Contents

1 Introduction

Computer Vision implementations have benefited from the increased speed and parallelism of general purpose GPU technology such as Nvidia's CUDA. However, the overhead of distributing such applications has traditionally been high, requiring installation of executable files along with compatible hardware and libraries. Meanwhile, the focus of everyday computing has been shifting towards the web browser, which offers the advantage of instant delivery, but is generally considered inappropriate for real-time vision due to the inability to run compiled code unhindered. The introduction of JavaScript APIs giving access to the webcam and exposing the OpenGL ES graphics library through WebGL gives an opportunity to overcome this limitation. Although typically used for drawing 3D graphics, WebGL allows arbitrary parallel calculations to be offloaded onto the GPU through the use of Shader Programs. By these means we create a GPU accelerated face detector which runs in the browser, with applications in areas such as augmented reality, web games and video chat. We employ the traditional cascade structure for face detection, pioneered by Viola and Jones, but adapted to run on the GPU, and using a different type of feature. Our aim is to exceed the framerates offered by current JavaScript-only implementations. We give a guide to the main concepts required to work with general purpose computation with WebGL, explain an initial implementation of face detection in JavaScript, then show how it can be adapted to WebGL, and then how it can be optimised for speed. We also show how this face detector can be used for a practical application of tracking the user's head, allowing head motion to translate to motion of the camera in a scene such that it appears like looking through a window, allowing 3D scenes to be observed in a natural way.

2 Background and Concepts

2.1 Face Detection

The seminal work on face detection is the Viola & Jones (2001) paper, which for the first time gave reliable detection while maintaining a speed capable of real- time use. These advances were due to a combination of different techniques. Firstly, the use of the "integral image" in order to greatly reduce the number of lookups to find the sum intensity of a region in an image, requiring only four reads for an area of any size. An integral image is essentially a cumulative map of the pixel intensities, such that the value at any position tells the sum of those pixels to the top left of it (Figure 1). By reading the four corners of a rectangle in an integral image, we can cancel out the area to the top and left, leaving only the sum of the area inside the rectangle (Figure 2). This makes it easy to compare the relative intensities of areas in images, which is fundamental to the features chosen by Viola and Jones, the Haar features (so named due to similarity with Haar Wavelets). Haar features compare the difference in brightness between regions of rectangles, in an attempt to pick up contrast variation. The feature value is the white region minus the shaded region of the rectangles in the figure. A 24×24 window is typically used for face detection, and during detection the values of Haar features at specific positions and sizes within that window (determined in the learning stage) need to be computed.

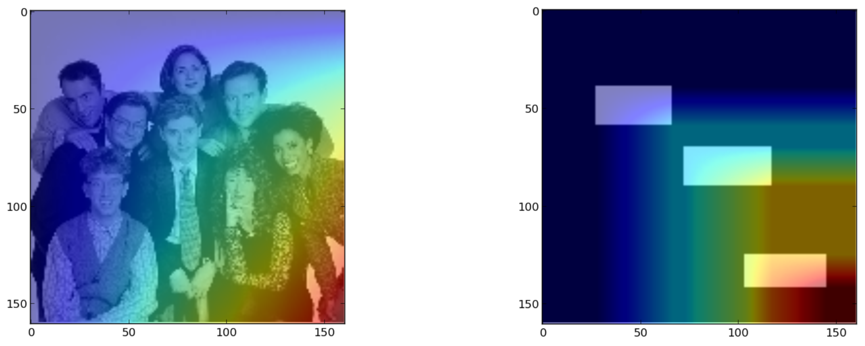


Figure 1: Visualisation of the integral image for two different pictures, as a colour map of the integral value superimposed on the original image. The value of the integral image at any location is the sum of all the pixels to the top left of it. This is apparent in the right image, where we have several white rectangles (pixel value 255) on a black background (pixel value 0), causing sharper changes.

Accounting for all variations in position and size (x,y,w,h) within the window there are many thousands of possible features. In order to pick a small number that are good for classifying, the method of Boosting is used (specifically AdaBoost), which allows multiple weak classifiers to be combined into one strong classifier. A weight is associated with each training sample, initially all equal, and the feature which best discriminates between the positive and negative samples is chosen. The weights are then updated such that missclassified samples using this feature are given a higher weight, and correct samples a lower one (so more effort goes towards "fixing" the missclassifications), then the feature selection is repeated with the new weights. In the end we get an ensemble of features that combined have a lower error.

