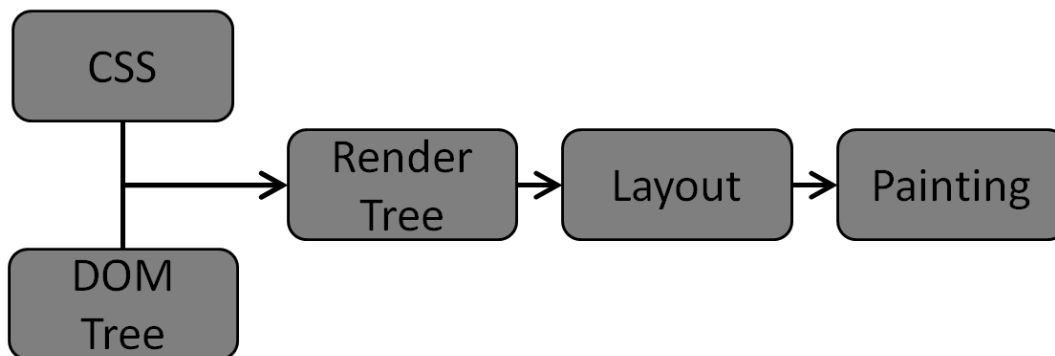
Given the DOM tree, the browser produces a render tree: a tree of visual elements in the order that they are painted. For the purpose of this course, think of these visual elements as *rectangles* (one for each HTML block) that the browser will paint. As mentioned before, the

rules used by the browser for displaying the visual element (e.g. the background color of the rectangle) are provided by the presentation style associated with that element.

The standard practice is to not include the presentation style information directly in the HTML tags (as was done above), but instead to describe the presentation style for each HTML tag used in the web page in a separate file, called the stylesheet. This reduces the HTML page size, and makes it easier to specify or change the style for an entire website (all web pages will refer to the same style sheet).

The stylesheet that defines the style for every element of the DOM tree is given by the Cascading Style Sheet (CSS) for that document. For example, as defined by the W3C box model (<http://www.w3.org/TR/CSS2/box.html>), each rectangle corresponding to certain HTML tags (known as block tags) will be surrounded by additional rectangles (namely padding, border, and margin). In addition, the style can contain information like fonts used, color of the text, style of the text, and so on.

The term “cascading” in CSS refers to the order in which presentation rules are applied. The presentation rule applied to an element of the render tree is automatically applied to its children in the render tree, unless the child nodes have different presentation rules specified in the CSS. All browsers will have a default stylesheet that defines the presentation style for DOM elements not included in a particular web page’s stylesheet. CSS will be useful for us later in this tutorial, when we discuss the new graphics-specific features of CSS.



After the render tree is generated, most browsers have a separate “layout” process that assigns a position (X and Y location within the browser window) and width and height to each element of the render tree. Think of this as a process of placing rectangles at certain locations within the browser window and assigning a width and height to each rectangles. After the layout process, the individual elements of the render tree are painted in the browser window in the order specified by the render tree.

Javascript

Javascript is script language used to programmatically (i.e. dynamically) add or change content for a web page or change the properties of existing content. For example, if you wanted to change the text color contained in this tag:

```
<div id= "divName" style= "color: red" >  
Sample Text  
</div>
```

when the user's mouse pointer hovered over the text, you could write a "onmouseover" handler like this:

```
<div id= "divName" onmouseover=  
    "document.getElementById('divTag').style.color='Blue';">  
    Sample Text  
</div>
```

which will change the "Sample Text" color from red to blue on mouse over.

Through Javascript, we can access the DOM for the web page by calling functions like "document.getElementById".

Over the years, the scope of Javascript has grown significantly. Instead of being a language only for making small changes to a mostly static web page, it is now used for performing significant calculations or building large web frameworks. This growth in scope is due, in large part, to the huge improvement in the performance of the Javascript interpreter in web browsers (a performance improvement of roughly 25% per year). The improvement in the Javascript interpreter performance is one of the things that enables us to write complex graphics applications for the web.

Note that many traditional C/C++/Java programmers find Javascript to be frustrating. Javascript is not a subset of Java (the name is, for the most part, a misnomer). Several aspects that many programmers take for granted in other programming languages (like strong types, block scope for variables) are not present in Javascript. Javascript does support some paradigms of modern programming languages (like object-oriented design and inheritance) but not in an intuitive, straight-forward manner. Moreover, browsers are extremely error tolerant, and Javascript compilation errors are not reported until the offending line is executed. Therefore, learning Javascript can be challenging for many software developers.

That being said, the web community has built several useful tools for helping Javascript programmers. Here are some of them:

Testing for support on different platforms (which browsers and platforms support with feature)
<http://caniuse.com>

Testing code performance

<http://jsperf.com>

(includes thousands of saved test cases, including those for graphics)

Talk on writing high-performance Javascript:

<http://www.yuiblog.com/blog/2010/04/21/video-hpjs/>

Canvas and CSS

Pushkar Joshi, Motorola Mobility

[HTML5 Canvas](#)

[Drawing Tool](#)

[Fill and Stroke Styles](#)

[Layout and Transformations](#)

[Image Editing](#)

[Animation](#)

[Comparison with SVG](#)

[CSS](#)

[Planes in Space](#)

[Animations](#)

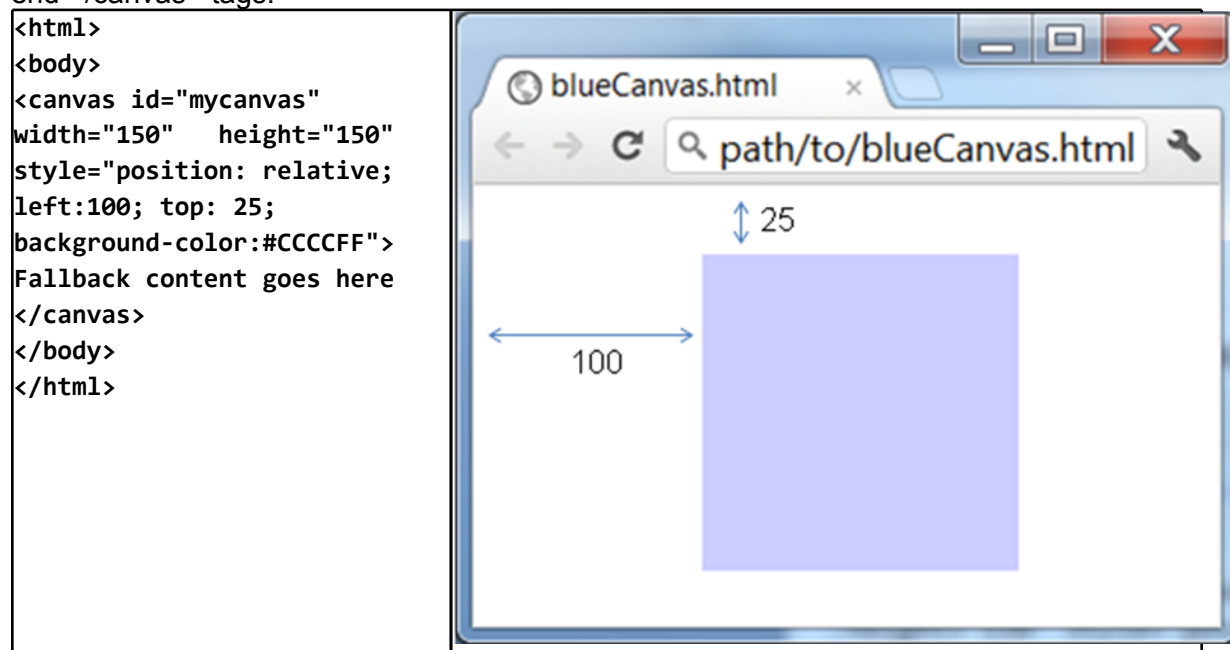
[Image Filters](#)

[General-Purpose CSS Shaders](#)

HTML5 Canvas

Once we are introduced to HTML, CSS, and Javascript, we are ready to learn about the canvas element that was introduced in HTML5. The canvas element is the easiest way to obtain an interactive drawing surface for a web page on a modern browser, and is currently extensively used for creating web-based games.

Similar to other block content tags like `<div>` or `<p>`, the `<canvas>` tag identifies a rectangular region of the browser window. Standard CSS operations (like setting the width, height, background color, and position) that can be performed on standard content tags like `<div>` can also be performed on the `<canvas>` tag. In case the browser cannot display the canvas (i.e. older browsers), we display a fallback message contained within the beginning `<canvas>` and end `</canvas>` tags.



The HTML document on the left produces a blue square canvas shown on the right. The canvas is offset by 100 pixels from the left and 25 pixels from the top, as specified by its style in the `<canvas>` tag above. Notice that with older browsers that do not support HTML5, the text "Fallback content goes here" will be displayed instead of the canvas.

Unlike the other HTML tags, the `<canvas>` tag offers a drawing context that can access and paint the individual pixels inside the canvas. People familiar with OpenGL or DirectX will be familiar with the notion of a drawing context. A drawing context is essentially the "surface" on which you can draw/paint your pixels. The standard method for accessing the drawing context is through Javascript. We have added some Javascript to the HTML document from earlier. This script queries the DOM and then calls the "getContext()" function of the canvas object:

```
<html>
<head>
<script type="application/javascript">
function draw() {
```



```
var canvas = document.getElementById("mycanvas");
if (canvas.getContext) {
    var context = canvas.getContext("2d");
    //issue drawing commands here...
}
}
</script>
</head>
<body onload="draw();" >
  <canvas id="mycanvas" width="150" height="150" style="position: relative; left:100px; top:
25px; background-color:#CCCCFF">
    Fallback content goes here
  </canvas>
</body>
</html>
```

Currently, two types of contexts are supported: a 2d context that offers the ability to manipulate the canvas like a bitmap, and a WebGL context that offers the ability to draw in 3D. In this section, we will cover the 2D drawing context, and WebGL will be covered in a separate section.

The coordinate system of the 2D context has its origin in the top left corner of the content of the box. The coordinates of any objects drawn in this context must lie in the range [0, canvas width] and [0, canvas height]. Any objects that do not lie within this range will be ignored and clipped.

The HTML5 canvas uses the “immediate” mode of drawing: the drawing commands are executed immediately after being issued, and the system saves no information about what was just drawn. The only state of the canvas saved by the browser is the color of the pixels inside the canvas. Later, we contrast this with SVG, which uses the “declarative” or “retained” graphics mode. Unlike the canvas element, every SVG element can be referenced through the DOM and edited later on.

In the rest of this section, we will describe some of the functionality possible with the HTML5 Canvas API that is particularly relevant for graphics developers.

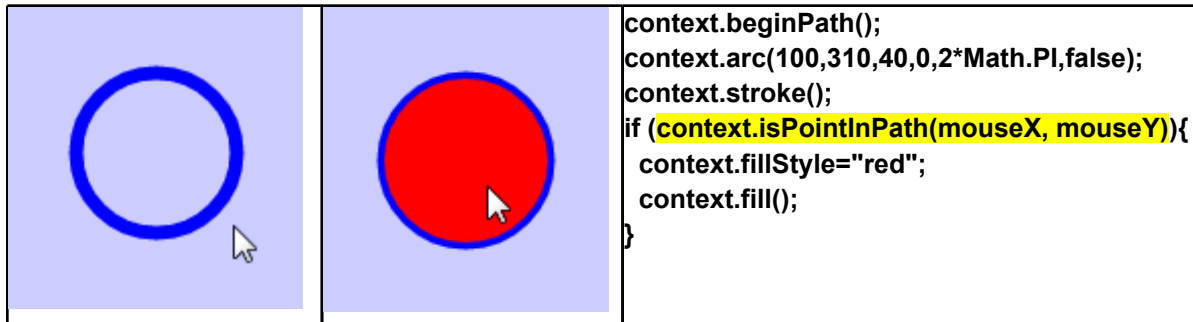
Drawing Tool

HTML5 introduced a quite powerful API for drawing and filling paths. Using this API, we can construct a vector design or sketching web application. See <http://mugtug.com/sketchpad/> for an example. If you are familiar with legacy drawing APIs like xLib or vector drawing APIs like PostScript, you will recognize the format of the path API. Similar to those APIs, we mimic the pen and plotter interface where every new object is drawn by first lifting and moving the drawing pen to the start location and then tracing along the path to be drawn.

	<pre> context.lineWidth=7; //the nose polyline context.beginPath(); context.moveTo(250,275); context.lineTo(200,375); context.lineTo(250,375); context.stroke(); //convert deg to radians var d2R = Math.PI/180; //the eyebrow: circular arcs context.beginPath(); context.arc(190, 220, 40, 210*d2R, 300*d2R); context.stroke(); context.beginPath(); context.arc(310, 220, 40, 240*d2R, 330*d2R); context.stroke(); //the eyeball: circles context.beginPath(); context.arc(190, 250, 40, 0, 2*Math.PI, false); context.stroke(); context.beginPath(); context.arc(310, 250, 40, 0, 2*Math.PI, false); context.stroke(); //the lower lip: cubic Bezier curve context.moveTo(200, 400); context.bezierCurveTo(225,475,275,475,300,400); //the upper lip: quadratic Bezier curve context.moveTo(200,400); context.quadraticCurveTo(250,450,300,400); context.stroke(); </pre>
--	---

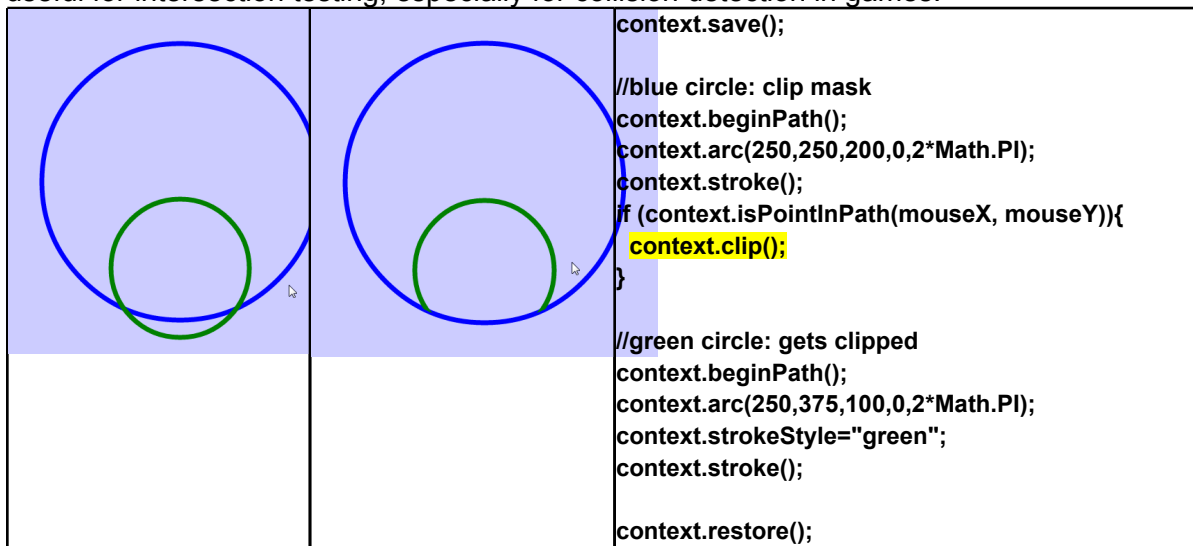
In the figure above, the code on the right produces the line drawing on the left for a 500x500 pixel canvas. This example demonstrates the ability to draw polylines, circular arcs (including full circles), cubic Bezier paths, and quadratic Bezier paths. Notice the use of the “beginPath()” function to indicate that a new path is being drawn for cases where calling “moveTo()” would be more complicated. Prior to “beginPath()” we must call the “stroke()” function to render the previous path.

The path API includes two geometric functions that are commonly needed for graphics tasks, so they are worth mentioning here.



Tracking when the mouse pointer enters a path by using the `isPointInPath()` function using the current mouse pointer coordinates. If the mouse is detected to be inside the path, we fill the path with a solid red color.

The “`isPointInPath(x,y)`” function returns true if the input (x,y) position is inside the path, assuming a non-zero winding number rule (i.e. same rule used for filling the path). This can be useful for intersection testing, especially for collision detection in games.






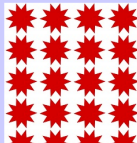
When the mouse pointer enters the blue path, we specify the drawing context to clip all subsequent drawing calls against the blue path. The clipping can be turned off by calling the context “`restore()`” function.

The “`clip()`” method is used to indicate that only the part of the canvas that’s inside the path will be rendered to the canvas. Make sure to include the “`save()`” function prior to calling the “`clip()`” method, so the clipping can be turned off by calling the corresponding “`restore()`” function.

Fill and Stroke Styles

Whatever shape has been added to the path so far will be filled when you issue the `fill()` command. Even open paths can be filled – for the purpose of the fill, the path is assumed to be closed by connecting the last point to the first point. The fill rule for complex (self-intersecting) paths is the non-zero winding number rule – the region of the path that has a non-zero winding number is filled. Obviously, this is independent of the orientation of the path.

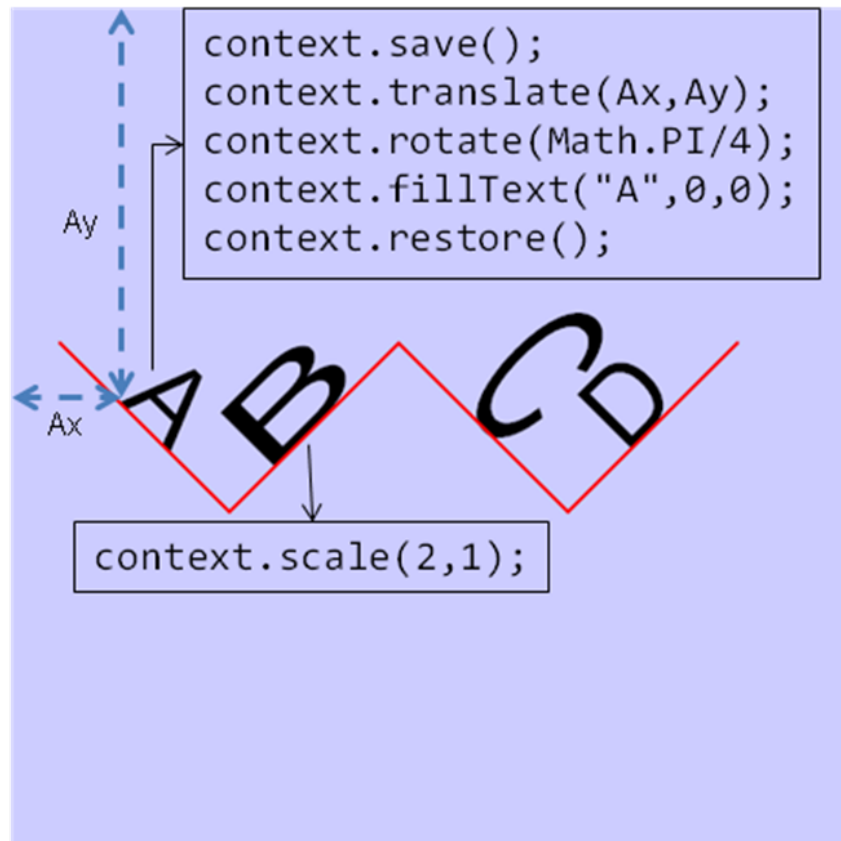
The shape can be filled with a solid color, a gradient, or a pattern (tiled images), as shown below:

	<pre>context.fillStyle="blue";</pre>
	<pre>var gradient = context.createRadialGradient(w/2,h/2,0, w/2,h/2,w/2); gradient.addColorStop(0, "blue"); gradient.addColorStop(1, "rgba(255,255,255,1.0)"); context.fillStyle = gradient;</pre>
	<pre>var gradient = context.createLinearGradient(w/4,h/2,w*0.75,h/2); gradient.addColorStop(0, "blue"); gradient.addColorStop(1, "rgba(255,255,255,1.0)"); context.fillStyle = gradient;</pre>
	<pre>var img = new Image(); img.src = 'star.jpg'; var myPattern = context.createPattern(img,'repeat'); context.fillStyle = myPattern;</pre>

The same rules that apply for “fillStyle()” also apply for “strokeStyle()”. That is, the stroke region for a path can be filled with a solid color, a gradient (radial or linear), or a repeating image pattern.

Layout and Transformations

Any shape drawn on the canvas can be transformed in order to position it anywhere within the canvas coordinate space. In this way the Canvas API can be used to layout 2D graphical elements on the screen.



Let us consider the task of typing the letters A, B, C, D along a zigzag path, as shown in the figure above. While we could use the path API to draw the letters, we'll take the simpler option of using the text api (i.e. the 'fillText()' or 'strokeText()' functions) built into the 2D drawing context.

The transformations affect the coordinate system of the drawing context. See the code block above for the letter a typical call to translate and rotate the letter A in order to place it in the proper position. Except in few cases, the order of the transformations is important. For example, calling the rotate function before the translate function would not have rotated the letter in place as above.

Experienced graphics developers will see the similarity between this transformation model and that present in prevalent graphics APIs like OpenGL or DirectX. Similar to those graphics API, we can concatenate the entire transformation and specify it as one homogeneous (3x3) matrix via the "transform()" and "setTransform()" functions in the Canvas API. Both functions take six arguments (the number of degrees of freedom available for 2D transforms). The difference between the "transform()" and "setTransform()" function is that the former concatenates the specified transformation to the current transformation, while the latter sets the specified transformation as the only transformation (we lose the history of the previous transforms).

You may have noticed calls to "save()" and "restore()". The "save" function pushes the current drawing context state (including the transformation, along with some other state variables) onto a stack. Subsequent calls to change the state (e.g. via additional transformations) will concatenate to the current state, but the original state can be recovered by the "restore()" function that will pop the top of the stack and set the current state to

the popped off value. Again, experienced graphics developers will notice the similarity between “save()” and `glPushMatrix()` and “restore()” and `glPopMatrix()` in OpenGL.