However, with the above, during detection all the chosen features will need to be computed for all the window positions. Viola and Jones noticed that this could be improved by having a cascade of different boosted classifiers, which are decent at rejecting non-faces but will rarely reject a true face, chaining these classifiers together such that a rejected window will be immediately discarded, but an accepted window will be passed on to the next level in the cascade to face further scrutiny (Figure 5. This means that normally only the windows with true faces need go through every single level, whereas a non-face may be

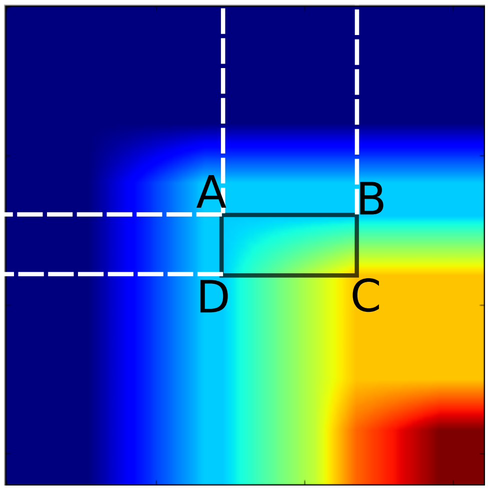


Figure 2: Finding the sum of intensities of a region with the integral image, using the second picture of the previous figure. Each point gives the sum of the pixel values to the top left. If we take point C then we want to “cancel out” the excess area outside the rectangle. We can subtract B and D, but then we have taken too much, since the area top-left of B and D overlap. Therefore we have to add A to get the right value. The sum of the area in the rectangle is then C-D-B+A.

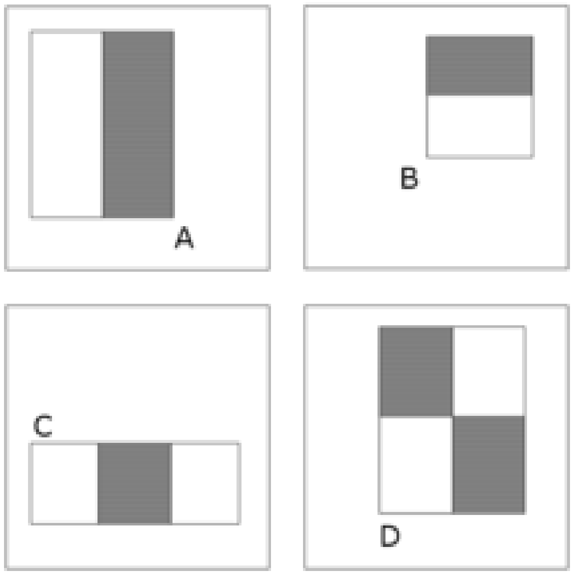
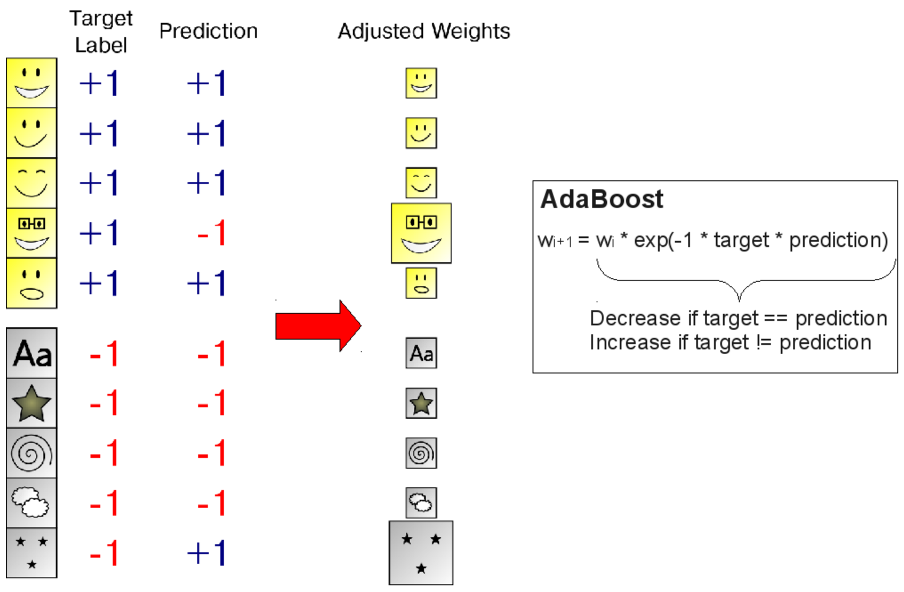