Image Editing

The Canvas API allows random access to the byte-level RGBA values of the individual pixels within the coordinate space of the canvas. Therefore, we can set the colors of any part of the canvas on a pixel-by-pixel level. We can also load arbitrary images into the canvas and manipulate their pixel values. Given these functions, we can implement a comprehensive set of image editing features using the canvas API.

Here are some code snippets that you will need in order to perform any image editing functionality:

Loading and displaying images

```
var imageObject = new Image();
imageObject.src = "<path_to_image_file>";
imageObject.onload = function() {
    context.drawImage(imageObject,0,0);
}
```

In the above code, we first create an image object and specify the path to the actual image file. Also, “context” is the standard Canvas 2D context, the “0,0” in the `drawImage` call specifies the top left corner of the image (at the canvas origin in this case), and the image is drawn only when it is fully loaded (i.e. is in the `onload` event handler for the image object).

Access pixels

Now that the image is drawn on the canvas, we get a pointer to its pixels. At any time we can obtain a 1D array of pixels that contain the current color values of the canvas element by using the `getImageData` function. We can get all or a subset of the canvas pixels. The returned 1D array of pixels is ordered left to right, followed by top to bottom.

```
var imageData = context.getImageData(0,0,canvaswidth,canvasheight);
```

Changing pixels

Once we have access to the pixel array, we can modify its values. The following example converts the colored pixels into a grayscale representation.

```
var pixels = imageData.data;
for (var i = 0, n = pixels.length; i < n; i += 4) {
    //gray = 30% red + 59% green + 11% blue
    var gray = (pixels[i]*0.3) + (pixels[i+1]*0.59) + (pixels[i+2]*0.11);
    pixels[i ] = gray;
    pixels[i+1] = gray;
    pixels[i+2] = gray;
}
```

Note that each pixel actually takes four spots in the pixel array (one for each of Red, Green, Blue, and Alpha values), which is why we increment our array iterator by 4.

Updating the canvas

After modifying the pixels, we need to update the image displayed by the canvas. We do so by replacing the current value of the pixels by the modified value:

```
context.putImageData(imageData, 0, 0);
```

As before, we can position the new pixels anywhere within the canvas, and in the example

above, we have positioned it at the canvas origin.

A common image editing operation performed using the Canvas API is the implementation of image filters. See the following link for a demo. of some Canvas API image filters, including some convolution (sharpen, Laplace, etc.) filters:

<http://www.html5rocks.com/en/tutorials/canvas/imagefilters/>

If your image editing application is limited to filters where you will perform the *same* operation for every pixel (e.g. you wish to perform a fixed stencil convolution over the image), we recommend you limit the use of HTML5 Canvas only for prototyping, and use WebGL or CSS filters (described later) for the actual release code. The latter option will prevent the expensive Javascript loop over all the pixels, and can instead be executed in parallel as GPU shaders.

Animation

Remember that the HTML5 canvas operates in “immediate” mode, and does not save any information about the objects drawn on it. Given this immediate mode, animation is performed by modifying the position of the animated object and simply redrawing the region of the canvas that has changed. In most cases, the modified region includes the entire canvas.

<http://bomomo.com/>

<http://www.blobsallad.se/>

Performance Improvement

Traditionally, the code for animation using Javascript utilizes a timer, where some code for updating the position of elements in the browser is invoked at regular intervals (using the “setTimeout(<callback>, <timeout_interval>)” function call). The problem with that approach is that complex animations can slow down the browser and produce an undesirable user experience. The new, preferred method is to use the new `requestAnimationFrame` function that indicates that we wish to animate the contents of the browser window at 60Hz (ideally) or as fast as possible for the browser (if not 60Hz). This allows the browser to optimize code for us, and also prevents the host computer from being unnecessarily slowed down due to our animation.

Therefore, we can use the following code structure (originally from <http://paulirish.com/2011/requestanimationframe-for-smart-animating/>):

```
//define the correct requestAnimationFrame function first
window.requestAnimationFrame = window.webkitRequestAnimationFrame ||
window.mozRequestAnimationFrame || window.msRequestAnimationFrame;

//invoke the requestAnimationFrame function in the render callback
function animateCanvas(time) {
  //specify that we wish to update the canvas at 60Hz
  window.requestAnimationFrame(animateCanvas, canvas);

  //display the entire canvas (includes both the changed and unchanged
  items)
  drawCanvas();
}
```

The following link describes the use of `requestAnimationFrame` and other hints for improving

performance on HTML5 Canvas:

<http://www.html5rocks.com/en/tutorials/canvas/performance/>

Comparison with SVG

Some readers will have observed the similarity between the path drawing and filling capabilities of the HTML5 Canvas and those of the Scalable Vector Graphics (SVG) specification commonly implemented on modern web browsers. In many aspects, the HTML5 Canvas is similar to SVG. In this sub-section, we point out the differences between the two, especially those that are relevant for graphics programmers.

The main difference between the HTML5 Canvas and SVG is their rendering mode: HTML5 Canvas uses immediate mode, while SVG uses declarative mode. By using declarative mode, we can access individual SVG elements through the DOM and modify their properties (such as position, color, visibility) without needing to re-draw the drawing area. The re-draw is handled by the browser. In this sense, an SVG element is similar to an HTML element --- we can change the individual element properties dynamically and the web page is re-drawn automatically.

The declarative mode of SVG also includes another feature: grouping. Vector artwork can be combined into one groups, several of which can be further combined into another group, and so on. Such a hierarchical organization or artwork in semantically relevant groups allows us to specify regions of influence of local transformations (useful for adding details to existing, animated vector artwork, for example).

Another difference between SVG and HTML5 Canvas is the use of filter effects in SVG. While SVG does not offer direct byte-level access to pixels like HTML5 Canvas does, a fixed set of image filters can be applied directly to the SVG elements, without needing to write them in (slower) Javascript. See this site for an example:

http://ie.microsoft.com/testdrive/Graphics/hands-on-css3/hands-on_svg-filter-effects.htm

We shall explore SVG filter effects in more detail in the next section on CSS3, when we describe CSS shaders.

CSS

Remember that after building the DOM tree, the browser performs a layout process. For the purpose of this course, think of the layout as the placement of rectangular regions (one for each HTML block element) within the 2D space of the browser window. As mentioned before, the rules used by the browser for displaying each visual element (e.g. the background color of the rectangular region) are provided by the CSS style associated with that element.

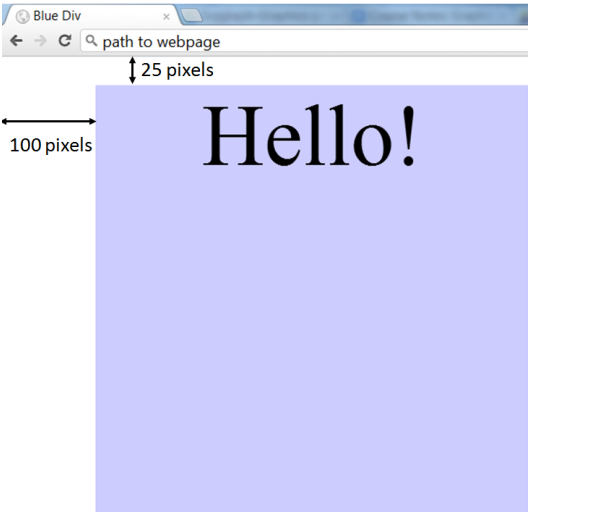
In this section, we describe some of the new features in CSS that are relevant for graphics programmers. These new features significantly improve the ability to dynamically change the display styles of HTML elements, most notably the position and orientation of those elements. CSS allows us to easily produce animations without needing any Javascript code to update positions programmatically. Since this transfers the animation functionality from non-native (Javascript) code to native and carefully optimized (browser) byte-code, it usually produces significant performance improvements.

Similar to the HTML5 Canvas section, we introduce the graphics capabilities of CSS by showing how CSS can be used to implement functionality commonly needed by graphics applications. We shall consider two applications: planes in space, and image filters.


Planes in Space

The term “planes in space” refers to the ability to place 2D planar polygons at any position and orientation in 3D space. Previously we could place any HTML block element at any position within the 2D window. Typically this was done by changing the “left” and “top” style attributes for that element.

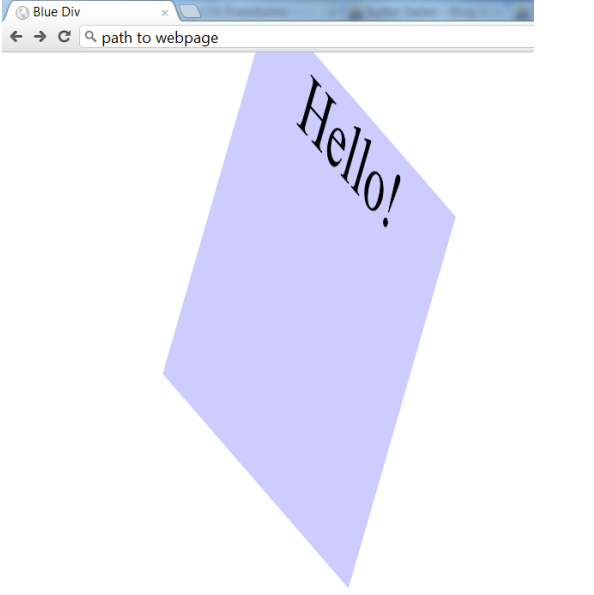
Consider the task of placing a <div> element at a horizontal distance 100 pixels and a vertical distance of 25 pixels from the top-left corner of the browser window. We use the following code:

<pre><html> <body> <div style="width: 500px; height: 500px; position: relative; left:100; top: 25; background-color:#CCCCFF"> Hello! </div> </body> </html></pre>	
---	--

We can add a 3D effect to the 2D element by making it rotate by 60 degrees about the vertical (Y) axis. If we use a browser using the Webkit layout engine (e.g. Google Chrome, Apple Safari), we use the following code:

<pre><html> <body> <div style="width: 500px; height: 500px; position: relative; left:100; top: 25; -webkit-transform: rotateY(60deg); background-color:#CCCCCCFF"> Hello! </div> </body> </html></pre>	
--	--

The transforms can be concatenated:

<pre><html> <body> <div style="width: 500px; height: 500px; position: relative; left:100; top: 25; -webkit-transform: rotateY(60deg) rotateZ(30deg); background-color:#CCCCCCFF"> Hello! </div> </body> </html></pre>	
---	---

We can specify most common types of 3D transformations (rotate, translate, scale, skew, and perspective) on any block element. We can even specify directly the 4x4 transformation matrix to be applied to the element. See <http://dev.w3.org/csswg/css3-transforms/> for the API currently proposed for CSS transforms. Also see <http://desandro.github.com/3dtransforms/> for an excellent explanation of the CSS transforms in depth.

The benefit of applying 3D transforms to a block element is that after transforming the

element, the browser continues to interact with contents of the element as before. Text in the element remains selectable, links still work, and videos or images are displayed with correct transformations. This makes the CSS3 transforms very useful for creating 3D interactive user interfaces. The links below give some examples:

A simple image flip on mouse over:

<http://www.webkit.org/blog-files/3d-transforms/image-flip.html>

More advanced image effects using CSS:

<https://developer.mozilla.org/en-US/demos/detail/3d-image-transitions/launch>

Placing webpage elements in 3D space:

<http://www.webkit.org/blog-files/3d-transforms/morphing-cubes.html>

Note: You may be wondering why we needed to use the prefix “-webkit-” for style attributes like transform. The newer CSS styles are not yet finalized by the W3C. In order to allow browsers to support the non-finalized feature or an incomplete implementation of the feature, individual browsers support the prefixed forms of the style attribute. Browsers with the Webkit layout engine (e.g. Google Chrome, Apple Safari) will respect the transform attribute with the “-webkit-” prefix. Similarly, Mozilla-based Firefox needs the “-moz-” prefix, Internet Explorer the “-ms-” prefix, and Opera the “-o-” prefix. This is a temporary solution until the CSS specification is finalized by the W3C and universally adopted by all browsers. Until then, you need to specify all the prefixes for the new style attributes so your code may run on all browsers. You can use tools like “Prefix free” (<http://leaverou.github.com/prefixfree/>) to produce the prefixed versions of the CSS attributes automatically at script runtime.

Animations

In the demos above, elements placed by CSS are animated. We can control the manner in which they move from one position to the other. A simple method to bring about animation is via CSS transitions. For example, our example above for rotating the <div> element can be animated by applying a transition on the transform style attribute. The transform style attribute itself is modified when the user clicks on the element.

```
<html>
<body>
<div style="width: 500px; height: 500px; position: relative; left:100; top: 25;
background-color:#CCCCCCF;
-webkit-transition: -webkit-transform 3s ease-in;"
onclick="this.style.webkitTransform='rotateY(60deg) rotateZ(30deg)'">
Hello!
</div>
</body>
</html>
```

In the above example, we set the style of the “-webkit-transition” to be “-webkit-transform 3s ease-in”. As you can guess, this means that we are adding a transition to the -webkit-transform attribute that is 3 seconds long, and we ease-in from the old value to the new one. We can specify transitions on all/none/some of the style attributes of the element, for any duration, and with different timing functions (ease-in, ease-out, linear, etc.). The complete specification for the transitions is provided here: <http://www.w3.org/TR/css3-transitions/>.

If animations using transitions are not sufficient, we can also create explicit animation of the style attributes using keyframes.

```
<html>
<head>
<style type="text/css">
@-webkit-keyframes wobble {
    0% {
        -webkit-transform: scale3d(1,1,1) rotateZ(0deg);
    }
    33% {
        -webkit-transform: scale3d(1.2,1.2,1) rotateZ(30deg);
    }
    66% {
        -webkit-transform: scale3d(1.2,1.2,1) rotateZ(-30deg);
    }
    100% {
        -webkit-transform: scale3d(1,1,1) rotateZ(-deg);
    }
}</style>
</head>
<body>
<div style="width: 50px; height: 50px; position: relative; left:100; top: 25;
background-color:#CCCCFF;
-webkit-animation-name: wobble;
-webkit-animation-duration: 3s;
-webkit-animation-iteration-count: infinite;
-webkit-animation-direction: alternate;
-webkit-animation-timing-function: linear;">
Hello!
</div>
</body>
</html>
```

In the above example, we first create some keyframes (at 0%, 33%, 66% and 100% of the animation duration). At each keyframe, we specify values of the style attributes that need to change. The animation keyframes are then specified as the value of the animation-name attribute, along with the duration of each cycle, the number of complete cycles (infinite for continuous looping animation), whether to step forwards, backwards or forward and backward, and the timing function (ease-in, ease-out, linear, etc.). The complete specifications for CSS animations is available here: <http://www.w3.org/TR/css3-animations/>.

A final note on the “planes in space” functionality is that soon browsers will be able to support *curved* planes in space for placing block tags. Curved planes will be possible through the use of CSS shaders, which are described in the next section.

More resources

General-purpose CSS animation: <http://tinyurl.com/csswalk>

Traditional animation principles implemented as CSS:

<http://coding.smashingmagazine.com/2011/09/14/the-guide-to-css-animation-principles-and-examples/>

Image Filters

Similar to the HTML5 Canvas, we can implement image filters using CSS. The image filters in CSS (called “filter effects”) are essentially the same as those for SVG: a fixed set of image filters applied to every pixel of the image. The proposed specification for CSS filter effects is <https://dvcs.w3.org/hg/FXTF/raw-file/tip/filters/index.html>.

Like SVG, CSS filter effects support some hard-coded filter functions (e.g. “brighten”, “sepia”, “grayscale”) that accept a few user parameters. The syntax is as simple as specifying the `filter` attribute, like:

```
-webkit-filter: blur(2px) grayscale(1);
```

Note that multiple filter functions can be specified, and the order of the filter functions is important.

The following link shows all the filters in action: <http://html5-demos.appspot.com/static/css/filters/index.html>

Similar to SVG filters, the benefit of using filter effects through CSS instead of HTML5 Canvas is improved performance: we do not need to iterate over every pixel in the image using Javascript, and can exploit the parallel computation ability of modern GPUs. For this reason, if you wish to use one of the filter effects in the specifications and CSS filter effects are supported by your target browser, we recommend you use CSS filter effects instead of Javascript-coded HTML5 Canvas image manipulation.

General-Purpose CSS Shaders

The exciting development of CSS filter effects is that in addition to the fixed filters, the CSS filter effects specification allows us to specify a *custom* shader as a filter. This is exciting because it has the potential to open up the massive parallel computation available on most GPUs for general-purpose computing via simple shader programs. These general-purpose programs need not have anything to do with graphics, and the CSS shaders can be used for large-scale parallel computation. Since this is a rather new specification, browser implementations that support general purpose CSS shaders are not yet available (at the time of this writing). This article gives a more in-depth description of CSS shaders: <http://www.adobe.com/devnet/html5/articles/css-shaders.html>

As the above link illustrates, with CSS shaders we can apply vertex shaders to the grid of polygons that is overlaid on the HTML element. The contents of the HTML element are rendered as a texture map (filtered appropriately) onto grid. Through these shaders, we can specify a filter that changes the position of the grid vertices, thereby creating *curved* planes in space, as was mentioned before.

WebGL

Zhenyao Mo, Kenneth Russell
Google, Inc.

[Introduction](#)
[Stream Processing](#)
[Vertex and Fragment Shaders](#)
[A Concrete Example](#)
 [Setting up WebGL](#)
 [Loading Shaders](#)
 [Loading Programs](#)
 [Setting up Geometry](#)
 [Drawing the Scene](#)
[Higher-Level Libraries](#)
[Achieving High Performance](#)
 [Picking in Google Body](#)
 [Particle Systems](#)
 [Sprite Engines](#)
 [Physical Simulation](#)
[Conclusion](#)

Introduction

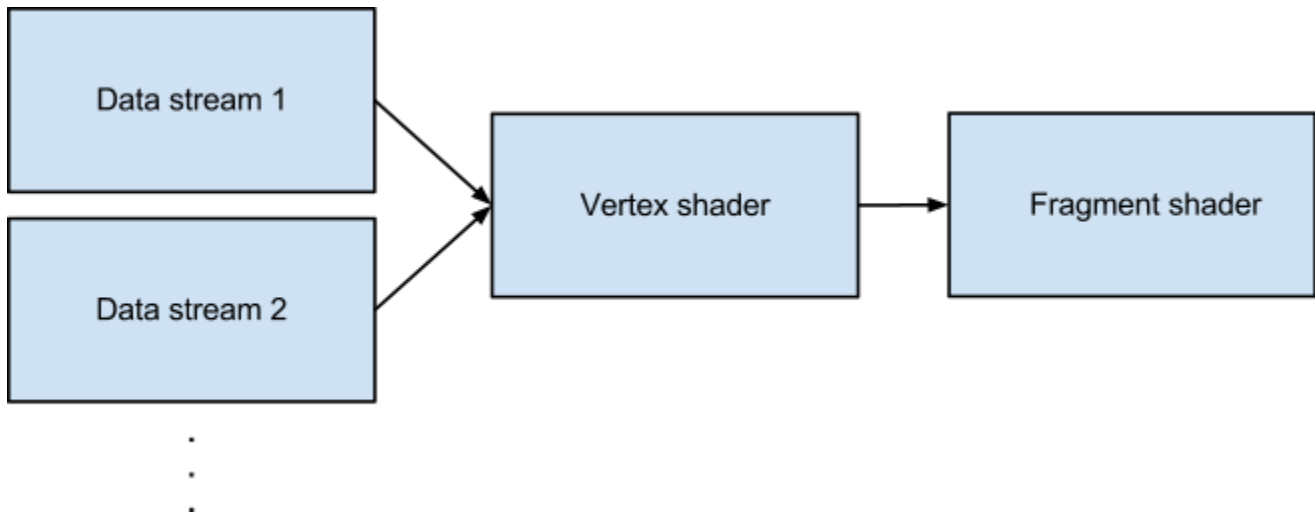
WebGL brings 3D graphics to the HTML5 platform. WebGL, in short, is an alternative rendering context for the HTML5 Canvas element, which provides the OpenGL ES 2.0 API to JavaScript.

OpenGL, and OpenGL ES, supply a proven rendering model, but one which differs from other graphics APIs on the Web. In OpenGL, OpenGL ES and WebGL, the triangle is the basic drawing primitive. Data for many triangles is prepared once, but drawn many times. The data uploaded to the graphics processing unit (GPU) includes concepts like vertex positions, colors, textures, and more. Shaders -- small programs that execute directly on the GPU -- determine the position of each triangle and the color of every pixel on the screen.

The WebGL, OpenGL ES and OpenGL APIs are developed by the Khronos Group, a non-profit industry consortium creating open standards for the authoring and acceleration of parallel computing, graphics, dynamic media, computer vision and sensor processing on a wide variety of platforms and devices.

Stream Processing

WebGL exposes a stream processing model. Each point in 3D space has one or more streams of data associated with it: for example, position, surface normal, color, or texture coordinate. In OpenGL (and WebGL), these streams are called *vertex attributes*. These streams of data flow through the vertex and fragment shaders.

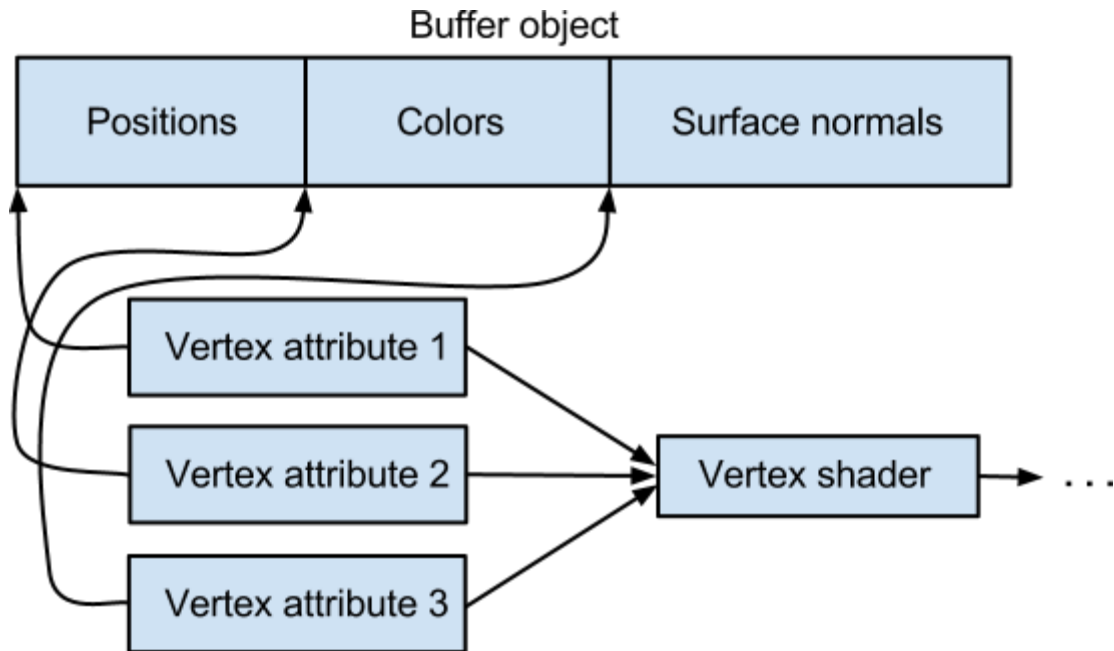


Vertex and Fragment Shaders

Vertex and fragment shaders are 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. Its primary goal is to output the location where the vertex should appear in the on-screen window. The vertex shader may also output one or more additional values -- called *varying variables* -- to the fragment shader.

For each triangle, the GPU figures out which pixels on the screen are covered by the triangle. The GPU then runs the fragment shader on each of those pixels. At each pixel, the GPU automatically blends the outputs of the vertex shader based on where the pixel lies within the triangle. The fragment shader then determines the color of the pixel based on those inputs.

Vertex data is uploaded into one or more *buffer objects* which reside on the GPU. The vertex attributes in the vertex shader are *bound* to the data in these buffer objects.



A Concrete Example

Now we'll go through a concrete example, adapted from Giles Thomas' [Learning WebGL Lesson 2](#). The code is checked in to the [webgl samples project](#) and can be [viewed directly](#) in a WebGL-enabled browser. The goal of this example is to de-mystify WebGL by showing all of the steps necessary to draw a colored triangle on the screen.



The vertex shader for this example is very simple:


```
attribute vec3 positionAttr;
attribute vec4 colorAttr;

varying vec4 vColor;

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

In this example, the vertex shader will be executed a total of three times, because we are only drawing one triangle containing three vertices. In a typical application, thousands or tens of thousands of triangles are typically drawn together.

The fragment shader for this example is also very simple:

```
precision mediump float;

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

The value of the `vColor` *varying variable* comes from a weighted combination of the colors specified at the three input vertices. Based on the location of the pixel within the triangle, the GPU automatically blends the color that was specified at each vertex.

The fragment shader will be executed between a dozen to tens of thousands of times, depending on the size of the canvas. Recall that the fragment shader executes at each pixel on the screen covered by a given triangle.

The shader text for this example is embedded directly in the web page using script elements. It is entirely up to the application how to manage the sources for its shaders. Script tags were chosen to hold the shaders for this example for simplicity. A real world application might download the shaders from the server using XMLHttpRequest, generate the source code for its shaders using JavaScript code, or one of many other options.

```
<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>
```

Setting up WebGL

Now we'll look at how the example initializes WebGL as it begins to run.

```
var gl = null;
var contextNames = [ "webgl", "experimental-webgl" ];
for (var ii = 0; ii < contextNames.length; ++ii) {
  try {
    gl = canvas.getContext(contextNames[ii]);
    if (gl)
      break;
  } catch (e) {
  }
}
if (!gl) {
  alert("Could not initialise WebGL, sorry :-(");
}
```

This isn't the best error detection logic; it's deliberately kept short in order to keep the example simple. Consult other examples in the `webglsamples` repository, such as the WebGL Aquarium, for better references when building real applications. You should link to <http://get.webgl.org/> if your application's initialization fails.

(Note the need to check for both the `webgl` and `experimental-webgl` context types. This is a temporary situation while the WebGL specification is reaching a certain degree of conformance across web browsers and operating systems. In the near future it is expected that all web browsers supporting WebGL will begin to advertise the `webgl` context type, and the `experimental-webgl` context type will no longer have to be queried by applications.)

Loading Shaders

The next step in a WebGL application is to load its shaders. A specific sequence of steps is followed to do this:

- Create the shader object – vertex or fragment.
- Specify its source code.
- Compile it.
- Check whether compilation succeeded.

The complete source code for these steps follows. Some error checking is elided for brevity.

```
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;
}
```

Note that the WebGL API calls (`shaderSource`, `compileShader`, etc.) match very closely the written description of what needs to occur during loading of the shader.

Loading Programs

The next step is to construct a *program object*. A program object combines the vertex and fragment shaders, and completely specifies how a piece of geometry is rendered. Again, a specific sequence of steps is followed to load the program:

- Load each shader separately.
- Attach each to the program.
- Link the program.
- Check whether linking succeeded.
- Prepare vertex attributes for later assignment.

The complete source code for these steps follows.

```
var program;
function initShaders() {
    program = gl.createProgram();
    var vertexShader = getShader(gl, "shader-vs");
    gl.attachShader(program, vertexShader);
    var fragmentShader = getShader(gl, "shader-fs");
    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

Now we need to set up the geometry that will be drawn to the screen. This step is independent of the initialization of the shaders and program object; it could just as well be done before the program was loaded. It is a two-step process; first a buffer object is allocated on the GPU, and then the geometric data containing all of the vertex streams is uploaded. There are many choices to make when deciding how to organize the geometry, including whether to use interleaved or non-interleaved vertex data, using one or more buffer objects, where to place vertex data within the buffer object, etc. As experience with the OpenGL and WebGL API is gained, many of these decisions become more easily apparent. In general, it is strongly desirable to use as few buffer objects as possible. Switching between them is expensive.

The complete code for the setup of this example's geometry follows.

```
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);
}
```

This is, again, a very simple example. Most real applications would store complex models on the server and transmit them using an efficient representation, such as that supplied by the [WebGL Loader](#) project.

Drawing the Scene

At this point, all of the pieces are in place in order to draw the scene. The steps to do so are very simple:

- Clear the viewing area.
- Set up vertex attribute streams.
- Issue the draw call.

The complete code follows.

```
function drawScene() {
  gl.viewport(0, 0, canvas.width, canvas.height);
  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);
}
```

That's it! The triangle now appears on the screen.

Higher-Level Libraries

Now that we've dragged you through a complete example, many libraries already exist to make it easier to use WebGL. One list (not comprehensive) is maintained on the WebGL wiki at http://www.khronos.org/webgl/wiki/User_Contributions#Frameworks. A few suggestions for frameworks to look at:

- [Three.js](#) (used in the [Rome](#) demo, [mr. doob's demos](#), and many others)
- [CubicVR](#) (used in Mozilla's WebGL demos such as [No Comply](#))
- [TDL](#) (used in the WebGL Aquarium and most of the other webglsamples demos)
- [CopperLicht](#) (same developer as Irrlicht)
- [PhiloGL](#) (focus on data visualization)
- [GLGE](#) (used for early prototypes of Google Body)
- [SceneJS](#) (unique and interesting declarative syntax)
- [SpiderGL](#) (lots of interesting visual effects)

Achieving High Performance

There are a few "big rules" associated with writing OpenGL programs. First and foremost is to *reduce the number of draw calls per frame*. OpenGL's efficiency compared to many other graphics APIs comes from its ability to send large amounts of geometry to the GPU with very little overhead. Sending down small batches, or even worse, one or two triangles per draw call,

does not give the GPU the opportunity to optimize the handling of many triangles at once.

In order to draw many triangles at once, it is typically necessary to sort the objects in the scene by their rendering state: for example, objects using the same texture should be drawn with consecutive draw calls, rather than drawing one object with texture 1, another object with texture 2, and a third object with texture 1.

Objects should be sorted and drawn according to the following criteria, in decreasing order of importance:

- Target framebuffer or context state
 - Blending, clipping, depth test, etc.
- Program, buffer, or texture
 - Switching these often requires a pipeline flush
- Uniforms and samplers
 - Switching these is relatively cheap, modulo JavaScript overhead

(Thanks to Ben Vanik at Google for this information.)

Wherever possible, sort the objects in the scene ahead of time, and maintain the objects as a sorted list for rendering purposes. Walking the object hierarchy and performing a sort of objects per frame can cancel the gains from batching draw calls. Generate your content (3D models, etc.) so that it can be easily batched; merge buffers, textures, etc.

In WebGL, all of the OpenGL “big rules” apply, along with another one: offload as much JavaScript to the GPU as possible, within reason. Often the GPU can be used to rephrase a computation that would otherwise need to be done in JavaScript, and you can achieve not only better parallelism but often better performance by doing so. The following examples show how this rule is applied in the context of a few real-world examples.

Picking in Google Body

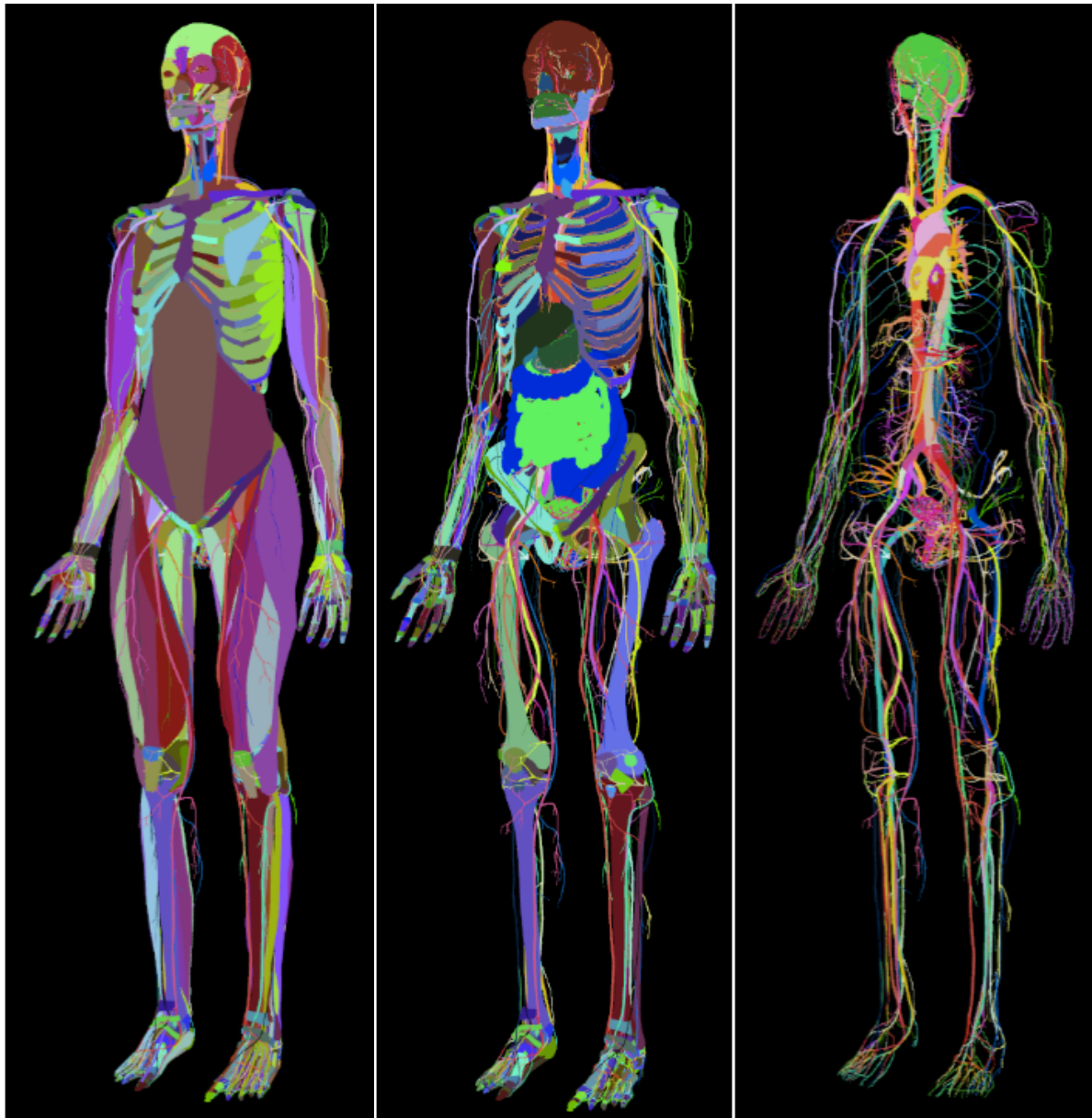
(Thanks to the Google Body team for this information.)

Google Body is a browser for the human anatomy. Originally developed at Google Labs, it is now available as [Zygote Body](#), from the company that developed the 3D models. The models in this application are highly detailed -- over a million triangles -- yet selection is very fast. One can click any body part to highlight it and see its name.

How can picking be implemented? One could consider doing ray-casting in JavaScript. When the mouse is clicked, set up a ray starting at the eye point, going through the near plane of the “camera”, and intersect that ray with all of the triangles in the scene. One could attempt to do quick discards if the ray doesn’t intersect an object’s bounding box, to avoid ray-triangle tests for objects that are obviously missed by the ray. Regardless, this is still a lot of math to do in JavaScript.

Instead, Google Body uses the GPU to implement picking. When the model is loaded, each organ is assigned a different color. When the mouse is clicked, the model is rendered offscreen with a different shader than usual. This shader draws each shape with its preassigned color,

without any texturing, lighting, or transparency. A threshold is used to determine whether to draw translucent layers during this process; layers which are “too transparent” to pick are simply not drawn. After this render pass is completed, the color under the mouse pixel is read back to the CPU using `readPixels`. The same technique works at different levels of granularity; for example, each triangle, rather than each object, could be assigned a different color to achieve finer detail when picking. The following three screenshots show the picking process while different layers of the model are visible.



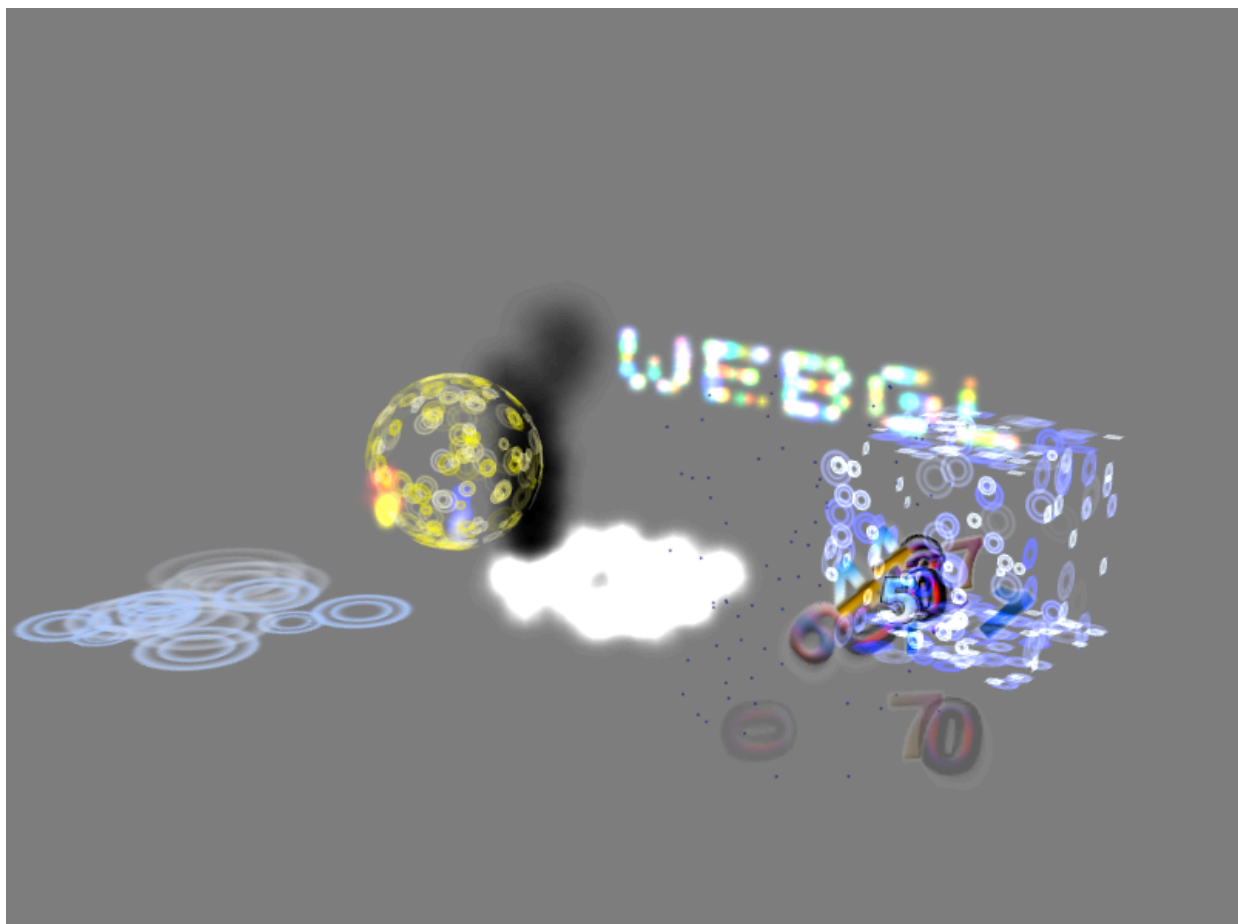
Note that this technique uses the GPU for what it is best at -- rendering. It essentially converts the problem of picking into one of rendering. Despite the need to perform a readback from the GPU to the CPU at the end of the algorithm, the performance gains from using the GPU are worth it.

Particle Systems

Particle systems are a technique commonly used to draw graphical effects like explosions, smoke, clouds, and dust.

The most obvious way to implement a particle system is to compute the positions of the particles on the CPU, upload the vertices to the GPU, and draw them in a single draw call. This technique can work for small particle systems, but does not scale well because many vertices are uploaded to the GPU each frame. Additionally, mathematical operations are not yet as fast in JavaScript as they are in C or C++. A different technique is desired.

Gregg Tavares has developed a [particle system demonstration](#) in the [WebGL demo repository](#) which animates roughly 2000 particles at 60 frames per second. All of the animation math is performed on the GPU. Each particle's motion is defined by an equation: initial position, velocity, acceleration, spin, and lifetime. When the particle is created, these parameters are set up once and never modified afterward. Each frame, only one parameter needs to be sent from the CPU to the GPU: the current time. The vertex shader evaluates the equation of motion and moves the particle into its desired location for the current frame. This technique ensures that the absolute minimum amount of JavaScript work is done per frame.



Nihilogic's [Worlds of WebGL](#) demonstration shows a similar particle system technique. In this demonstration, particles assemble to form various shapes, falling to the floor between scenes and animating smoothly between them. The animation is done similarly to Gregg Tavares' particle system above. For each scene, random positions for the particles are chosen at setup time. The time parameter interpolates between two vertex attribute streams at any given time: one stream contains the particle positions on the floor, and the other the particle positions for the current shape. Once the current shape has been assembled, the next interpolation target becomes the particles on the floor again. JavaScript does almost no computation.

Sprite Engines

In roughly early 2011, Facebook released a sprite engine benchmark called [JSGameBench](#) comparing various techniques for rendering animated sprites within a web browser, including moving around DOM elements and drawing with both 2D Canvas and WebGL. Sprites are generally similar to particle systems, but are terminology more commonly used when authoring certain kinds of 2D games. JSGameBench doesn't appear to be under active development any more, but some lessons can be learned about its structure and performance characteristics.

At the time JSGameBench was released, its WebGL backend performed one draw call per sprite. It seemed that drawing the entire sprite field with one draw call would be an obvious performance win, so a prototype sprite engine library was developed to test this hypothesis.

The first question was whether it was even possible to draw all of the sprites at once; they use alpha blending, so the order they are drawn is important. It's a little known fact that OpenGL actually guarantees the order the triangles are drawn in when `glDrawArrays` and `glDrawElements` are called. Apparently GPUs contain quite a bit of silicon -- the Render Output unit, or ROP -- to provide this guarantee; thanks to Nat Duca at Google for this information. Therefore it was technically possible to draw the entire sprite field at once, since the order the sprites were drawn to the screen would be consistent from frame to frame.

The next question was how to merge together the sprites' multiple images. One obvious way would be to put all of the sprites' images into a single texture, so that the draw call would reference only one texture. This seemed like an inflexible solution, since it would be bounded by the maximum size of an individual texture. Instead, the desired solution was to pass down multiple textures to the fragment shader, and have the shader choose which one to sample for the sprite's image.

Conceptually, we would like to send down a uniform array of samplers, e.g., `uniform sampler2D textures[4]`, and compute an index into this array. Unfortunately, the WebGL shading language, which is essentially the same as the OpenGL ES shading language, doesn't allow this kind of indexing expression in a fragment shader. The only kind of indexing expression allowed is one involving constants and loop indices.

The first attempt at the texture selection fragment shader looked like this. It passed down a vector of coefficients for each vertex, which selected one of the four incoming textures.