Figure 3: Types of Haar Features (within larger windows) (Wikipedia/Public Domain)



Target Label
Prediction
Adjusted Weights
+1
+1
+1
+1
+1
+1
+1
-1
+1
+1
-1
-1
-1
-1
-1
-1
-1
-1
-1
+1
Aa
AdaBoost
w_{i+1} = w_i * exp(-1 * target * prediction)
Decrease if target == prediction
Increase if target != prediction

Figure 4: An illustration of Adaptive Boosting (AdaBoost), and the weight update step. The weight of misclassified elements is "boosted", so that they will be considered more important on the next step. ((Thewlis, 2012))

rejected right from the start. This avoids examining every single chosen feature for every single window, and makes sense intuitively since the vast majority of windows in most images will not have a face, so it is beneficial to reject them early, speeding up the procedure.

A widely used implementation of Viola Jones face detection is the OpenCV library, which provides several pre-trained cascades in an XML serialisation format. Many of the javascript face detection solutions are based on OpenCV's code.

Lienhart & Maydt (2002) propose an extra set of "tilted" Haar features at 45 degrees, computed thanks to a rotated integral image, in order to better represent distinctive characteristics such as slanted edges which would otherwise be missed.

Besides Haar features, another mechanism that can be used is Local Binary Patterns (LBP). Originally used for pattern description by Ojala, Pietikhenl & Harwoodet al. (1994), the basic LBP simply describes the neighbourhood of a pixel in terms of whether the 8 surrounding pixels are darker or lighter than the central one, and going clockwise from the top, can be represented as an 8 bit number by writing 1 if the center is lighter, 0 otherwise.

Zhang, Chu & Xianget al. (2007) extended LBP to a multi-block representation, dividing a rectangle into 3x3 blocks and comparing the outer ones with the centre, for use with face detection. Properties of LBP are that it is more robust