```
gl_FragColor =
    (texture2D(u_texture0, v_texCoord) * v_textureWeights.x +
     texture2D(u_texture1, v_texCoord) * v_textureWeights.y +
```

```
texture2D(u_texture2, v_texCoord) * v_textureWeights.z +  
texture2D(u_texture3, v_texCoord) * v_textureWeights.w);
```

This worked, but unfortunately, the resulting demo was slower than the existing WebGL backend of JSGameBench -- about 66% of the performance. Experiments were done to remove the "explosion" sprite, which is the largest of all the sprites (256x256, filling a 2048x2048 texture), and selecting the sprites from the remaining three sheets. This yielded a significant speedup, which strongly indicated that the texture bandwidth on the card was being exhausted.

Nat Duca suggested to use a series of if-tests in the fragment shader to sample the desired texture. Previous experience had been to avoid if-tests in shaders at all costs; in earlier work, every time an if-test in a shader had been replaced with a non-branching operation like a clamp or step, performance had improved. Nat indicated that on modern cards, if the branch will go the same way over large regions (which it will in this case; it's constant across the entire surface of the sprite), it will work well. The fragment shader was rewritten as follows:

```
vec4 color;  
if (v_textureWeights.x > 0.0)  
    color = texture2D(u_texture0, v_texCoord);  
else if (v_textureWeights.y > 0.0)  
    color = texture2D(u_texture1, v_texCoord);  
else if (v_textureWeights.z > 0.0)  
    color = texture2D(u_texture2, v_texCoord);  
else // v_textureWeights.w > 0.0  
    color = texture2D(u_texture3, v_texCoord);  
gl_FragColor = color;
```

This technique worked well; on the development machine, it rendered 250% or more sprites at the same frame rate than the WebGL backend of JSGameBench at the time. JSGameBench was subsequently updated to use similar batching techniques.

The source code for, and more details on, this sprite engine prototype are available in the [WebGL samples](#) project in the `sprites` subdirectory; see the [documentation](#) and [live demo](#).

Physical Simulation

WebGL supports floating-point textures as an extension, meaning that every texel can store one or more floating-point values. The fact that the GPU can operate on so much floating-point data at once means that it is possible to perform advanced techniques in WebGL such as physical simulation. Any iterative computation where each step relies only on nearby neighbors is a good candidate for moving to the GPU.

Evgeny Demidov has developed several [demonstrations](#) showing how to simulate waves, interference patterns, 2D fluid dynamics and other techniques in WebGL.

Evan Wallace has developed [demonstrations](#) utilizing floating-point textures to [simulate interactive water in a pool](#) and even do [ray tracing](#) in WebGL.

Conclusion

WebGL is an evolving specification and ecosystem. We look forward to your participation in the community!

- [WebGL landing page at the Khronos Group](#)
- [WebGL wiki](#)
- [WebGL specification](#) (editor's draft)
- [WebGL developers' mailing list](#) (for discussing the use of WebGL)
- [WebGL public mailing list](#) (for discussing the specification)



Graphics Programming on the Web

WebCL Course Notes

Siggraph 2012

Mikaël Bourges-Sévenier¹
Motorola Mobility, Inc.

Abstract

This document introduces WebCL [1], a new standard under development by the Khronos Group, for high-performance computing in web browsers. Since WebCL wraps OpenCL, the course starts by reviewing important OpenCL [2] concepts. Next, we detail how to program with WebCL in the browser and on devices such as GPUs. Finally, we discuss WebCL – WebGL [3] interoperability and provide complete examples of moving from WebGL shaders to WebCL. Last, we provide tips and tricks to ease such translation and to optimize WebCL code performance.

¹mikeseven@acm.org



Table of Content

1	What is WebCL?	3
2	Glossary and conventions	3
3	Thinking in parallel	4
4	OpenCL memory model	4
5	Programming with WebCL	7
5.1	Host/Browser side	7
5.1.1	Platform layer	8
5.1.2	Runtime layer	9
5.2	Device side	15
6	Interoperability with WebGL	16
6.1	Fun with 2 triangles	17
6.1.1	General CL-GL interoperability algorithm	18
6.1.2	Using shared textures	18
6.1.3	Using shared buffers	19
6.1.4	Example	20
6.2	Other applications	22
7	Tips and tricks	22
7.1	From GLSL to OpenCL C	23
7.2	Barriers	23
7.3	Local work-group size	23
7.4	Learn parallel patterns!	23
8	WebCL implementations	23
9	Perspectives	24
Appendix A	CL-GL code	25
A.1	Graphics object	25
A.2	Compute object	27
A.3	Mandelbulb kernel (direct conversion)	31
A.4	Mandelbulb kernel (optimized)	34
Appendix B	OpenCL and CUDA terminology	37
Bibliography		39
	Specifications	39
	Programming guides	39
	Books	39
	WebCL prototypes	39
	Articles and Presentations	39



1 What is WebCL?

In short, WebCL is to OpenCL what WebGL is to OpenGL. WebCL is a JavaScript API over OpenCL API; Khronos Group is defining all these international standards. Historically, OpenGL was defined as a standard for hardware accelerated graphics, hence Graphics Language. OpenGL was first a fixed pipeline a programmer could change various states to produce images. Then, OpenGL pipeline became programmable using shaders, pieces of C like code that can be inserted at some points of the OpenGL rendering pipeline.

As the need for more complex applications arise, programmers realized that shaders could be used for more general programming problems, taking advantage of the massively parallel nature of GPUs; this became known as GPGPU. But shaders can only provide limited features for such applications.

Few years ago, Apple proposed OpenCL to the Khronos Group, a more general framework for computing, hence the term Compute Language. Not only OpenCL allows usage of GPUs but also any devices that has a driver in the machine: CPUs, DSPs, accelerators, and so on.

It is important to note that OpenCL doesn't provide any rendering capability, unlike OpenGL; it only processes data, lots of data. The source of such data could be OpenGL buffers such as vertex buffers, pixel buffers, render buffers, and so on.

To understand WebCL, it is necessary to understand the OpenCL programming model.

2 Glossary and conventions

Work-item	The basic unit of work of an OpenCL device
Work-group	Work-items execute together as a work-group
Kernel	The code of a work-item, a C99 function
Program	A collection of kernels and other functions, same as a dynamic library
Context	The environment within which work-items executes. This includes devices, their memories, their command queues, and so on

In this course, we will use the following conventions:

- Code is a yellow box
 - All lines are numbered
 - WebCL/OpenCL keywords and methods are in **bold red**
 - Comments are in **light green**
 - Language keywords are in **bold dark purple**
 - Strings are in **blue**

```
1  __kernel
2  void multiply(__global const float *a, // a, b, c values are in global memory
```

- The method console.log() is simplified to log().
- All WebCL calls throw exceptions (unlike WebGL that return error codes). For simplicity, we may omit try/catch in this document, but you should not!
- OpenCL qualifiers start with __ (two '_'). For example, one could use __kernel or kernel interchangeably. In this document, we always use __kernel.
- We use interchangeably CL for OpenCL and WebCL, GL for OpenGL ES 2.x and WebGL.
- OpenCL files end with extension '.cl'. On web pages, we use <script type = "x-webcl">, although both are not defined by any standard.



3 Thinking in parallel

Programming a massively parallel device is challenging and, for many developers, may require learning new programming skills. By massively parallel, we mean that many hardware-processing units run at once or, said differently, many hardware threads are running concurrently. While CPUs tend to have 2, 4, or 8 cores, GPUs can have thousands of cores. Even on mobile devices, GPUs with hundred of cores are coming. For web developers used to sequential event-based programs, with JavaScript language not providing threading support, it is a radical shift.

The following example shows the main idea:

- A traditional loop over a (large) set of data can be replaced by a data-parallel kernel
- Each work-item runs a copy of the kernel function.
- With n work-items, the computation is executed in 1 pass vs. n passes with a traditional loop

The OpenCL concepts are introduced in the next section.

```

3 // in JavaScript
4
5 function multiply(a, b, n)
6 {
7     var c=[];
8     for(var i = 0; i < n; i++)
9         c[i] = a[i] * b[i];
10
11     return c;
12 }

```

```

1 // in OpenCL
2
3 __kernel
4 void multiply(__global const float *a, // a, b, c values are in global memory
5              __global const float *b,
6              __global float *c, int n)
7 {
8     int id = get_global_id(0); // work-item globalID
9     if(id >= n) return; // make sure work-item don't read/write past array size
10    c[id] = a[id] * b[id];
11 }

```

Code 1 – Representing a JavaScript method into a WebCL kernel.

4 OpenCL memory model

Before we enter into OpenCL programming details, it is important to understand its platform model:

- A **Host** contains one or more **Compute Devices**. A Host has its own memory.
- Each Compute Device (e.g. CPU, GPU, DSP, FPGA...) is composed of one or more **compute units** (e.g. cores). Each Compute Device has its own memory.
- Each **Compute Unit** is divided in one or more **Processing Elements** (e.g. hardware threads). Each processing element has its own memory.

In general, we will refer to Host for the device onto which the WebCL program is executed (i.e. within the browser). We refer to Device for a compute device onto which an OpenCL Kernel is executed. Hence, a CPU can be both a Host and a Compute Device.

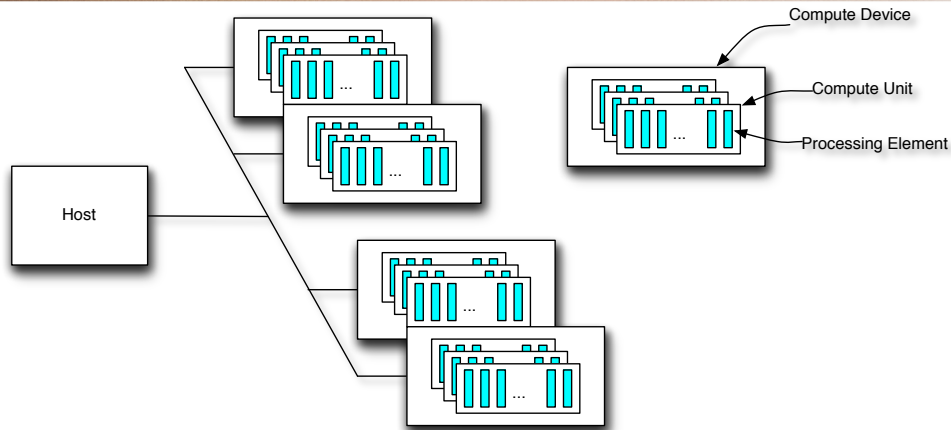


Figure 1 - OpenCL platform model

OpenCL defines 4 types of memory spaces within a Compute Device:

- **Global memory** – corresponds to the device RAM. This is where input data are stored. Available to all work groups/items. Similar to system memory over a slow bus, rather slow memory. Not cached.
- **Constant memory** – cached global memory
- **Local memory** – high-speed memory shared among work-items of a compute unit (i.e. for a work-group). Similar to L1 cache. Reasonably fast memory.
- **Private memory** – registers of a work-item. Very fast memory.

However, private memory is small and local memory is often no more than 64 KB. As a result, programmer must choose carefully which variables leave in a memory space for the best performance / memory access performance tradeoff.

Another type of memory is **Texture Memory**, which is similar to Global Memory but is cached, optimized for 2D spatial locality, and designed for streaming reads with constant latency. In other words, if your device has image support and your data can fit in texture memory, it may be better than using buffers in global memory.

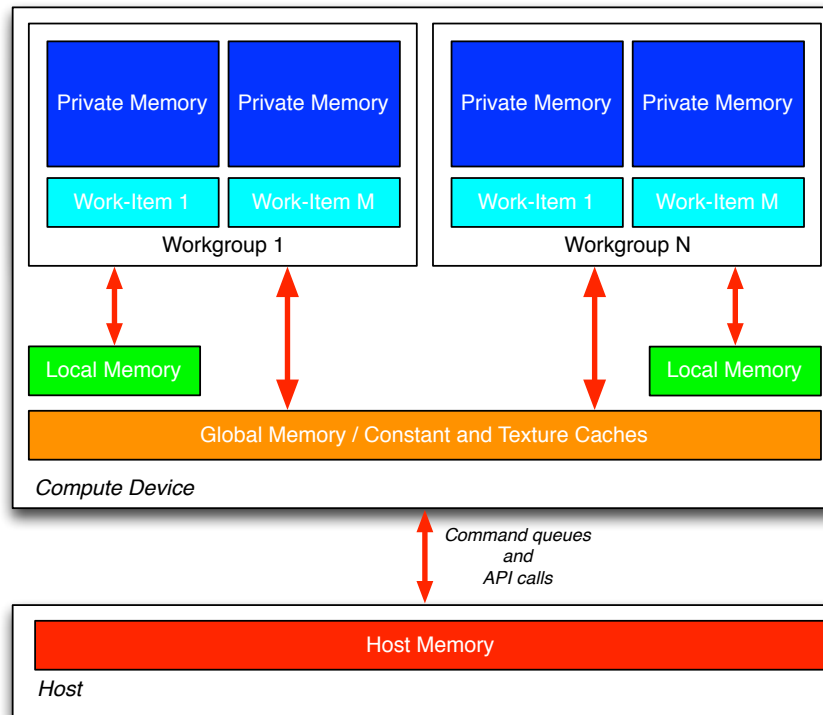


Figure 2 – OpenCL memory model

Finally, at an even lower level, work-items are scheduled as a group called *warp* (Nvidia) or *wavefront* (AMD); this is the smallest unit of parallelism on a device. Individual work-items in a warp/wavefront start together at the same program address, but they have their own address counter and register state and are therefore free to branch and execute independently [8]. Threads on a CPU are generally heavyweight entities and context switches (when the operating system swap two threads on and off execution channels) are therefore expensive. By comparison, threads on a GPU (i.e. work-items) are extremely lightweight entities. Since registers are allocated to active threads, no swapping of registers and state occurs between GPU threads. Once threads complete, its resources are de-allocated.

Each work-item has a global ID into an N-Dimensional index space, where N can be 1, 2 or 3. An N-dimensional range (or NDRange) is defined by an array of N values specifying the extent of the index space in each dimension starting at an offset F (0 by default). Within a work-group, a work-item also has a local ID.

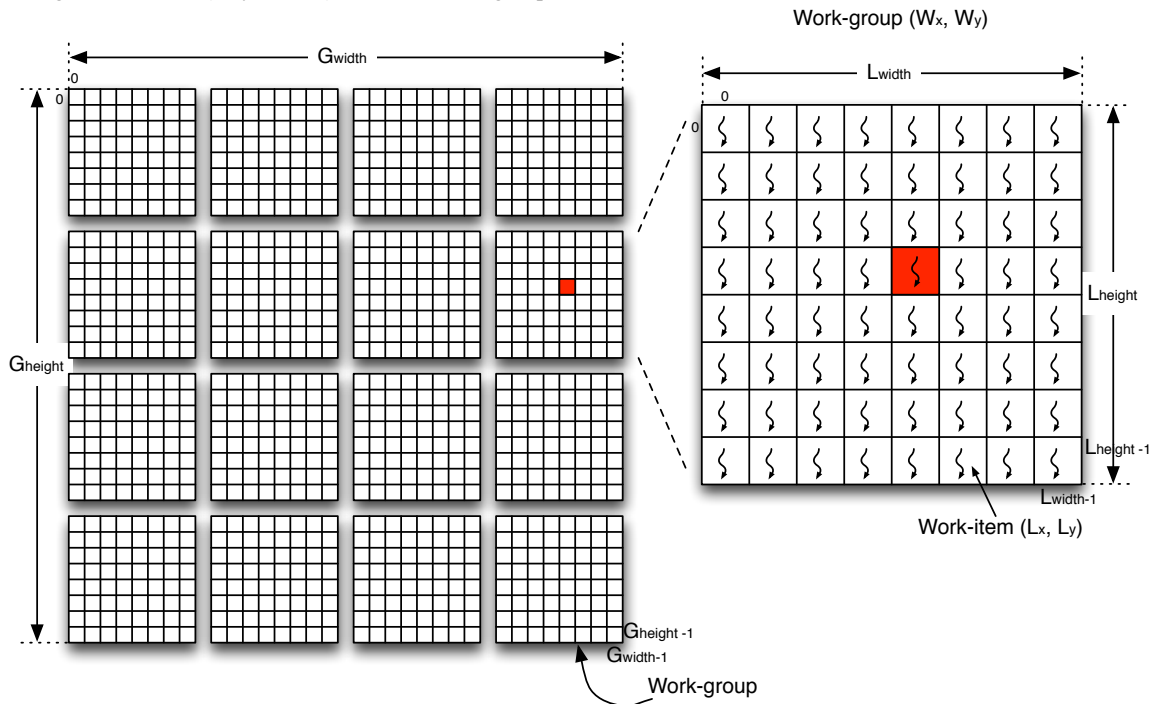


Figure 3 – Global and Local IDs for a 2D problem.

Using a 2D example, as depicted in Figure 3, with a global NDRange of size $[G_{width}, G_{height}]$ and local NDRange of size $[L_{width}, L_{height}]$,

1. Indexes always go from 0 to range-1 in each dimension
2. localID of work-item at index (l_x, l_y) is $l_x + l_y * L_{width}$
3. globalID of work-item at index (g_x, g_y) is $g_x + g_y * G_{width}$

To favor memory coalescing (i.e. the device accesses memory in a batch rather than individual accesses that would require serialized accesses to memory), it is useful to keep:

1. The G_{width} of the problem as a multiple of the maximum work-group size, eventually adding extra columns with appropriate padding. The maximum work-group size is given by `cl.KERNEL_WORK_GROUP_SIZE`
2. The L_{width} of a work-group as a multiple of the warp/wavefront size. This value is given by `cl.KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE`

Both limits can be queried on a WebCLKernel object once it is created. They are extremely important for maximum throughput.

Host and devices communicate via buffers defined in an OpenCL context. Commands are sent to devices via command-queues. Commands are used for memory transfers from host and devices, between memory objects in a device, and to execute programs.



5 Programming with WebCL

Programming with WebCL is composed of 2 parts:

- The host side (e.g. in the web browser) that sets up and controls the execution of the program
- The device side (e.g. on a GPU) that runs computations i.e. kernels.

5.1 Host/Browser side

All WebCL methods may throw exceptions (rather than error codes as in WebGL), so you should wrap your WebCL methods with try/catch, even though for simplicity we will omit them in this document.

```

1  try {
2    webclobject.method(...);
3  }
4  catch(ex) {
5    // an exception occurred
6    log(ex);
7  }

```

Code 2 – Always wrap WebCL method calls with try/catch!

Unlike WebGL, WebCL is a global object so that it can be used in a Web page or within a Web Worker. Consequently, we first need to create a WebCL object:

```

1  var cl = new WebCL();

```

Code 3 – Creating the WebCL object.

The remainder of this section will detail how to use all WebCL objects in Figure 4.

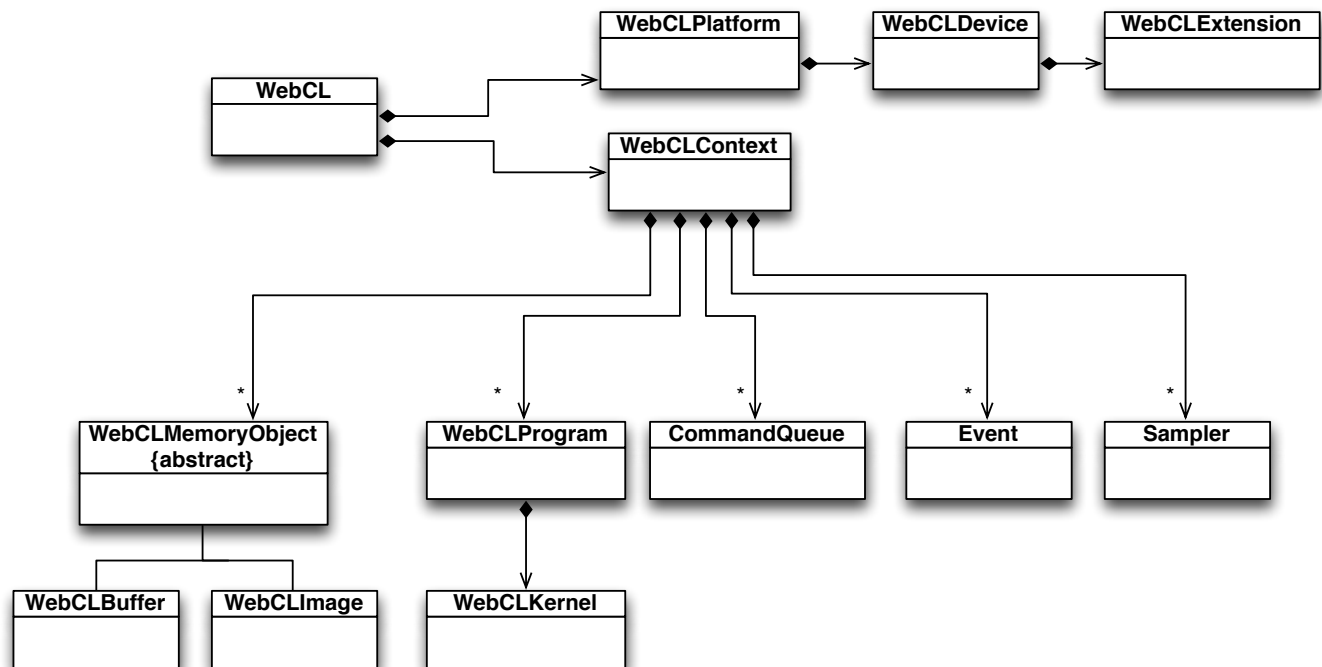


Figure 4 – WebCL objects.

The typical workflow is described in Figure 5 and consists in 3 phases:

- Initialize the platform layer
- Load and compile programs/kernels
- Interact with devices through the runtime layer

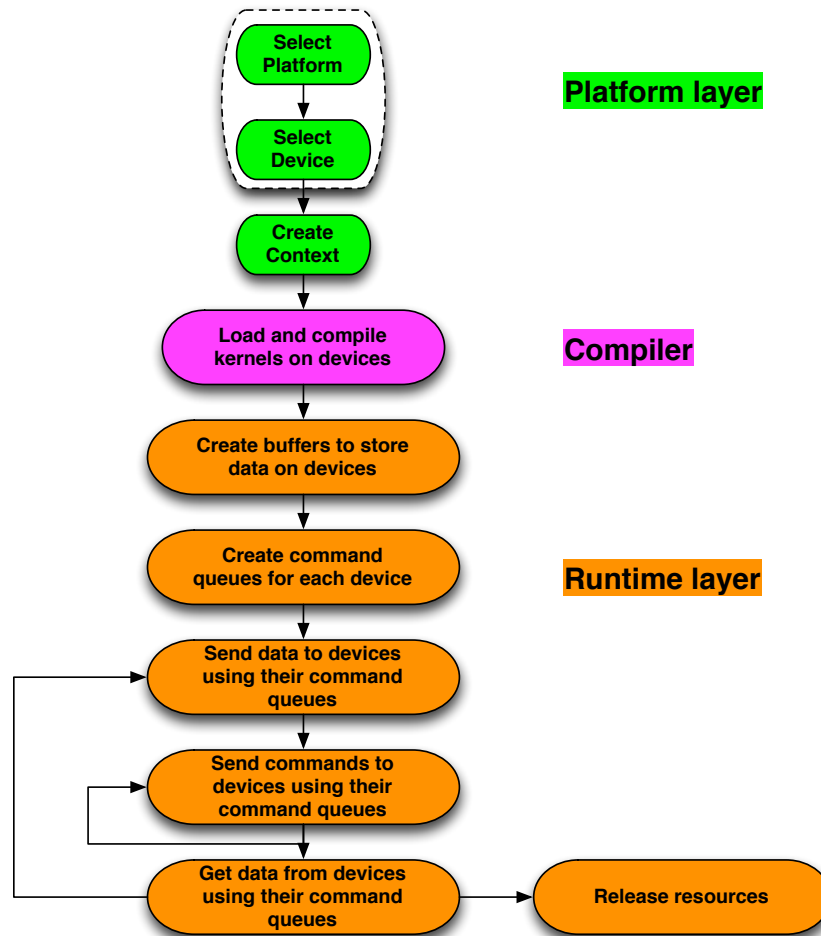


Figure 5 – OpenCL startup sequence

5.1.1 Platform layer

The OpenCL platform layer implements platform-specific features. They allow applications to query OpenCL devices, device configuration information, and to create OpenCL contexts using one or more devices.

```

1 // let's get all platforms on this machine
2 var platforms = cl.getPlatforms();
3
4 // dump information about each platform
5 for (var i = 0, il = platforms.length; i < il; ++i) {
6     var p = platforms[i];
7     var profile = p.getInfo(WebCL.PLATFORM_PROFILE);
8     var version = p.getInfo(WebCL.PLATFORM_VERSION);
9     var extensions = p.getInfo(WebCL.PLATFORM_EXTENSIONS);
10
11 // list of devices on this platform p
12 var devices = p.getDevices(WebCL.DEVICE_TYPE_ALL);
13
14 // find appropriate device
15 for (var j = 0, jl = devices.length; j < jl; ++j) {
16     var d = devices[j];
17     var devExts = d.getInfo(WebCL.DEVICE_EXTENSIONS);
18     var devGMem = d.getInfo( WebCL.DEVICE_GLOBAL_MEM_SIZE);
19     var devLMem = d.getInfo( WebCL.DEVICE_LOCAL_MEM_SIZE);
20     var devCompUnits = d.getInfo( WebCL.DEVICE_MAX_COMPUTE_UNITS);
21     var dev_hasImage = d.getInfo( WebCL.DEVICE_IMAGE_SUPPORT);
22     var devHasImage = d.getInfo( WebCL.DEVICE_IMAGE_SUPPORT);
23
24 // select device that match your requirements
25 ...
  
```



```

26     }
27
28     // assuming we found the best device, we can create the context
29     var context = cl.createContext( {
30         'platform': platform,
31         'device': device,
32     } );

```

Code 4 – Query WebCL platforms and devices features.

In general, to ensure your algorithm is portable across various devices (even on the same machine!), it is necessary to know details about features on each device. For example, if you require image support, ensure the device you choose support them and up to what size, and how many images can be supported at once. If your kernel requires atomics, make sure device's extensions return 'cl_khr_int64_base_atomics'. On embedded devices, knowing that 'cl_khr_fp16' is supported (i.e. 16-bit floats or half-floats) can lead to twice more performance. When optimizing algorithms, knowing the maximum workgroup size, the number of work-items per dimension, the number of parameters to a kernel function, the maximum size of a memory object, and other features, are crucial elements to adapt your applications at runtime.

On the other end, if you just want to use the best device on the machine and let the browser find it for you, you could just do:

```

1     var ctx = cl.createContext( {
2         deviceType : cl.DEVICE_TYPE_GPU
3     } );
4
5     // query the platform/device found by the browser
6     try {
7         devices = ctx.getInfo(cl.CONTEXT_DEVICES);
8         catch(ex) {
9             throw "Error: Failed to retrieve compute devices for context!";
10        }
11
12        var device = null, platform = null;
13
14        for(var i=0, il=devices.length; i < il; ++i) {
15            device_type = devices[i].getInfo(cl.DEVICE_TYPE);
16            if (device_type == cl.DEVICE_TYPE_GPU) {
17                device = devices[i];
18                break;
19            }
20        }
21
22        if (device)
23            platform = device.getInfo(cl.DEVICE_PLATFORM);

```

Code 5 – Let the browser figures the best platform/device for a context.

Note: in practice, the algorithm in Code 5 is often simplified with

```

1     var devices = ctx.getInfo(cl.CONTEXT_DEVICES);
2     var device = devices[0];
3     var platform = device.getInfo(cl.DEVICE_PLATFORM);

```

but this assumes the machine has only 1 GPU device!

Now that we have created a WebCLContext object, we need to set it up for our program and run it!

5.1.2 Runtime layer

The runtime layer manages OpenCL objects such as command-queues, memory objects, program objects, kernel objects in a program and calls that allow you to enqueue commands to a command-queue such as executing a kernel, reading, or writing a memory object.

WebCL defines the following objects:

- Command Queues
- Memory objects (Buffer and Images)
- Sampler objects describe how to sample an image being read by a kernel
- Program objects that contain a set of kernel functions identified with `__kernel` qualifier in the program source



- Kernel objects encapsulate the specific `__kernel` functions declared in a program source and its argument values to be used when executing the `__kernel` function
- Event objects used to track the execution status of a command as well as to profile a command
- Command synchronization objects such as Markers and Barriers

5.1.2.1 Loading and building programs

WebCL, like WebGL 1.0, assumes program to be provided in source code form i.e. a large string. Currently, any WebCL device is required to have an internal compiler. The source code is first loaded to the device, then compiled. As with any compiler, CL defines standard compilation options including the standard `-D` (predefined name and value) and `-I` (include directory). Code 6 shows how to properly catch compilation errors using `WebCLProgram.getBuildInfo()`.

```

1 // Create the compute program from the source strings
2 program = ctx.createProgram(source);
3
4 // Build the program executable with relaxed math flag
5 try {
6   program.build(device, "-cl-fast-relaxed-math");
7 } catch (err) {
8   throw 'Error building program: ' + err
9     + program.getBuildInfo(device, cl.PROGRAM_BUILD_LOG);
10 }

```

Code 6 – Load and build a CL program.

Note: WebCL currently only supports source code as a set of strings.

At this point, our program is compiled, and contains one or more kernel functions. These kernel functions are the entry points of our program, similar to entry points of a shared library. To refer to each kernel function, we create a `WebCLKernel` object:

```

1 // Create the compute kernels from within the program
2 kernel = program.createKernel('kernel_function_name');

```

Code 7 – Create a kernel object for each kernel function in the program.

In the next section, we will discover how to pass arguments to the kernel functions.

5.1.2.2 Passing arguments to kernels

A kernel function may have one or more arguments, like any function. Since JavaScript only offers the type Number for numerical values, we need to pass the type of such value to the kernel object for each argument. For other type of values, we must use WebCL objects:

- `WebCLBuffer` and `WebCLImage` that wrap a Typed Array [1]
- `WebCLSampler` for sampling an image

5.1.2.3 Creating memory objects

A `WebCLBuffer` object stores a one-dimensional collection of elements. Elements of a buffer can be scalar type (e.g. int, float), vector data type, or user-defined structure.

```

1 // create a 1D buffer
2 var buffer = ctx.createBuffer(flags, sizeInBytes, optional srcBuffer);

```

Flag	Description
<code>cl.MEM_READ_WRITE</code>	Default. Memory object is read and written by kernel
<code>cl.MEM_WRITE_ONLY</code>	Memory object only written by kernel
<code>cl.MEM_READ_ONLY</code>	Memory object only read by kernel
<code>cl.MEM_USE_HOST_PTR</code>	Implementation uses storage memory in <code>srcBuffer</code> . <code>srcBuffer</code> must be specified.
<code>cl.MEM_ALLOC_HOST_PTR</code>	Implementation requests OpenCL to allocate host memory.
<code>cl.MEM_COPY_HOST_PTR</code>	Implementation request OpenCL to allocate host memory and copy

data from srcBuffer memory. srcBuffer must be specified.

Reading from a WRITE_ONLY memory object, or Writing to a READ_ONLY memory object, is undefined. These flags are mutually exclusive.

srcBuffer must be a Typed Array already allocated by the application and sizeInBytes \geq srcBuffer.byteLength.

MEM_USE_HOST_PTR is mutually exclusive with MEM_ALLOC_HOST_PTR and MEM_COPY_HOST_PTR. However, MEM_COPY_HOST_PTR can be specified with MEM_ALLOC_HOST_PTR. On AMD and NVidia GPUs and on some operating systems, using MEM_ALLOC_HOST_PTR may result in pinned host memory to be used, which may result in improved performance [8][9].

A sub-buffer can be created from an existing WebCLBuffer object as a new WebCLBuffer object.

```
1 // create a sub-buffer
2 var subbuffer = buffer.createSubBuffer(flags, offset, size);
```

Note: only reading from a buffer object and its sub-buffer objects or reading from multiple overlapping sub-buffer objects is defined. All other concurrent reading or writing is undefined.

A WebCLImage is used to store a 1D, 2D, or 3D dimensional texture, render-buffer, or image. The elements of an image object are selected from a predefined list of image formats. However, currently, WebCL only supports 2D images.

```
1 // create a 32-bit RGBA WebCLImage object
2 // first, we define the format of the image
3 var InputFormat = {
4   'order' : cl.RGBA,
5   'data_type' : cl.UNSIGNED_INT8,
6   'size': [ image_width, image_height ],
7   'rowPitch': image_pitch
8 };
9
10 // Image on device
11 var image = ctx.createImage(cl.MEM_READ_ONLY | cl.MEM_USE_HOST_PTR, format, imageBuffer);
```

'order' refers to the memory layout in which pixel data channels are stored in the image. 'data_type' is the type of the channel data type.

'size' refers to the image size.

'rowPitch' refers to the scan-line pitch in bytes. If imageBuffer is null, it must be 0. Otherwise, it must be at least image_width * sizeInBytesOfChannelElement, which is the default if rowPitch is not specified.

imageBuffer is a Typed Array that contain the image data already allocated by the application. imageBuffer.byteLength \geq rowPitch * image_height. The size of each element in bytes must be a power of 2.

A WebCLSampler describes how to sample an image when the image is read in a kernel function. It is similar to WebGL samplers.

```
1 // create a sampler object
2 var sampler = ctx.createSampler(normalizedCoords, addressingMode, filterMode);
```

normalizedCoords is cl.TRUE or true indicates image coordinates specified are normalized.

addressingMode indicated how out-of-range image coordinates are handled when reading an image. This can be set to CL_ADDRESS_MIRRORED_REPEAT, CL_ADDRESS_REPEAT, CL_ADDRESS_CLAMP_TO_EDGE, CL_ADDRESS_CLAMP and CL_ADDRESS_NONE.

filterMode specifies the type of filter to apply when reading an image. This can be cl.FILTER_NEAREST or cl.FILTER_LINEAR.

5.1.2.4 Passing arguments to a kernel

Passing arguments to a kernel function is complicated by JavaScript un-typed nature: JavaScript provides a Number object and there is no way to know if this is a 32-bit integer, a 16-bit short, a 32-bit float, and so on. In fact, JavaScript numbers are typically 64-bit double. As a result, developers must provide the type of arguments used in a kernel function.



The WebCLKernel.setArg() method has two definitions: one for scalar and vector types and one for memory objects (buffers and images) and sampler objects. Table 1 provides the relationships between OpenCL C types and values used in kernel methods' arguments and setArg() arguments.

Values referring to local memory use the special type cl.type.LOCAL_MEMORY_SIZE because local variables can't be initialized by host or device but host can tell the device how many bytes to allocate for a kernel argument.

As a rule of thumb, scalar values are passed by value directly in setArg(). Buffers/Images/Vectors values are passed by commands to transfer their host memory to the device memory.

```

1 // Sets value of kernel argument idx with value of scalar/vector type
2 kernel.setArg(idx, value, type);
3
4 // Sets value of kernel argument idx with value as memory object or sampler
5 kernel.setArg(idx, a_webCLObject);

```

Code 8 – WebCLKernel.setArg() definition

For example,

```

1 // Sets value of argument 0 to the integer value 5
2 kernel.setArg(0, 5, cl.type.INT);
3
4 // Sets value of argument 1 to the float value 1.34
5 kernel.setArg(1, 1.34, cl.type.FLOAT);
6
7 // Sets value of argument 2 as a 3-float vector
8 // buffer should be a FloatBuffer
9 kernel.setArg(2, buffer, cl.type.FLOAT | cl.type.VEC3);
10
11 // Sets value of argument 3 to a buffer (same for image and sampler)
12 kernel.setArg(3, buffer);
13
14 // Allocate 4096 bytes of local memory for argument 4
15 kernel.setArg(4, 4096, cl.LOCAL_MEMORY_SIZE);

```

Code 9 – Setting kernel arguments.

Kernel argument type	setArg() value	setArg() cl.type	Remarks
char, uchar	scalar	CHAR, UCHAR	1 byte
short, ushort	scalar	SHORT, USHORT	2 bytes
int, uint	scalar	INT, UINT	4 bytes
long, ulong	scalar	LONG, ULONG	4 bytes
float	scalar	FLOAT	4 bytes
half, double	scalar	HALF, DOUBLE	No on all implementations 2 bytes (half), 8 bytes (double)
<char...double>N	WebCLBuffer	VECN	N = 2, 3, 4, 8, 16 May be null if global or constant value
char, ..., double *	WebCLBuffer		May be null if global or constant value
image2d_t	WebCLImage		
sampler_t	WebCLSampler		
__local		LOCAL_MEMORY_SIZE	Size initialized in kernel

Table 1 – Relationships between C types used in kernels and setArg()'s cl.type.*

If the argument of a kernel function is declared with the __constant qualifier, the size in bytes of the memory object cannot exceed cl.DEVICE_MAX_CONSTANT_BUFFER_SIZE.

Note 1: OpenCL allows passing structures as byte arrays to kernels but WebCL currently doesn't for portability. The main reason is that endianness between host and devices may be different and this would require developers to format their data for each device's endianness even on the same machine.

Note 2: all WebCL API calls are thread-safe, except `kernel.setArg()`. However, `kernel.setArg()` is safe as long as concurrent calls operate on different `WebCLKernel` objects. Behavior is undefined if multiple threads call on the same `WebCLKernel` object at the same time.

5.1.2.5 Controlling device execution with command queues

Operations on WebCL objects such as memory, program and kernel objects are performed using command queues. A command queue contains a set of operations or commands. Applications may use multiple independent command queues without synchronization as long as commands don't apply on shared objects between command queues. Otherwise, synchronization is required.

Commands are queued in order but execution may be in order (default) or out of order. This means that if a command-queue contains command A and command B, an in-order command-queue object guarantees that command B is executed when command A finishes. If an application configures a command-queue to be out-of-order, there is no guarantee that commands finish in the order they were queued. For out-of-order queues, a wait for events or a barrier command can be enqueued in the command-queue to guarantee previous commands finish before the next batch of commands is executed. Out-of-order queues are an advanced topic we won't cover in this course. Interested readers should refer to Derek Gerstmann Siggraph Asia 2009 on Advanced OpenCL Event Model Usage [20]. Moreover, device support for out-of-order queues is optional in OpenCL and many current drivers don't support it. It is useful to test for out-of-order support and, if an exception is thrown, then create an in-order queue.

```

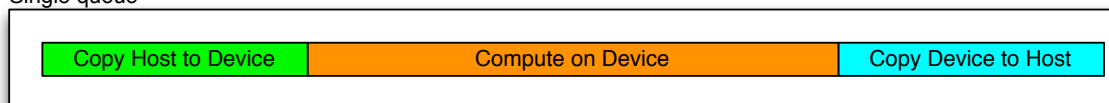
1 // Create an in-order command queue (default)
2 var queue = ctx.createCommandQueue(device);
3
4 // Create an in-order command queue with profiling of commands enabled
5 var queue = ctx.createCommandQueue(device, cl.QUEUE_PROFILING_ENABLE);
6
7 // Create an out-of-order command queue
8 var queue = ctx.createCommandQueue(device, cl.QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE);

```

Note: a command queue is attached to a specific device. And multiple command queues can be used per device. One application is to overlap kernel execution with data transfers between host and device [8]. Figure 6 shows the timing benefit if a problem could be separated in half:

- The first half of the data is transferred from host to, taking half the time of the full data set. Then kernel is executed, possibly in half time needed with the full data set. And finally result is transferred back to device in half the time of the full result set.
- Just after the first half is transferred, the second half is transferred from host to device, and the same process is repeated.

Single queue



Multiple queues

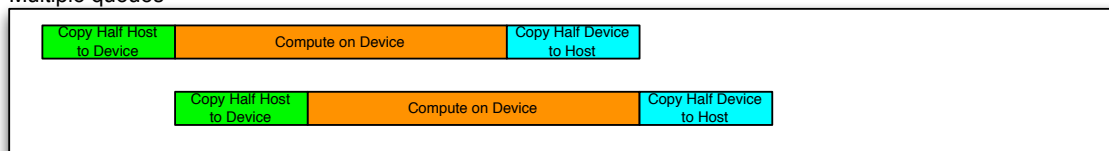


Figure 6 – Using multiple command-queues for overlapped data transfer.

5.1.2.6 Command-queue execution

Once a set of commands have been queued, WebCL offers two ways to execute the command-queue:

```

1 // execute a task
2 queue.enqueueTask(kernel);
3
4 // execute a NDRange

```




```
5 queue.enqueueNDRange(kernel, offsets, globals, locals);
```

With enqueueTask(), the kernel is executed using a single work-item. This is a very restricted form of enqueueNDRange().

enqueueNDRange() has first parameters:

- kernel – the kernel to execute
- offsets – offsets to apply to globals. If null, then offsets=[0, 0, 0]
- globals – the problem size per dimension
- locals – the number of work-items per work-group per dimension. If null, the device will choose the appropriate number of work-items

Recall Figure 3 where globals and locals relationships are depicted. If we want to execute a kernel over an image of size (width, height), then globals may be [width, height] and locals may be [16, 16].

Since enqueueNDRange() will fail if locals size is more than cl.KERNEL_WORK_GROUP_SIZE, in practice, it may be useful to do

```
1 locals[0] = kernel.getWorkGroupInfo(device, cl.KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE);
2 locals[1] = kernel.getWorkGroupInfo(device, cl.KERNEL_WORK_GROUP_SIZE) / locals[0];
3 globals[0] = locals[0] * divUp(width, locals[0]);
4 globals[1] = locals[1] * divUp(height, locals[1]);
5
6 // Helper to get next up value for integer division of x/y
7 function divUp(x, y) {
8     return (x % y == 0) ? (x / y) : (x / y + 1);
9 }
```

Code 10 – A way to optimally setup locals and globals NDRanges.

5.1.2.7 Command Synchronization

Nearly all commands available in WebCLCommandQueue class have two final parameters:

- event_list – an array of WebCLEvents
- event – an event returned by the device to monitor the execution status of a command

By default, event_list and event are null for a command, meaning that the command is executed as blocking the host thread until it is queued in the device's command queue. If a programmer doesn't want to block the host thread while a command is being executed, the device can return an event and the host code can register a callback to be notified once the command complete.