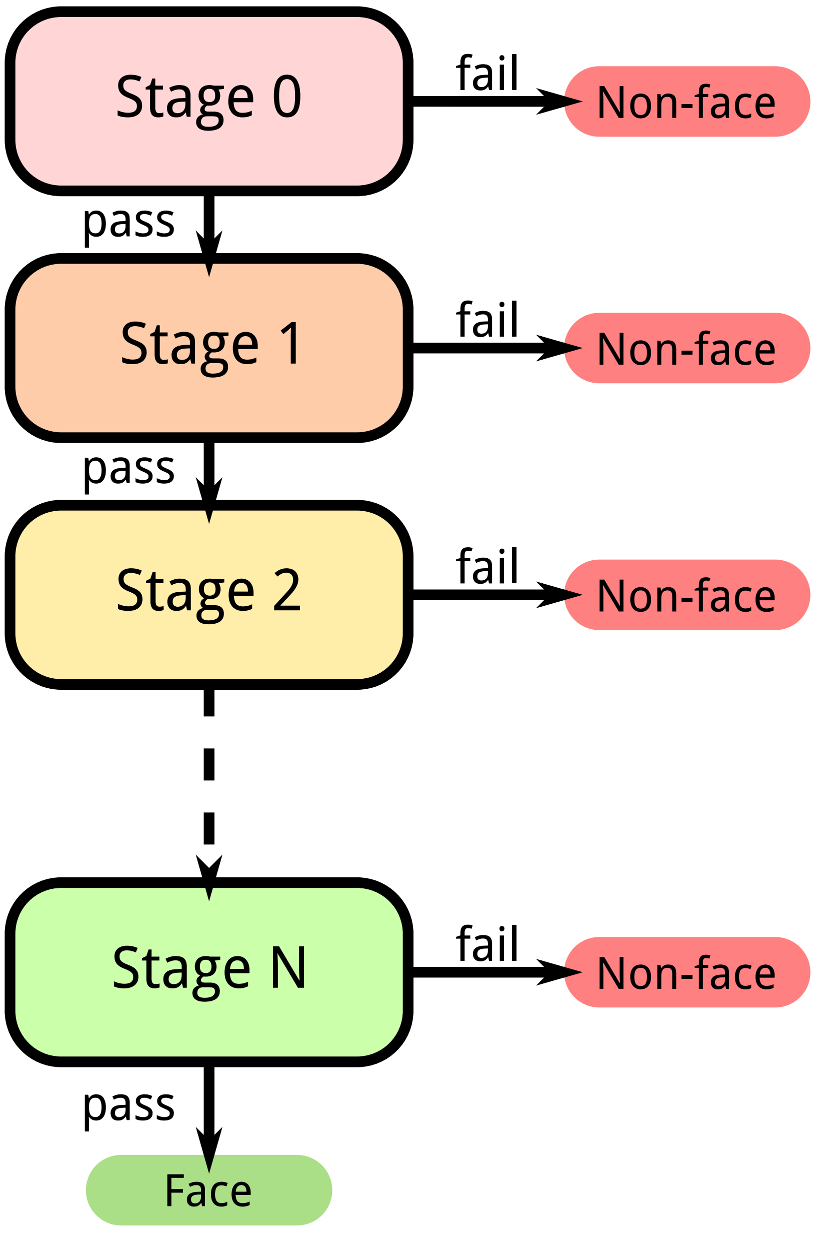


Figure 5: The Cascade Structure. Suspected non-faces are rejected immediately, potential faces go on to further stages

to illumination variation, since it only records the lighter/darker relationship rather than a quantitative amount like with Haar. The ability to represent the configuration of an LBP rectangle in just one byte also makes it space efficient, especially useful when GPU textures are involved, since one could pack four patterns in one "pixel".

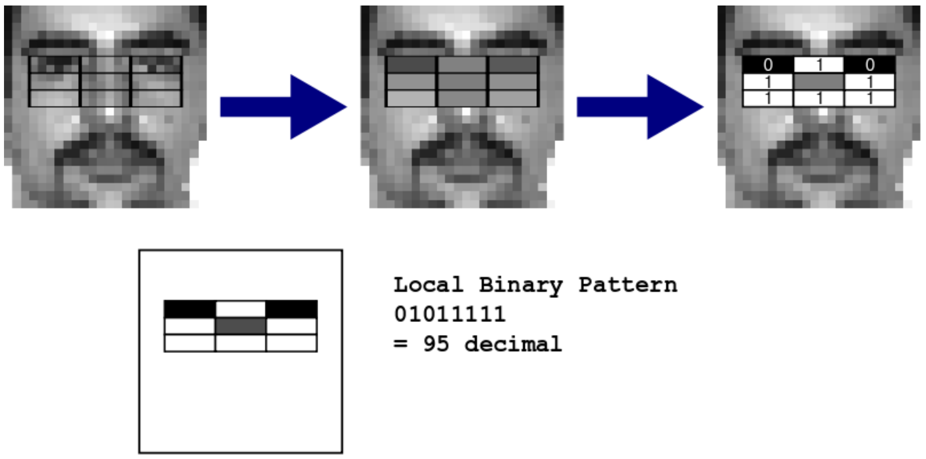


Figure 6: Local Binary Patterns for face detection

OpenCV also includes an LBP face detection implementation, along with a pre-trained XML file specifying a cascade for detecting frontal faces. It differs from the Haar cascades in that, for each weak classifier in a stage, rather than a simple threshold there is a list of the possible patterns (0 to 255) that contribute either positively or negatively towards a candidate window being a face, represented internally as a bit vector of size 256, made up of 8 32 bit integers. A diagram is shown in Figure 7.

2.2 Vision in the Browser

The desire to integrate Computer Vision with the web has some history. Existing approaches largely make use of custom browser plugins able to run native code, such as the face detection used in Google Hangout Google (n.d.). This has the disadvantage of requiring the user to trust and install the plugin in question, or may rely on browser-specific technology such as Microsoft's ActiveX or Google's Native Client. Adobe Flash has proved another contender Molchanov (n.d.), being a commonly installed plugin able to provide access to the webcam, but its bytecode-based VM is slower than native code, and its popularity is waning due to incompatible mobile devices and the introduction of comparable features in the HTML5 specifications.