```
1 // Enqueue kernel
2 try {
3     kernel_event=new cl.WebCLEvent();
4     queue.enqueueTask(kernel, null, kernel_event);
5 } catch(ex) {
6     throw "Couldn't enqueue the kernel. "+ex;
7 }
8
9 // Set kernel event handling routines
10 try {
11     kernel_event.setCallback(cl.COMPLETE, kernel_complete, "The kernel finished successfully.");
12 } catch(ex) {
13     throw "Couldn't set callback for event. "+ex;
14 }
15
16 // Read the buffer
17 var data=new Float32Array(4096);
18 try {
19     read_event=new cl.WebCLEvent();
20     queue.enqueueReadBuffer(cl.Buffer, false, 0, 4096*4, data, null, read_event);
21 } catch(ex) {
22     throw "Couldn't read the buffer. "+ex;
23 }
24
25 // register a callback on completion of read_event
26 read_event.setCallback(cl.COMPLETE, read_complete, "Read complete");
27
28 // wait for both events to complete
29 queue.waitForEvents([kernel_event, read_event]);
30
31 // kernel callback
32 function kernel_complete(event, data) {
```



```

33 // event.status = cl.COMplete or error if negative
34 // event.data is null
35 // data should contain "The kernel finished successfully."
36 }
37
38 // read buffer callback
39 function read_complete(event, data) {
40 // event.status = cl.COMplete or error if negative
41 // event.data contains a WebCLMemoryObject with values from device
42 // data contains "Read complete"
43 }

```

Code 11 –Using WebCLEvent callbacks.

In Code 11, for the commands we wish to get notified on their `cl.COMplete` status, we first create a `WebCLEvent` object, pass it to the command, then register a JavaScript callback function.

Note 1: the last argument of `WebCLEvent.setCallback()` can be anything. And this argument is passed untouched as the last argument of the callback function.

Note 2: in the case of enqueue Read/Write `WebCLBuffers` or `WebCLImages`, as in line 22, `clBuffer` ownership is transferred from host to device. Thus, when `read_complete()` callback is called, `clBuffer` ownership is transferred back from device to host. This means that once the ownership of `clBuffer` is transferred (line 22), the host cannot access or use this buffer any more. Once the callback is called, line 40, the host can use the buffer again.

5.1.2.8 Profiling commands

To enable timing of commands, one creates a command-queue with option `cl.QUEUE_PROFILING_ENABLE`. Then, `WebCLEvents` can be used to time a command. Code 12 shows how to profile an `enqueueReadBuffer()` command.

```

1 // Create a command queue for profiling
2 try {
3 queue = context.createCommandQueue(device, cl.QUEUE_PROFILING_ENABLE);
4 } catch(ex) {
5 throw "Couldn't create a command queue for profiling. "+ex;
6 }
7
8 // Read the buffer with a profiling event
9 var prof_event=new cl.WebCLEvent();
10 try {
11 queue.enqueueReadBuffer(data_buffer, true, 0, data.byteLength, data, null, prof_event);
12 } catch(ex) {
13 throw "Couldn't read the buffer. "+ex;
14 }
15
16 // Get profiling information in nanoseconds
17 time_start = prof_event.getProfilingInfo(cl.PROFILING_COMMAND_START);
18 time_end = prof_event.getProfilingInfo(cl.PROFILING_COMMAND_END);
19 total_time = time_end - time_start;

```

Code 12 – How to profile a command.

Note: timestamps are given in nanoseconds (10^{-9} seconds).

Likewise, to profile the duration of a kernel:

```

1 // Enqueue kernel
2 try {
3 queue.enqueueNDRangeKernel(kernel, null, globals, locals, null, prof_event);
4 } catch(ex) {
5 throw "Couldn't enqueue the kernel. "+ex;
6 }

```

Code 13 – Profiling a kernel.

5.2 Device side

Kernels are written in a derivative of C99 with the following caveats:

- A file may have multiple `__kernel` functions (similar to a library with multiple entry points)
- No recursion since there is no call stack on devices
- All functions are inlined to the kernel functions
- No dynamic memory (e.g. `malloc()`, `free()`...)



- No function pointer
- No standard libc libraries (e.g. memcpy(), strcmp()...)
- No standard data structures (except vector operations)
- Helper functions
 - Barriers
 - Work-item functions
 - Atomic operations
 - Vector operations
 - Math operations and fast native (hardware accelerated) math operations
 - IEEE754 floating-point
 - 16-bit floats and doubles (optional)
- Built-in data types
 - 8, 16, 32, 64-bit values
 - Image 2D (and 3D but not in WebCL 1.0), Sampler, Event
 - 2, 3, 4, 8, 16-component vectors
- New keywords
 - Function qualifiers: `__kernel`
 - Address space qualifiers: `__global`, `__local`, `__constant`, `__private` (default),
 - Access qualifiers: `__read_only`, `__write_only`, `__read_write`,
- Preprocessor directives (`#define`, `#pragma`)

Appendices A.3 and A.4 provide examples of kernels.

6 Interoperability with WebGL

Recall that WebCL is for computing, not for rendering. However, if your data already resides in the GPU and you need to render it, wouldn't it be faster to tell OpenGL to use it rather than reading it from the GPU memory to CPU memory and send it again to OpenGL on your GPU? This is where WebGL comes in.

Since WebCL is using data from WebGL, the WebGL context must be created first. Then, a shared WebCL context can be created. This GL share group object manages shared GL and CL resources such as

- Textures objects – contain texture data in image form,
- Vertex buffers objects (VBOs) – contains vertex data such as coordinates, colors, and normal vectors,
- Renderbuffer objects – contain images used with GL framebuffer objects.

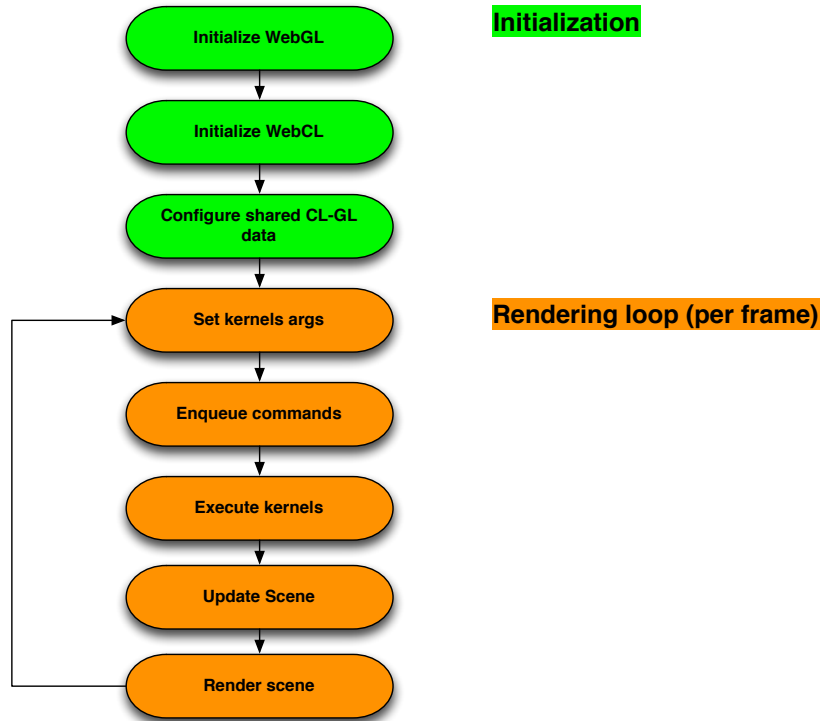


Figure 7 – Typical algorithm for WebCL – WebGL applications

6.1 Fun with 2 triangles

Applications such as image processing and ray tracing produce an output image whose pixels are drawn onto the screen. For such applications, it suffices to map the output image onto 2 unlit screen-aligned triangles rendered by GL. A compute kernel provides more flexible ways to optimize generic computations than a fragment shader. More importantly, texture memory is cached and thus provides a faster way to access data than regular (global) memory. However, in devices without image memory support, one should use WebCLBuffers and update GL textures with Pixel Buffer Objects.

In this section, we use Iñigo Quilez excellent ShaderToy's Mandelbulb fragment shader [24] converted as a CL kernel, depicted in Figure 8. The whole WebGL scene consists in 2 textured triangles filling a canvas. WebCL generates the texture at each frame. Therefore, for a canvas of dimension [width, height], WebCL will generate width * height pixels. We will detail each step and the full program is given in Appendix A. In [24], you can find more cool shaders that you can easily convert by the following the guidelines for this sample.

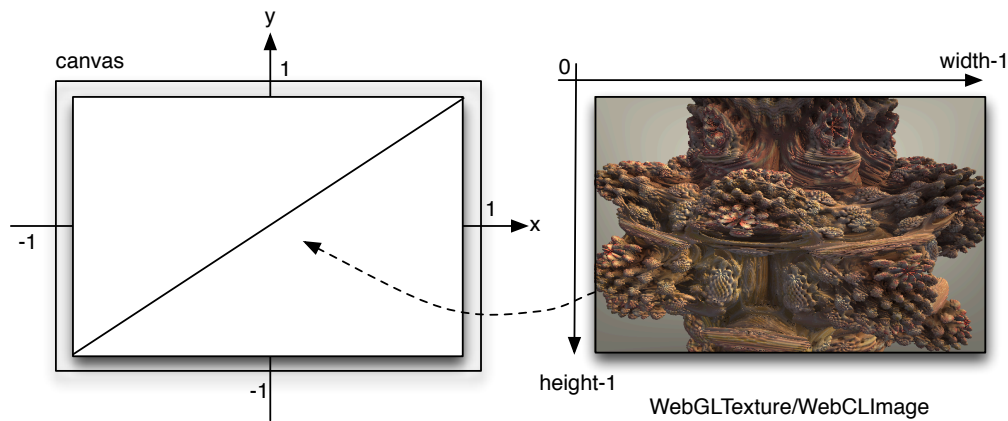


Figure 8 – Two triangles filling the canvas to draw a WebCL generated image.



6.1.1 General CL-GL interoperability algorithm

Since CL uses GL buffers for compute, WebGL context must first be initialized and then WebCL context is created sharing that WebGL context. Once both contexts are initialized, it is possible to create shared objects by creating first the WebGL object, then the corresponding WebCL object from the WebGL object.

The general algorithm is as follows:

```

1  function Init_GL() {
2      // Create WebGL context
3      // Init GL shaders
4      // Init GL buffers
5      // Init GL textures
6  }
7
8  function Init_CL() {
9      // Create WebCL context from WebGLContext
10     // Compile programs/kernels
11     // Create command queues
12     // Create buffers
13 }
14
15 function Create_shared_CLGL_objects {
16     // Create WebGL object glObj (vertex array, texture, renderbuffer)
17     // Create WebCL object clObj from WebGL object glObj
18 }
19
20 // called during rendering, possibly at each frame
21 // or in a separate Web Worker
22 function Execute_kernel(...) {
23     // Make sure all GL commands are finished
24     gl.flush();
25
26     // acquire shared WebCL object
27     queue.enqueueAcquireGLObjets(clObj);
28
29     // Execute CL kernel
30     // set global and local parameters
31     try {
32         queue.enqueueNDRangeKernel(kernel, null, global, local);
33     } catch (err) {
34         throw "Failed to enqueue kernel! " + err;
35     }
36
37     // Release CL object
38     queue.enqueueReleaseGLObjets(clObj);
39
40     // make sure all CL commands are finished
41     queue.flush();
42 }
43
44 // This is the main rendering method called at
45 // each frame
46 function display(timestamp) {
47     // Execute some GL commands
48     ...
49
50     Execute_kernel( ... );
51
52     // Execute more CL and GL commands
53     ...
54 }

```

Code 14 – General algorithm for WebCL-WebGL interoperability.

The remainder of this section will focus on how to create shared CLGL objects and how to use them.

6.1.2 Using shared textures

Initialize a WebCLImage object from a WebGLImage object as follows:

```

1  // Create OpenGL texture object
2  Texture = gl.createTexture();
3  gl.bindTexture(gl.TEXTURE_2D, Texture);
4  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);
5  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);

```



```

6  gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, TextureWidth, TextureHeight, 0, gl.RGBA,
   gl.UNSIGNED_BYTE, null);
7  gl.bindTexture(gl.TEXTURE_2D, null);
8
9  // Create OpenGL representation of OpenGL texture
10 try {
11     clTexture = ctx.createFromGLTexture2D(cl.MEM_WRITE_ONLY, gl.TEXTURE_2D, 0, Texture);
12 }
13 catch(ex) {
14     throw "Error: Failed to create WebGLImage. "+ex;
15 }
16
17 // To use this texture, somewhere in your code, do as usual:
18 glBindTexture(gl.TEXTURE_2D, Texture)

```

Code 15 – Initialize a WebCLImage object from a WebGLImage object.

Set the WebCLImage as an argument of your kernel:

```

1  kernel.setArg(0, clTexture);
2  kernel.setArg(1, TextureWidth, cl.type.UINT);
3  kernel.setArg(2, TextureHeight, cl.type.UINT);

```

Finally, here is how to use this WebCLImage inside your kernel code:

```

1  __kernel
2  void compute(__write_only image2d_t pix, uint width, uint height)
3  {
4      const int x = get_global_id(0);
5      const int y = get_global_id(1);
6
7      // compute pixel color as a float4
8
9      write_imagef(pix, (int2)(x,y), color);
10 }

```

Code 16 – Using a WebCLImage data inside a kernel.

Note: it should be possible to use `write_imagei()` or `write_imageui()` with `int4` colors. However, at the time of writing (May 2012), this doesn't seem to work with latest AMD and NVidia drivers. The code presented in this section is the only way I found to work with textures between CL and GL.

6.1.3 Using shared buffers

A WebCLBuffer is created from a WebGLBuffer as follows. On line 6, it is important to specify the correct `sizeInBytes` of the buffer.

```

1  // create a WebGLBuffer
2  pbo = gl.createBuffer();
3  gl.bindBuffer(gl.ARRAY_BUFFER, pbo);
4
5  // buffer data
6  gl.bufferData( gl.ARRAY_BUFFER, sizeInBytes, gl.DYNAMIC_DRAW);
7  gl.bindBuffer( gl.ARRAY_BUFFER, null);
8
9  // Create WebCLBuffer from WebGLBuffer
10 try {
11     clPBO = context.createFromGLBuffer( cl.MEM_WRITE_ONLY, pbo);
12 }
13 catch(ex) {
14     throw "Error: Failed to create WebCLBuffer. "+ex;
15 }

```

Code 17 – Using a WebCLImage data inside a kernel.

Since a GL `ARRAY_BUFFER` can be used for vertices, normals, colors, texture coordinates, texture data, and more, WebCL can be used to schedule processing of these buffers.

If the device doesn't support texture interoperability between CL and GL, a buffer can be used to update a WebGLImage sub-texture as follows with the assumption that WebCLBuffer contains RGBA values for each pixel.

```

1  // Create OpenGL texture object
2  Texture = gl.createTexture();
3  gl.bindTexture(gl.TEXTURE_2D, Texture);
4  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);
5  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);

```



```

6  gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, TextureWidth, TextureHeight, 0, gl.RGBA,
   gl.UNSIGNED_BYTE, null);
7  gl.bindTexture(gl.TEXTURE_2D, null);
8
9  // To render using this texture
10 gl.bindTexture(gl.TEXTURE_2D, TextureId);
11 gl.bindBuffer(gl.PIXEL_UNPACK_BUFFER, pbo);
12 gl.texSubImage2D(gl.TEXTURE_2D, 0, 0, 0, TextureWidth, TextureHeight, gl.RGBA, gl.UNSIGNED_BYTE,
   null);

```

Code 18 – Updating a texture from a WebGLBuffer

6.1.4 Example

The following example consists of 2 module objects, whose code is given in Appendix A:

- Graphics – encapsulates WebGL calls
- Compute – encapsulates WebCL calls

The code is rather large for just setting up WebCL and WebGL but, fear not, this is just boilerplate you can reuse!

The main method works as follows:

- Create a Canvas object
- Instantiate Graphics and Compute objects
- Launch the main rendering method
 - If the window is resized, we call Graphics to configure the shared GL texture. Then, we call Compute to configure the CL texture from this GL texture.
 - At each frame, we reset the kernel argument with the current timestamp in seconds. Then, the kernel is executed.
 - Finally, Graphics renders the frame

```

1  var COMPUTE_KERNEL_ID = "mandelbulb.cl"; // <script> id
2  var COMPUTE_KERNEL_NAME = "compute";    // name of __kernel
3  var Width;
4  var Height;
5  var Reshaped = true;
6  var log = console.log;
7
8  /*
9   * reshape() is called if document is resized
10  */
11 function reshape(evt) {
12     Width = evt.width;
13     Height = evt.height;
14     Reshaped = true;
15 }
16
17 (function main() {
18     log('Initializing');
19
20     document.setTitle("Mandelbulb demo");
21     var canvas = document.createElement("mycanvas", Width, Height);
22
23     // install UX callbacks
24     document.addEventListener('resize', reshape);
25
26     // init WebGL
27     var gfx=Graphics();
28     try {
29         gfx.init(canvas);
30     }
31     catch(err) {
32         alert('[Error] While initializing GL: '+err);
33         gfx.clean();
34         return;
35     }
36
37     // init WebCL
38     var compute=Compute();
39     try {
40         compute.init(gfx.gl(), COMPUTE_KERNEL_ID, COMPUTE_KERNEL_NAME);
41     }
42     catch(err) {
43         alert('[Error] While initializing CL: '+err);

```



```

44     compute.clean();
45     return;
46 }
47
48 // render scene
49 var startTime=-1;
50 var fpsFrame=0, fpsTo=0;
51
52 (function update(timestamp) {
53     if(timestamp) {
54         if(startTime===-1) {
55             startTime=fpsTo=timestamp;
56         }
57         var ltime = timestamp-startTime;
58     }
59
60     // reinit shared data if document is resized
61     if (Reshaped) {
62         log('reshaping texture');
63         try {
64             var glTexture=gfx.configure_shared_data(Width,Height);
65             var clTexture=compute.configure_shared_data(gfx.gl(), glTexture);
66             Reshaped=false;
67         }
68         catch(err) {
69             alert('[Error] While reshaping shared data: '+ex);
70             return;
71         }
72     }
73
74     // set kernel arguments
75     compute.resetKernelArgs(ltime/1000.0, Width, Height);
76
77     // compute texture for this timestamp
78     try {
79         compute.execute_kernel(gfx.gl());
80     }
81     catch(err) {
82         alert('[Error] While executing kernel: '+ex);
83         return;
84     }
85
86     // render scene with updated texture from CL
87     try {
88         gfx.display(ltime);
89     }
90     catch(err) {
91         alert('[Error] While rendering scene '+err);
92         return;
93     }
94
95     // Calculate framerate
96     fpsFrame++;
97     var dt=timestamp - fpsTo;
98     if( dt>1000 ) {
99         var ffps = 1000.0 * fpsFrame / dt;
100        log("myFramerate: " + ffps.toFixed(1) + " fps");
101        fpsFrame = 0;
102        fpsTo = timestamp;
103    }
104
105    requestAnimationFrame(update);
106 })();
107 }());

```

Code 19 – Main method for CL-GL program.

The kernel for such applications has the form:

```

1  __kernel
2  void compute(__write_only image2d_t pix, const float time)
3  {
4      const int x = get_global_id(0);
5      const int y = get_global_id(1);
6      const int xl = get_local_id(0);

```




```
7   const int yl = get_local_id(1);
8   const int tid = xl+yl*get_local_size(0); // local work-item ID
9   const int width = get_global_size(0);
10  const int height = get_global_size(1);
11
12  // init local memory
13  ...
14  // perform interesting computations for pixel (x,y)
15  ...
16  // write (r,g,b,a) value at pixel (x,y)
17  write_imagef(pix, (int2)(x,y), rgba);
18 }
```

Code 20 – Kernel for texture-based rendering.

Note 1: we don't pass the size of the shared texture since the dimension of our problem is the full size of the texture itself. In other words, when executing the kernel with `enqueueNDRange()`, the `globals` argument is [`width`, `height`], and that's what we retrieve in lines 9 and 10 in Code 20.

Note 2: for this example, we only pass the timestamp of the current frame to the kernel. For user interactivity, one should also pass mouse coordinates, window size, and other user/application attributes.

In Appendix, we provide the fragment shader code of the Mandelbulb shader by Iñigo Quilez [24], as well as the direct transformation to OpenCL and an optimized OpenCL version. We chose this example because the ray-marching algorithm (also known as sphere tracing [25]) used to render the mandelbulb fractal requires lots of operations per pixel; a good candidate for CL optimizations. Note that this is not necessarily the fastest way to render such mathematical objects. On our machine, this leads to 6 fps for WebGL version [24], 8 fps for non-optimized OpenCL version (Appendix A.2), and 12 fps for the optimized OpenCL version (Appendix A.4).

6.2 Other applications

In general, CL applications perform many matrix operations, whether the result is to be rendered directly onto the screen (e.g. in a texture) or not. For example, the famous N-body simulation calculates at each frame the position of astronomical objects, which are then rendered by GL [23]. An array of structures that contains position and other attributes is shared between host and device; the device performing all the calculations of the interactions between objects.

CL can also share vertex buffers and render buffers with GL. This allows developers to do all kind of complex geometry and special effects that can be inserted in GL's rendering pipeline.

7 Tips and tricks

NVidia and AMD excellent programming guides [8][9] provide lots of tips to optimize OpenCL programs. In our experience, we recommend following this strategy:

- Use Host code for serial code, use Device code for parallel code
- Write your code in serialized form (i.e. test it on a host CPU) and identify the areas that are good candidates for data-parallel optimizations
 - As a rule of thumb: identify where iterations are repeated on data, these are good candidates for data-parallel optimizations
- In your kernel, initialize first local memory with data from global memory that will be used often in your algorithm
- Group memory transfers together, this favors memory coalescing
- Identify where synchronization between work-items of the same work-group is necessary
- At the end of your kernel, write results from local memory back to global memory
- Rewrite your algorithm to minimize control flow divergence (i.e. `if`, `switch`, `for`, `do`, `while`). If threads in the same warp/wavefront take different execution paths, these execution paths will be serialized, thereby reducing throughput until the execution paths converge again.



7.1 From GLSL to OpenCL C

In converting GLSL shader to OpenCL C, we recommend following these guidelines:

- GLSL's `vecN` type are changed to OpenCL's `floatN` type
 - Initializations in OpenCL are: `(floatN)(val1,...,valN)` instead of `vecN(val1,...,valN)` in GLSL
- by default all floating point values are double in CL, make sure to add 'f' at the end.
- out arguments of methods must be pointers
- if numerical precision is not too important, compile with `-cl-fast-relaxed-math`, `-cl-mad-enable`, and use `native_*` functions (i.e. `native_sin()` instead of `sin()`).
- Use `rsqrt()` instead of `1.0f/sqrt()`

7.2 Barriers

Barriers are an important mechanism to wait for all work-items to be synchronized at points in the code. However, it is very important NOT to use barriers in `if/else` constructs. The reason is that some work items may not sync at the barrier, while others may block at the barrier; resulting in a deadlock of the GPU (i.e. you would have to reset your machine!).

The pattern to use a barrier is:

- Load values into local memory
- `barrier(CL_LOCAL_MEM_FENCE);` // wait for load to finish
- Use local memory in your algorithm
- `barrier(CL_LOCAL_MEM_FENCE);` // wait for all work-items

7.3 Local work-group size

When running a kernel, the method `enqueueNDRangeKernel()`, takes the parameters:

- `global_work_size` – the global number of work-items in N dimensions i.e. the size of the problem.
- `local_work_size` – the number of work-items that make up a work-group. Synchronization between work-items (with barriers) can only be within a work-group.

If `local_work_size[0] * local_work_size[1] * ... * local_work_size[N-1] > kernel.getWorkGroupInfo(device, cl.KERNEL_WORK_GROUP_SIZE)`, the program won't execute!

`cl.KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE` can be used to make block-size multiple of that size. AMD calls that size wavefront size and NVidia calls it warp size. Note: this value is often 32 for NVidia GPUs, 64 for AMD GPUs.

Since kernels can't allocate memory dynamically, one trick could be to compile a small program to get such kernel dependent values, add them on top of your real program code as constants (or `#define`) before compiling it.

7.4 Learn parallel patterns!

Parallel programming is not new. In fact, it might be as old as modern computers. Since the 60s lots of work has been done on supercomputers and many patterns have been found but they are still an active area of research. Learning how to use these patterns can simplify your code and more importantly lead to faster performance for your programs [13][21]. Algorithms such as map, reduce, scan, scatter/gather, stencils, pack [21], Berkeley Parallel Computing Laboratory's pattern language for parallel computing [32], and Murray Cole's algorithmic skeletons [0] are examples of such parallel algorithms and methodologies you need to know.

8 WebCL implementations

At the time of writing, the following prototypes are available:

- Nokia WebCL prototype [16] as a Mozilla FireFox extension
- Mozilla FireFox implementation [18]
- Samsung WebKit prototype [17]
- Motorola Mobility node-webcl module [15], a Node.JS based implementation.



Motorola Mobility node-webl implementation is based on Node.JS, which uses Google V8 JavaScript engine, as in Google Chrome browser. This implementation is up to date with the latest WebCL specification and allows quick prototyping of WebCL features before they become available in browsers. Coupled with Node.JS features, it also enables server-side applications using WebCL. All examples in this course have been developed and tested first with node-webl.

9 Perspectives

This course provided the foundations for developers to experiment with the exciting world of high-performance computing on the web. OpenCL is a rather young technology and it is not uncommon to find bugs in current implementations. However, WebCL implementations would abstract these technical issues for safer, more robust, more secure, and more portable applications, as the specification mature with feedback from users, hardware manufacturers and browser vendors. Meanwhile, prototype WebCL implementations are already available and we hope this course gave you all the excitement to start hacking your GPUs today for cool applications tomorrow ☺



Appendix A CL-GL code

This appendix provides source code for applications described in section 6.1.4. The first two sections provide the Graphics and Compute module objects. The third section is a direct translation from GLSL to OpenCL kernel language using techniques described in section 7.1. The last section is an example optimized version using local memory and work-groups.

A.1 Graphics object

```

1  /*
2  * Graphics module object contains all WebGL initializations for a simple
3  * 2-triangle textured screen aligned scene and its rendering.
4  */
5  function Graphics() {
6      var gl;
7      var shaderProgram;
8      var TextureId = null;
9      var VertexPosBuffer, TexCoordsBuffer;
10
11     /*
12     * Init WebGL array buffers
13     */
14     function init_buffers() {
15         log(' create buffers');
16         var VertexPos = [ -1, -1,
17                         1, -1,
18                         1, 1,
19                         -1, 1 ];
20         var TexCoords = [ 0, 0,
21                          1, 0,
22                          1, 1,
23                          0, 1 ];
24
25         VertexPosBuffer = gl.createBuffer();
26         gl.bindBuffer(gl.ARRAY_BUFFER, VertexPosBuffer);
27         gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(VertexPos), gl.STATIC_DRAW);
28         VertexPosBuffer.itemSize = 2;
29         VertexPosBuffer.numItems = 4;
30
31         TexCoordsBuffer = gl.createBuffer();
32         gl.bindBuffer(gl.ARRAY_BUFFER, TexCoordsBuffer);
33         gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(TexCoords), gl.STATIC_DRAW);
34         TexCoordsBuffer.itemSize = 2;
35         TexCoordsBuffer.numItems = 4;
36     }
37
38     /*
39     * Compile vertex and fragment shaders
40     *
41     * @param gl WebGLContext
42     * @param id <script> id where the source of the shader resides
43     */
44     function compile_shader(gl, id) {
45         var shaders = {
46             "shader-vs" : [
47                 "attribute vec3 aCoords;",
48                 "attribute vec2 aTexCoords;",
49                 "varying vec2 vTexCoords;",
50                 "void main(void) {",
51                 "    gl_Position = vec4(aCoords, 1.0);",
52                 "    vTexCoords = aTexCoords;",
53                 "}"] .join("\n"),
54             "shader-fs" : [
55                 "#ifdef GL_ES",
56                 "    precision mediump float;",
57                 "#endif",
58                 "varying vec2 vTexCoords;",
59                 "uniform sampler2D uSampler;",
60                 "void main(void) {",

```



```

61         "    gl_FragColor = texture2D(uSampler, vTexCoords.st);",
62         "]" ].join("\n"),
63     });
64
65     var shader;
66     var str = shaders[id];
67
68     if (id.match(/-fs/)) {
69         shader = gl.createShader(gl.FRAGMENT_SHADER);
70     } else if (id.match(/-vs/)) {
71         shader = gl.createShader(gl.VERTEX_SHADER);
72     } else {
73         throw 'Shader '+id+' not found';
74     }
75
76     gl.shaderSource(shader, str);
77     gl.compileShader(shader);
78
79     if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
80         throw gl.getShaderInfoLog(shader);
81     }
82
83     return shader;
84 }
85
86 /*
87  * Initialize vertex and fragment shaders, link program and scene objects
88  */
89 function init_shaders() {
90     log(' Init shaders');
91     var fragmentShader = compile_shader(gl, "shader-fs");
92     var vertexShader = compile_shader(gl, "shader-vs");
93
94     shaderProgram = gl.createProgram();
95     gl.attachShader(shaderProgram, vertexShader);
96     gl.attachShader(shaderProgram, fragmentShader);
97     gl.linkProgram(shaderProgram);
98
99     if (!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS))
100         throw "Could not link shaders";
101
102     gl.useProgram(shaderProgram);
103
104     shaderProgram.vertexPositionAttribute = gl.getAttribLocation(shaderProgram, "aCoords");
105     gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);
106
107     shaderProgram.textureCoordAttribute = gl.getAttribLocation(shaderProgram, "aTexCoords");
108     gl.enableVertexAttribArray(shaderProgram.textureCoordAttribute);
109
110     shaderProgram.samplerUniform = gl.getUniformLocation(shaderProgram, "uSampler");
111 }
112
113 /*
114  * Render the scene at a timestamp.
115  *
116  * @param timestamp in ms as given by new Date().getTime()
117  */
118 function display(timestamp) {
119     // we just draw a screen-aligned texture
120     gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);
121
122     gl.enable(gl.TEXTURE_2D);
123     gl.bindTexture(gl.TEXTURE_2D, TextureId);
124
125     // draw screen aligned quad
126     gl.bindBuffer(gl.ARRAY_BUFFER, VertexPosBuffer);
127     gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
128         VertexPosBuffer.itemSize, gl.FLOAT, false, 0, 0);
129
130     gl.bindBuffer(gl.ARRAY_BUFFER, TexCoordsBuffer);
131     gl.vertexAttribPointer(shaderProgram.textureCoordAttribute,
132         TexCoordsBuffer.itemSize, gl.FLOAT, false, 0, 0);
133
134     gl.activeTexture(gl.TEXTURE0);
135     gl.uniform1i(shaderProgram.samplerUniform, 0);

```



```

136
137     gl.drawArrays(gl.TRIANGLE_FAN, 0, 4);
138
139     gl.bindTexture(gl.TEXTURE_2D, null);
140     gl.disable(gl.TEXTURE_2D);
141
142     gl.flush();
143 }
144
145 /*
146  * Initialize WebGL
147  *
148  * @param canvas HTML5 canvas object
149  */
150 function init(canvas) {
151     log('Init GL');
152     gl = canvas.getContext("experimental-webgl");
153     gl.viewportWidth = canvas.width;
154     gl.viewportHeight = canvas.height;
155
156     init_buffers();
157     init_shaders();
158 }
159
160 /*
161  * Configure shared data i.e. our WebGLImage
162  *
163  * @param TextureWidth width of the shared texture
164  * @param TextureHeight height of the shared texture
165  */
166 function configure_shared_data(TextureWidth, TextureHeight) {
167     if (TextureId) {
168         gl.deleteTexture(TextureId);
169         TextureId = null;
170     }
171
172     gl.viewportWidth = TextureWidth;
173     gl.viewportHeight = TextureHeight;
174
175     // Create OpenGL texture object
176     gl.activeTexture(gl.TEXTURE0);
177     TextureId = gl.createTexture();
178     gl.bindTexture(gl.TEXTURE_2D, TextureId);
179     gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);
180     gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);
181     gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, TextureWidth, TextureHeight, 0,
182         gl.RGBA, gl.UNSIGNED_BYTE, null);
183     gl.bindTexture(gl.TEXTURE_2D, null);
184
185     return TextureId;
186 }
187
188 return {
189     'gl': function() { return gl; },
190     'TextureId': function() { return TextureId; },
191     'configure_shared_data': configure_shared_data,
192     'init': init,
193     'display': display,
194     'clean': function() {}
195 };
196 }

```

A.2 Compute object

The compute object reads a kernel from a string. The string may come from a `<script type="x-webcl">` or from a file.

```

1 /*
2  * Compute contains all WebCL initializations and runtime for our kernel
3  * that update a texture.
4  */
5 function Compute() {
6     var cl=new WebCL();
7     var /* cl_context */      clContext;

```



```

8   var /* cl_command_queue */ clQueue;
9   var /* cl_program */      clProgram;
10  var /* cl_device_id */     clDevice;
11  var /* cl_device_type */   clDeviceType = cl.DEVICE_TYPE_GPU;
12  var /* cl_image */        clTexture;
13  var /* cl_kernel */       clKernel;
14  var max_workgroup_size, max_workitem_sizes, warp_size;
15  var TextureWidth, TextureHeight;
16  var COMPUTE_KERNEL_ID;
17  var COMPUTE_KERNEL_NAME;
18  var nodejs = (typeof window === 'undefined');
19
20  /*
21   * Initialize WebCL context sharing WebGL context
22   *
23   * @param gl WebGLContext
24   * @param kernel_id the <script> id of the kernel source code
25   * @param kernel_name name of the __kernel method
26   */
27  function init(gl, kernel_id, kernel_name) {
28    log('init CL');
29    if(gl === 'undefined' || kernel_id === 'undefined'
30       || kernel_name === 'undefined')
31      throw 'Expecting init(gl, kernel_id, kernel_name)';
32
33    COMPUTE_KERNEL_ID = kernel_id;
34    COMPUTE_KERNEL_NAME = kernel_name;
35
36    // Pick platform
37    var platformList = cl.getPlatforms();
38    var platform = platformList[0];
39
40    // create the OpenCL context
41    clContext = cl.createContext({
42      deviceType: clDeviceType,
43      shareGroup: gl,
44      platform: platform });
45
46    var device_ids = clContext.getInfo(cl.CONTEXT_DEVICES);
47    if (!device_ids) {
48      throw "Error: Failed to retrieve compute devices for context!";
49    }
50
51    var device_found = false;
52    for(var i=0,l=device_ids.length;i<l;++i) {
53      device_type = device_ids[i].getInfo(cl.DEVICE_TYPE);
54      if (device_type == clDeviceType) {
55        clDevice = device_ids[i];
56        device_found = true;
57        break;
58      }
59    }
60
61    if (!device_found)
62      throw "Error: Failed to locate compute device!";
63
64    // Create a command queue
65    try {
66      clQueue = clContext.createCommandQueue(clDevice, 0);
67    }
68    catch(ex) {
69      throw "Error: Failed to create a command queue! "+ex;
70    }
71
72    // Report the device vendor and device name
73    var vendor_name = clDevice.getInfo(cl.DEVICE_VENDOR);
74    var device_name = clDevice.getInfo(cl.DEVICE_NAME);
75    log("Connecting to " + vendor_name + " " + device_name);
76
77    if (!clDevice.getInfo(cl.DEVICE_IMAGE_SUPPORT))
78      throw "Application requires images: Images not supported on this device.";
79
80    init_cl_buffers();
81    init_cl_kernels();
82  }

```



```

83
84  /*
85   * Initialize WebCL kernels
86   */
87  function init_cl_kernels() {
88      log('  setup CL kernel');
89
90      clProgram = null;
91
92      if(!nodejs) {
93          var sourceScript = document.getElementById(COMPUTE_KERNEL_ID);
94          if (!sourceScript)
95              throw "Can't find CL source <script>";
96
97          var str = "";
98          var k = sourceScript.firstChild;
99          while (k) {
100             if (k.nodeType == 3) {
101                 str += k.textContent;
102             }
103             k = k.nextSibling;
104         }
105         if (sourceScript.type == "x-webcl")
106             source = str;
107         else
108             throw "<script> type should be x-webcl";
109     }
110     else {
111         log("Loading kernel source from file '" + COMPUTE_KERNEL_ID + "'.");
112         source = fs.readFileSync(__dirname + '/' + COMPUTE_KERNEL_ID, 'ascii');
113         if (!source)
114             throw "Error: Failed to load kernel source!";
115     }
116
117
118     // Create the compute program from the source buffer
119     try {
120         clProgram = clContext.createProgram(source);
121     }
122     catch(ex) {
123         throw "Error: Failed to create compute program! "+ex;
124     }
125
126     // Build the program executable
127     try {
128         clProgram.build(clDevice, '-cl-unsafe-math-optimizations -cl-single-precision-constant -cl-
fast-relaxed-math -cl-mad-enable');
129     } catch (err) {
130         throw "Error: Failed to build program executable!\n"
+ clProgram.getBuildInfo(clDevice, cl.PROGRAM_BUILD_LOG);
131     }
132
133
134     // Create the compute kernels from within the program
135     try {
136         clKernel = clProgram.createKernel(COMPUTE_KERNEL_NAME);
137     }
138     catch(ex) {
139         throw "Error: Failed to create compute row kernel! "+ex;
140     }
141
142     // Get the device intrinsics for executing the kernel on the device
143     max_workgroup_size = clKernel.getWorkGroupInfo(clDevice, cl.KERNEL_WORK_GROUP_SIZE);
144     max_workitem_sizes=clDevice.getInfo(cl.DEVICE_MAX_WORKITEM_SIZES);
145     warp_size=clKernel.getWorkGroupInfo(clDevice, cl.KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE);
146     log('  max workgroup size: '+max_workgroup_size);
147     log('  max workitem sizes: '+max_workitem_sizes);
148     log('  warp size: '+warp_size);
149 }
150
151 /*
152  * (Re-)set kernel arguments
153  *
154  * @param time timestamp in ms (as given by new Date().getTime())
155  * @param image_width width of the image
156  * @param image_height height of the image

```




```

157  */
158  function resetKernelArgs(time, image_width, image_height) {
159      TextureWidth = image_width;
160      TextureHeight = image_height;
161
162      // set the kernel args
163      try {
164          // Set the Argument values for the row kernel
165          clKernel.setArg(0, clTexture);
166          clKernel.setArg(1, time, cl.type.FLOAT);
167      } catch (err) {
168          throw "Failed to set row kernel args! " + err;
169      }
170  }
171
172  /*
173   * Initialize WebGL buffers
174   */
175  function init_cl_buffers() {
176      //log(' create CL buffers');
177  }
178
179  /*
180   * Configure shared data with WebGL i.e. our texture
181   *
182   * @param gl WebGLContext
183   * @param glTexture WebGLTexture to share with WebGL
184   */
185  function configure_shared_data(gl, glTexture) {
186      // Create OpenCL representation of OpenGL Texture
187      clTexture = null;
188      try {
189          clTexture = clContext.createFromGLTexture2D(cl.MEM_WRITE_ONLY,
190              gl.TEXTURE_2D, 0, glTexture);
191      } catch (ex) {
192          throw "Error: Failed to create CL Texture object. " + ex;
193      }
194
195      return clTexture;
196  }
197
198  /*
199   * Execute kernel possibly at each frame before rendering results with WebGL
200   *
201   * @param gl WebGLContext
202   */
203  function execute_kernel(gl) {
204      // Sync GL and acquire buffer from GL
205      gl.flush();
206      clQueue.enqueueAcquireGLObjects(clTexture);
207
208      // Set global and local work sizes for kernel
209      var local = [];
210      local[0] = warp_size;
211      local[1] = max_workgroup_size / local[0];
212      var global = [ clu.DivUp(TextureWidth, local[0]) * local[0],
213                   clu.DivUp(TextureHeight, local[1]) * local[1] ];
214
215      // default values
216      //var local = null;
217      //var global = [ TextureWidth, TextureHeight ];
218
219      try {
220          clQueue.enqueueNDRangeKernel(clKernel, null, global, local);
221      } catch (err) {
222          throw "Failed to enqueue kernel! " + err;
223      }
224
225      // Release GL texture
226      clQueue.enqueueReleaseGLObjects(clTexture);
227      clQueue.flush();
228  }
229
230  return {
231      'init':init,

```



```

232     'configure_shared_data': configure_shared_data,
233     'resetKernelArgs': resetKernelArgs,
234     'execute_kernel': execute_kernel,
235     'clean': function() {}
236 }
237 }

```

A.3 Mandelbulb kernel (direct conversion)

The Mandelbulb 3D fractal, raymarched and colored with orbit traps and fake ambient occlusion by Iñigo Quilez [24] with authorization, is converted directly to an OpenCL kernel.

```

1 // forward declarations
2 bool isphere( float4 sph, float3 ro, float3 rd, float2 *t );
3 bool iterate( float3 q, float *resPot, float4 *resColor );
4 bool ifractal( float3 ro, float3 rd, float *rest, float maxt, float3 *resnor, float4 *rescol,
5 float fov );
6
7 inline bool isphere( float4 sph, float3 ro, float3 rd, float2 *t ) {
8     float3 oc = ro - sph.xyz;
9     float b = dot(oc,rd);
10    float c = dot(oc,oc) - sph.w*sph.w;
11
12    float h = b*b - c;
13    if( h<0 )
14        return false;
15
16    float g = sqrt( h );
17    t->x = - b - g;
18    t->y = - b + g;
19
20    return true;
21 }
22
23 #define NumIte 7
24 #define Bailout 100
25
26 inline bool iterate( float3 q, float *resPot, float4 *resColor ) {
27     float4 trap = (float4)(100);
28     float3 zz = q;
29     float m = dot(zz,zz);
30     if( m > Bailout ) {
31         *resPot = 0.5f*log(m); ///pow(8.0f,0.0f);
32         *resColor = (float4)(1);
33         return false;
34     }
35
36     for( int i=1; i<NumIte; i++ ) {
37         float x = zz.x; float x2 = x*x; float x4 = x2*x2;
38         float y = zz.y; float y2 = y*y; float y4 = y2*y2;
39         float z = zz.z; float z2 = z*z; float z4 = z2*z2;
40
41         float k3 = x2 + z2;
42         float k2 = rsqrt( k3*k3*k3*k3*k3*k3*k3 );
43         float k1 = x4 + y4 + z4 - 6*y2*z2 - 6*x2*y2 + 2*z2*x2;
44         float k4 = x2 - y2 + z2;
45
46         zz.x = q.x + 64*x*y*z*(x2-z2)*k4*(x4-6.0*x2*z2+z4)*k1*k2;
47         zz.y = q.y + -16*y2*k3*k4*k4 + k1*k1;
48         zz.z = q.z + -8*y*k4*(x4*x4 - 28*x4*x2*z2 + 70*x4*z4 - 28*x2*z2*z4 + z4*z4)*k1*k2;
49
50         m = dot(zz,zz);
51
52         trap = min( trap, (float4)(zz.xyz*zz.xyz,m) );
53
54         if( m > Bailout )
55         {
56             *resColor = trap;
57             *resPot = 0.5f*log(m)/pow(8.0f,i);
58             return false;
59         }
60     }

```



```

61     *resColor = trap;
62     *resPot = 0;
63     return true;
64 }
65
66 inline bool ifractal( float3 ro, float3 rd, float *rest, float maxt,
67     float3 *resnor, float4 *rescol, float fov ) {
68     float4 sph = (float4)( 0.0, 0.0, 0.0, 1.25 );
69     float2 dis;
70
71     // bounding sphere
72     if( !isphere(sph,ro,rd,&dis) )
73         return false;
74
75     // early skip
76     if( dis.y<0.001f ) return false;
77
78     // clip to near!
79     if( dis.x<0.001f ) dis.x = 0.001f;
80
81     if( dis.y>maxt) dis.y = maxt;
82
83     float dt;
84     float3 gra;
85     float4 color;
86     float4 col2;
87     float pot1;
88     float pot2, pot3, pot4;
89
90     float fovfactor = 1.0f/sqrt(1+fov*fov);
91
92     // raymarch!
93     for( float t=dis.x; t<dis.y; ) {
94         float3 p = ro + rd*t;
95
96         float Surface = clamp( 0.001f*t*fovfactor, 0.000001f, 0.005f );
97
98         float eps = Surface*0.1f;
99
100        if( iterate(p,&pot1,&color) ) {
101            *rest = t;
102            *resnor=normalize(gra);
103            *rescol = color;
104            return true;
105        }
106
107        iterate(p+(float3)(eps,0.0,0.0),&pot2,&col2);
108        iterate(p+(float3)(0.0,eps,0.0),&pot3,&col2);
109        iterate(p+(float3)(0.0,0.0,eps),&pot4,&col2);
110
111        gra = (float3)( pot2-pot1, pot3-pot1, pot4-pot1 );
112        dt = 0.5f*pot1*eps/length(gra);
113
114        if( dt<Surface ) {
115            *rescol = color;
116            *resnor = normalize( gra );
117            *rest = t;
118            return true;
119        }
120
121        t+=dt;
122    }
123
124    return false;
125 }
126
127 __kernel
128 void compute(__write_only image2d_t pix, float time) {
129     int x=get_global_id(0), y=get_global_id(1);
130     const int width = get_global_size(0);
131     const int height = get_global_size(1);
132     float2 resolution=(float2)(width,height);
133     float2 gl_FragCoord=(float2)(x,y);
134
135     float2 p = (float2)(-1.f + 2.f * gl_FragCoord.xy / resolution.xy);

```