JavaScript, the de-facto language of the web, has been applied to certain vision tasks with some success. Despite the disadvantage of being an interpreted

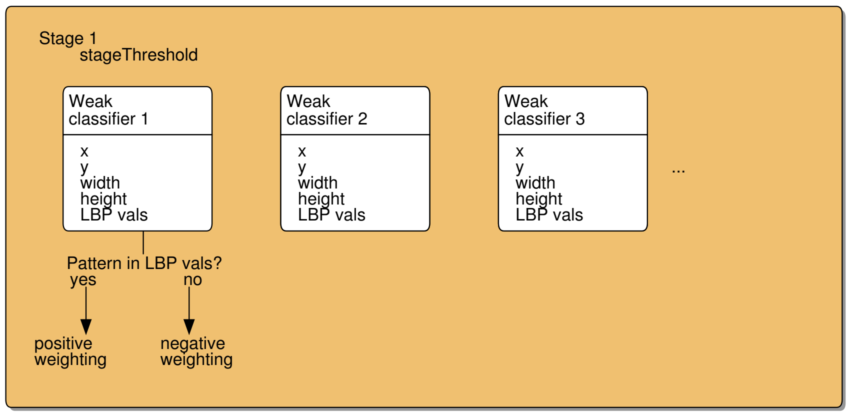


Figure 7: A look inside one of the stages of a local binary pattern classifier. Each weak classifier has an associated pattern, given by the rectangle (x,y,width,height), which determines the region in the window where the pattern is found. There is then a list of which LBP values should be weighted positive, and which negative. The weightings are summed for each weak classifier, and if they surpass the threshold for the stage, the window is accepted.

language, recent efforts towards speeding up JavaScript engines, such as Google Chrome's V8, have led to great improvements through techniques such as JIT ("Just In Time") compilation. Another leap forward in making JavaScript suitable for Vision was the "getUserMedia" API, introduced in 2011, giving JavaScript direct access to the user's webcam (upon consent). There are existing implementations of single-image face detection in JavaScript, such as Liu (n.d.), and there is the more amusing cat face detector Arthur (n.d.). Face detection on video is also possible, but typically employs techniques such as downsizing the video or skipping frames. Upon embarking on the present project, the author could locate no similar such endeavour to implement a Vision suite in JavaScript, but fate being as it is, a similar project by the name of "jsfeat" Zatepyakin (n.d.) appeared on the scene within a couple of weeks. The author chooses to take this as proof of current demand for vision on the web. Since jsfeat does not include GPU acceleration, it will serve as a useful baseline. In demonstrations on the web, when set to use the full range of scales, it can detect faces at around 27 fps, although it is working on input images of 160x120 resolution, and then using a "step" so that it skips every other pixel at the finest scale, and this step increases for larger scales.

The effort to give JavaScript standards-based access to computational and graphical libraries that can take advantage of dedicated hardware was pioneered by the Khronos group, maintainers of the OpenGL and OpenCL specifications.

These libraries are commonly used in native programs for 3D graphics and parallel computation respectively.

In 2009 Khronos began to draft a specification for WebGL, which would give JavaScript 3D drawing capabilities, through an extended context of the “canvas” element which was introduced in version 5 of the HTML specification. WebGL is based on the OpenGL ES 2.0 specification, which itself is a cut down version of OpenGL initially designed for the benefit of mobile devices, lacking OpenGL’s deprecated fixed rendering pipeline (which has built in support for lighting and perspective transforms) to give a more lightweight library, leaving it to the developer to explicitly specify vertex transformations and texture values needed to render a scene. WebGL’s functions and behaviour are largely identical to OpenGL ES 2.0, so resources and documentation are often applicable to both. Some additional considerations are needed for the browser-based host such as security restrictions on image access and support for web-specific data types like HTMLVideoElement for grabbing texture from a video (eg. from a webcam) on the page. The specification for WebGL 1.0 was released in 2011 and experimental support is present in the latest versions of Google Chrome and Mozilla Firefox. Safari and Opera also support WebGL although it is currently disabled by default. Microsoft’s Internet Explorer does not support it, unless third party plugins are used.

Following WebGL, work was started on a WebCL specification, providing JavaScript bindings to the OpenCL parallel computation library, to allow code to be sped up using hardware on the GPU or multi-core CPU, a use case being physics calculations in games. Although it is still in draft form, implementations have been released by Nokia and Samsung in the form of browser plugins. WebCL would certainly be a strong candidate for Vision related tasks in the browser, especially given the wide use of OpenCL and Nvidia’s CUDA for implementing high performance Vision systems. However, it is not yet implemented natively in any browser, and it remains to be seen whether it will be adopted by browser manufacturers. Since the goal of this project is to implement Computer Vision in current browsers with no extra steps required on behalf of the user, at the moment WebCL is unfortunately not an option, and its cousin WebGL is the only viable choice for running code on the GPU from a browser.