```

136 float2 s = p*(float2)(1.33,1.0);
137
138 float3 light1 = (float3)( 0.577f, 0.577f, 0.577f );
139 float3 light2 = (float3)( -0.707f, 0, 0.707f );
140
141 float fov = 1;
142 float r = 1.4f+0.2f*cospi(2.f*time/20.f);
143 float3 campos = (float3)( r*sinpi(2.f*time/20.f),
144                          0.3f-0.4f*sinpi(2.f*time/20.f),
145                          r*cospi(2.f*time/20.f) );
146 float3 camtar = (float3)(0,0.1,0);
147
148 //camera matrix
149 float3 cw = normalize(camtar-campos);
150 float3 cp = (float3)(0,1,0);
151 float3 cu = normalize(cross(cw,cp));
152 float3 cv = normalize(cross(cu,cw));
153
154 // ray dir
155 float3 rd;
156 float3 nor, rgb;
157 float4 col;
158 float t;
159
160 rd = normalize( s.x*cu + s.y*cv + 1.5f*cw );
161
162 bool res=ifractal(campos,rd,&t,1e20f,&nor,&col,fov);
163
164 if( !res ) {
165     rgb = 1.3f*(float3)(1,.98,0.9)*(0.7f+0.3f*rd.y);
166 }
167 else {
168     float3 xyz = campos + t*rd;
169
170     // sun light
171     float dif1 = clamp( 0.2f + 0.8f*dot( light1, nor ), 0.f, 1.f );
172     dif1=dif1*dif1;
173
174     // back light
175     float dif2 = clamp( 0.3f + 0.7f*dot( light2, nor ), 0.f, 1.f );
176
177     // ambient occlusion
178     float ao = clamp(1.25f*col.w-.4f,0.f,1.f);
179     ao=0.5f*ao*(ao+1);
180
181     // shadow
182     if( dif1>0.001f ) {
183         float lt1;
184         float3 ln;
185         float4 lc;
186         if( ifractal(xyz,light1,&lt1,1e20,&ln,&lc,fov) )
187             dif1 = 0.1f;
188     }
189
190     // material color
191     rgb = (float3)(1);
192     rgb = mix( rgb, (float3)(0.8,0.6,0.2), (float3)(sqrt(col.x)*1.25f) );
193     rgb = mix( rgb, (float3)(0.8,0.3,0.3), (float3)(sqrt(col.y)*1.25f) );
194     rgb = mix( rgb, (float3)(0.7,0.4,0.3), (float3)(sqrt(col.z)*1.25f) );
195
196     // lighting
197     rgb *= (0.5f+0.5f*nor.y)*
198           (float3)(.14,.15,.16)*0.8f +
199           dif1*( float3)(1.0,.85,.4) +
200           0.5f*dif2*( float3)(.08,.10,.14);
201     rgb *= (float3)( pow(ao,0.8f), pow(ao,1.00f), pow(ao,1.1f) );
202
203     // gamma
204     rgb = 1.5f*(rgb*0.15f + 0.85f*sqrt(rgb));
205 }
206
207 float2 uv = 0.5f*(p+1.f);
208 rgb *= 0.7f + 0.3f*16.0f*uv.x*uv.y*(1.0f-uv.x)*(1.0f-uv.y);
209 rgb = clamp( rgb, (float3)(0), (float3)(1) );
210

```



```
211 write_imagef(pix,(int2)(x,y),(float4)(rgb,1.0f));
212 }
```

A.4 Mandelbulb kernel (optimized)

The main idea is to move to local memory all parameters necessary for computation.

```
1  #define WARPSIZE 256
2
3  typedef struct {
4      float3 origin;
5      float r;
6      float2 dis;
7  } Sphere;
8
9  typedef struct {
10     float3 origin;
11     float3 dir;
12     float3 nor;
13     float4 col;
14     float fovfactor;
15     float t;
16     float3 rgb;
17     Sphere sph;
18 } __attribute__((aligned(16))) Ray;
19
20
21 // forward declarations
22 bool isphere( __local Ray *ray );
23 bool iterate( const float3 q, float *resPot, float4 *resColor );
24 bool ifractal( __local Ray *ray);
25
26 inline bool isphere( __local Ray *ray ) {
27     const float3 oc = ray->origin - ray->sph.origin;
28     const float b = dot(oc,ray->dir);
29     const float c = dot(oc,oc) - ray->sph.r*ray->sph.r;
30
31     const float h = b*b - c;
32     if( h<0 )
33         return false;
34
35     const float g = native_sqrt( h );
36     ray->sph.dis = (float2) ( - b - g, - b + g);
37
38     return true;
39 }
40
41 __constant int NumIte=8;
42 __constant float Bailout=100;
43 __constant float EPS=0.001f;
44 __constant float MAXT=1e20f;
45 __constant float3 light1 = (float3)( 0.577f, 0.577f, 0.577f );
46 __constant float3 light2 = (float3)( -0.707f, 0, 0.707f );
47
48 inline bool iterate( const float3 q, float *resPot, float4 *resColor )
49 {
50     float4 trap = (float4)(100);
51     float3 zz = q;
52     float m = dot(zz,zz);
53     if( m > Bailout ) {
54         *resPot = 0.5f*native_log(m); // /pow(8.0f,0.0f);
55         *resColor = (float4)(1);
56         return false;
57     }
58
59 #pragma unroll 4
60     for( int i=0; i<NumIte; i++ ) {
61         const float x = zz.x; const float x2 = x*x; const float x4 = x2*x2;
62         const float y = zz.y; const float y2 = y*y; const float y4 = y2*y2;
63         const float z = zz.z; const float z2 = z*z; const float z4 = z2*z2;
64
65         const float k3 = x2 + z2;
66         const float k2 = rsqrt( k3*k3*k3*k3*k3*k3*k3 );
```



```

67     const float k1 = x4 + y4 + z4 - 6*y2*z2 - 6*x2*y2 + 2*z2*x2;
68     const float k4 = x2 - y2 + z2;
69
70     zz.x = q.x + 64*x*y*z*(x2-z2)*k4*(x4-6.0*x2*z2+z4)*k1*k2;
71     zz.y = q.y + -16*y2*k3*k4*k4 + k1*k1;
72     zz.z = q.z + -8*y*k4*(x4*x4 - 28*x4*x2*z2 + 70*x4*z4 - 28*x2*z2*z4 + z4*z4)*k1*k2;
73
74     m = dot(zz,zz);
75
76     trap = min( trap, (float3)(zz.xyz*zz.xyz,m) );
77
78     if( m > Bailout ) {
79         *resColor = trap;
80         *resPot = 0.5f*native_log(m)/native_powr(8.0f,i);
81         return false;
82     }
83 }
84
85 *resColor = trap;
86 *resPot = 0;
87 return true;
88 }
89
90 inline bool ifractal( __local Ray *ray) {
91     __local Sphere *sph=&ray->sph;
92     sph->origin = (float3)( 0);
93     sph->r = 1.25f;
94
95     // bounding sphere
96     if( !isphere(ray) )
97         return false;
98
99     // early skip
100    if( sph->dis.y<EPS ) return false;
101
102    // clip to near!
103    if( sph->dis.x<EPS ) sph->dis.x = EPS;
104
105    if( sph->dis.y>MAXT) sph->dis.y = MAXT;
106
107    float dt;
108    float3 gra;
109    float4 color, col2;
110    float pot1, pot2, pot3, pot4;
111
112    // raymarch!
113    float t=sph->dis.x, Surface, eps;
114    float3 p = ray->origin + ray->dir * t;
115
116    while(t < sph->dis.y) {
117        if( iterate(p,&pot1,&color) ) {
118            ray->t = t;
119            ray->nor = fast_normalize(gra);
120            ray->col = color;
121            return true;
122        }
123
124        Surface = clamp( EPS*t*ray->fovfactor, 0.000001f, 0.005f );
125        eps = Surface*0.1f;
126
127        iterate(p+(float3)(eps,0.0,0.0),&pot2,&col2);
128        iterate(p+(float3)(0.0,eps,0.0),&pot3,&col2);
129        iterate(p+(float3)(0.0,0.0,eps),&pot4,&col2);
130
131        gra = (float3)( pot2-pot1, pot3-pot1, pot4-pot1 );
132        dt = 0.5f*pot1*eps/fast_length(gra);
133
134        if( dt<Surface ) {
135            ray->col = color;
136            ray->nor = fast_normalize( gra );
137            ray->t = t;
138            return true;
139        }
140
141        t += dt;

```



```

142     p += ray->dir * dt;
143 }
144
145     return false;
146 }
147
148     __kernel
149 void compute(__write_only image2d_t pix, const float time) {
150     const int x = get_global_id(0);
151     const int y = get_global_id(1);
152     const int xl = get_local_id(0);
153     const int yl = get_local_id(1);
154     const int tid = xl+yl*get_local_size(0);
155     const int width = get_global_size(0)-1;
156     const int height = get_global_size(1)-1;
157
158     const float2 resolution = (float2)(width,height);
159     const float2 gl_FragCoord = (float2)(x,y);
160
161     const float2 p = (float2)(-1.f + 2.f * gl_FragCoord / resolution);
162     const float2 s = p*(float2)(1.33,1.0);
163
164     const float fov = 0.5f, fovfactor = rsqrt(1+fov*fov);
165
166     const float ct= native_cos(2*M_PI_F*time/20.f), st= native_sin(2*M_PI_F*time/20.f);
167     const float r = 1.4f+0.2f*ct;
168     const float3 campos = (float3)( r*st, 0.3f-0.4f*st, r*ct );
169     const float3 camtar = (float3)(0,0.1,0);
170
171     //camera matrix
172     const float3 cw = fast_normalize(camtar-campos);
173     const float3 cp = (float3)(0,1,0);
174     const float3 cu = fast_normalize(cross(cw,cp));
175     const float3 cv = fast_normalize(cross(cu,cw));
176
177     // ray
178     __local Ray rays[WARPSIZE+1],*ray=rays+tid;
179     ray->origin=campos;
180     ray->dir = fast_normalize( s.x*cu + s.y*cv + 1.5f*cw );
181     ray->fovfactor = fovfactor;
182
183     barrier(CLK_LOCAL_MEM_FENCE);
184
185     const bool res=ifractal(ray);
186
187     if( !res ) {
188         // background color
189         ray->rgb = 1.3f*(float3)(1,0.98,0.9)*(0.7f+0.3f*ray->dir.y);
190     }
191     else {
192         // intersection point
193         const float3 xyz = ray->origin + ray->t * ray->dir;
194
195         // sun light
196         float dif1 = clamp( 0.2f + 0.8f*dot( light1, ray->nor ), 0.f, 1.f );
197         dif1=dif1*dif1;
198
199         // back light
200         const float dif2 = clamp( 0.3f + 0.7f*dot( light2, ray->nor ), 0.f, 1.f );
201
202         // ambient occlusion
203         const float aot = clamp(1.25f*ray->col.w-.4f, 0.f, 1.f);
204         const float ao=0.5f*aot*(aot+1);
205
206         // shadow: cast a lightray from intersection point
207         if( dif1 > EPS ) {
208             __local Ray *lray=rays+256;
209             lray->origin=xyz;
210             lray->dir=light1;
211             lray->fovfactor = fovfactor;
212             if( ifractal(lray) )
213                 dif1 = 0.1f;
214         }
215
216         // material color

```



```

217 ray->rgb = (float3)(1);
218 ray->rgb = mix( ray->rgb, (float3)(0.8,0.6,0.2), (float3)(native_sqrt(ray->col.x)*1.25f) );
219 ray->rgb = mix( ray->rgb, (float3)(0.8,0.3,0.3), (float3)(native_sqrt(ray->col.y)*1.25f) );
220 ray->rgb = mix( ray->rgb, (float3)(0.7,0.4,0.3), (float3)(native_sqrt(ray->col.z)*1.25f) );
221
222 // lighting
223 ray->rgb *= (0.5f+0.5f * ray->nor.y)*
224 (float3)(.14,.15,.16)*0.8f +
225 dif1*(float3)(1.0,.85,.4) +
226 0.5f*dif2*(float3)(.08,.10,.14);
227 ray->rgb *= (float3)( native_powr(ao,0.8f), native_powr(ao,1.0f), native_powr(ao,1.1f) );
228
229 // gamma
230 ray->rgb = 1.5f*(ray->rgb*0.15f + 0.85f*native_sqrt(ray->rgb));
231 }
232
233 const float2 uv = 0.5f*(p+1.f);
234 ray->rgb *= 0.7f + 0.3f*16.f*uv.x*uv.y*(1.f-uv.x)*(1.f-uv.y);
235
236 ray->rgb = clamp( ray->rgb, (float3)(0), (float3)(1) );
237
238 write_imagef(pix,(int2)(x,y),(float4)(ray->rgb,1.0f));
239 }

```

Appendix B OpenCL and CUDA terminology

NVidia provides CUDA, an older API than OpenCL very used on their devices. CUDA and WebCL/OpenCL share similar concepts but a different terminology that we give below, borrowed from AMD article [30] and adapted to WebCL.

Terminology

WebCL/OpenCL	CUDA
Compute Unit (CU)	Streaming Multiprocessor (SM)
Processing Element (PE)	Streaming Processor (SP)
Work-item	Thread
Work-group	Thread block
Global memory	Global memory
Constant memory	Constant memory
Local memory	Shared memory
Private memory	Local memory

Writing kernels: qualifiers

WebCL/OpenCL	CUDA
__kernel function	__global__ function
(no annotation necessary)	__device__ function
__constant variable	__constant__ variable
__global variable	__device__ variable
__local variable	__shared__ variable

Writing kernels: indexing

WebCL/OpenCL	CUDA
get_num_groups()	gridDim
get_local_size()	blockDim
get_group_id()	blockIdx
get_local_id()	threadIdx
get_global_id()	No direct equivalent. Combine blockDim,



	blockIdx, and threadIdx to get a global index.
get_global_size()	No direct equivalent. Combine gridDim and blockDim to get the global size

Writing kernels: synchronization

WebCL/OpenCL	CUDA
barrier()	__syncthreads()
No equivalent	__threadfence()
mem_fence(CLK_GLOBAL_MEM_FENCE CLK_LOCAL_MEM_FENCE0)	__threadfence_block()
read_mem_fence()	No equivalent
write_mem_fence()	No equivalent

Important API objects

WebCL/OpenCL	CUDA
WebCLDevice	CUdevice
WebCLContext	CUcontext
WebCLProgram	CUmodule
WebCLKernel	CUfunction
WebCLMemoryObject	CUdeviceptr
WebCLCommandQueue	No equivalent

Important API calls

WebCL/OpenCL	CUDA
No initialization required	cuInit()
WebCLContext.getInfo()	cuDeviceGet()
WebCLContext.create()	cuCtxCreate()
WebCLContext.createCommandQueue()	No equivalent
WebCLProgram.build()	No equivalent. CUDA programs are built off-line
WebCLContext.createKernel()	cuModuleGetFunction()
WebCLCommandQueue.enqueueWriteBuffer()	cuMemcpyHtoD()
WebCLCommandQueue.enqueueReadBuffer()	cuMemcpyDtoH()
Using locals of WebCLCommandQueue.enqueueNDRange()	cuFuncSetBlockShape()
WebCLKernel.setArg()	cuParamSet()
Using WebCLKernel.setArg()	cuParamSetSize()
WebCLCommandQueue.enqueueNDRangeKernel()	cuLaunchGrid()
Implicit through garbage collection	cuMemFree()



Bibliography

Specifications

- [1] Aarnio, T. and Bourges-Sevenier, M. WebCL Working Draft. *Khronos WebCL Working Group*. <https://cvs.khronos.org/svn/repos/registry/trunk/public/webcl/spec/latest/index.html>.
- [2] Munshi, A. OpenCL Specification 1.1. *Khronos OpenCL Working Group*. <http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf>.
- [3] Marrin, C. WebGL Specification. *Khronos WebGL Working Group*. <http://www.khronos.org/registry/webgl/specs/latest/>.
- [4] Munshi, A. and Leech, J. OpenGL ES 2.0.25. *Khronos Group*. http://www.khronos.org/registry/gles/specs/2.0/es_full_spec_2.0.25.pdf.
- [5] Simpson, R.J. The OpenGL ES Shading Language. *Khronos Group*. http://www.khronos.org/registry/gles/specs/2.0/GLSL_ES_Specification_1.0.17.pdf.
- [6] Herman, D. and Russell, K., eds. Typed Array Specification. *Khronos.org*. <http://www.khronos.org/registry/typedarray/specs/latest/>.
- [7] OpenCL 1.1 Reference Pages. OpenCL 1.1 Reference Pages. *Khronos.org*. <http://www.khronos.org/registry/cl/sdk/1.1/docs/man/xhtml/>.

Programming guides

- [8] NVidia OpenCL Programming Guide for the CUDA Architecture. 2012. *NVidia OpenCL Programming Guide for the CUDA Architecture*.
- [9] AMD Accelerated Parallel Processing OpenCL. 2011. *AMD Accelerated Parallel Processing OpenCL*.

Books

- [10] Gaster, B., Howes, L., Kaeli, D.R., Mistry, P., and Schaa, D. 2011. *Heterogeneous Computing with OpenCL*. Morgan Kaufmann.
- [11] Scarpino, M. 2011. *OpenCL in Action: How to Accelerate Graphics and Computations*. Manning Publications.
- [12] Munshi, A., Gaster, B., Mattson, T.G., Fung, J., and Ginsburg, D. 2011. *OpenCL Programming Guide*. Addison-Wesley Professional.
- [13] Kirk, D. and Hwu, W.-M. 2010. *Programming Massively Parallel Processors*. Morgan Kaufmann.
- [14] Hillis, W.D. and Steele, G. 1986. *Data parallel algorithms*.

WebCL prototypes

- [15] Motorola Mobility. Node-webcl, an implementation of Khronos WebCL specification using Node.JS. <https://github.com/Motorola-Mobility/node-webcl>
- [16] Nokia Research. WebCL. <http://webcl.nokiaresearch.com/>
- [17] Samsung Research. WebCL prototype for WebKit. <http://code.google.com/p/webcl/>
- [18] Mozilla. FireFox WebCL branch. <http://hg.mozilla.org/projects/webcl/>

Articles and Presentations

- [19] Cole, M.I. 1989. Algorithmic skeletons: structured management of parallel computation
- [20] Gerstmann, D. 2009. *Advanced OpenCL*. Siggraph 2009.
- [21] McCool, M.D. 2010. Structured parallel programming with deterministic patterns. *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, 5–5.
- [22] Bordoloi, U.D. 2010. Optimization Techniques: Image Convolution. 1–25. <http://developer.amd.com/zones/openclzone/events/assets/optimizations-imageconvolution1.pdf>.



- [23] BDT Nbody Tutorial. BDT Nbody Tutorial. *Brown Deer Technology*. http://www.browndeertechnology.com/docs/BDT_OpenCL_Tutorial_NBody-rev3.html.
- [24] Iñigo Quilez. ShaderToy with Mandelbulb shader. <http://www.iquilezles.org/apps/shadertoy/?p=mandelbulb>
- [25] Donnelly, W. GPU Gems - Chapter 8. Per-Pixel Displacement Mapping with Distance Functions. *developer.nvidia.com*. http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter08.html.
- [26] Mattson, T.G., Buck, I., Houston, M., and Gaster, B. 2009. OpenCL - A standard platform for programming heterogeneous parallel computers. *SC'09*. <http://www.crc.nd.edu/~rich/SC09/docs/tut149/OpenCL-tut-sc09.pdf>.
- [27] Feng, W.-C., Lin, H., Scogland, T., and Zhang, J. 2012. OpenCL and the 13 dwarfs: a work in progress.
- [28] Lefohn, A., Kniss, J., and Owens, J.D. Chapter 33. Implementing Efficient Parallel Data Structures on GPUs. In: *GPU Gems 2*. Addison-Wesley.
- [29] Krüger, J. and Westermann, R. Chapter 44. A GPU Framework for Solving Systems of Linear Equations. In: *GPU Gems 2*. Addison-Wesley.
- [30] Porting CUDA Applications to OpenCL. Porting CUDA Applications to OpenCL. *developer.amd.com*. <http://developer.amd.com/zones/OpenCLZone/programming/pages/portingcudatoopencl.aspx>.
- [31] Hensley, J., Gerstmann, D., and Harada, T. OpenCL by Example. *SIGGRAPH Asia 2010*.
- [32] A Pattern Language for Parallel Programming. A Pattern Language for Parallel Programming. *parlab.eecs.berkeley.edu*. <http://parlab.eecs.berkeley.edu/wiki/patterns/patterns>.

OpenCL™ and the OpenCL™ logo are trademarks of Apple Inc. used by permission by Khronos.