Despite being intended for drawing 3D graphics, OpenGL and hence WebGL is quite capable for performing arbitrary calculations, due to the use of shaders, which are programs that run on the GPU for the purpose of modifying the geometry and colour of a scene. Using OpenGL for arbitrary computation seems to have become quite fashionable around 2005, as evidenced by the number of pages dedicated to it in the GPU Gems 2 book, but then seems to have fallen out of favour after NVIDIA released CUDA in 2007, giving a more convenient purpose-built framework for computation in the GPU. Nevertheless, with the rise of OpenGL ES in smartphones and WebGL in the browser, using GL for computation remains attractive. In OpenGL there are two types of shader, vertex shaders, which specify the geometrical position in 3D given an array of vertex points and vertex-specific attributes, and fragment shaders, which specify the colour (RGB and alpha transparency) of the output pixels after rasterisation. Shaders in OpenGL are written in GLSL (The OpenGL Shader Language) a C-like language which offers many of the conveniences found in computation-oriented GPU platforms such as CUDA. An advantage of handling computation in shaders is that they operate in a parallel, computationally independent manner on many vertices or pixels, essentially the SIMD (Single Instruction Multiple Data) paradigm. This offers a large advantage over sequential computation on the CPU, provided the algorithm can be structured in such a way as to take advantage of massive parallelisation over many data elements, as is often the case with image processing algorithms where a certain operation is desired to be performed on every pixel. The fragment shader is where the main potential for parallelisation over data lies, since it can look up data in textures, run procedures, and output a value for every pixel in the canvas. However, computation in the vertex shader can also be valuable, since it precedes the fragment shader in the OpenGL pipeline and is able to pass data to it in the form of "varyings", which are vertex-specific variables then interpolated across the rasterised surface. This gives the possibility to precalculate information in the vertex shader that will be used by many pixels.

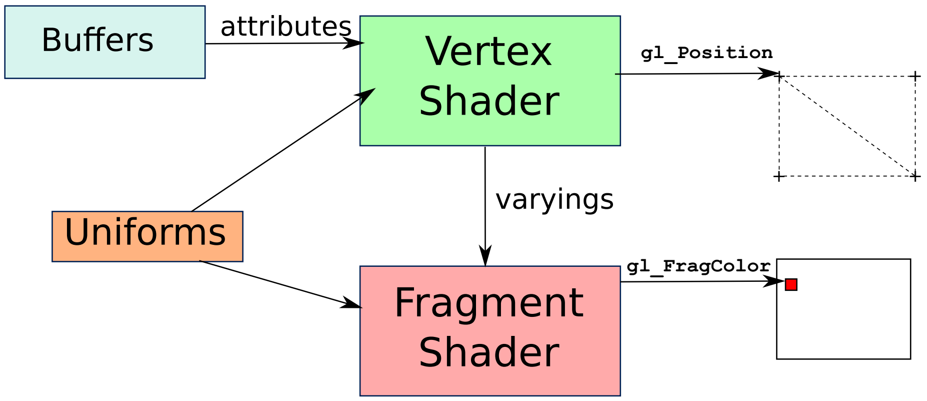


Figure 8: The WebGL pipeline

So far we have been talking about pixels and 3D vertices, which are not very useful if we want to work with 2D images or arbitrary numerical data. The trick to getting a simple 2D surface is to render a viewport-aligned rectangle (in practice, by drawing two triangles). As Tavares (2011b) describes, WebGL is essentially a 2D API if we want it to be, and we can work in pixels rather than OpenGL's "clipspace" units (which are -1.0 to 1.0) by normalising by the canvas resolution. As for passing data to the shaders, a limited amount of floating point or integer variables and arrays may be passed as "uniforms" to the shaders (which remain constant for each vertex or pixel), but for handling large data such as images and arrays textures must be used. Instead of drawing to the screen, the output may be rendered to a framebuffer with a texture attached, allowing it to be used as an input texture for another stage. Each pixel in a normal texture is 4 bytes (RGBA), which is suitable for representing some numerical data, but in many cases it is preferable to use floating points. This can be done using the WebGL extension `OES_texture_float`, which is supported on most platforms. However, until recently, there was no defined way to read back floating point values to JavaScript, requiring inventive solutions such as packing floats into bytes Consortium (n.d.). The latest draft specifications of `EXT_color_buffer_half_float` Khronos (2012b) and `WEBGL_color_buffer_float` Khronos (2012a), amend the readPixels() function to permit float types, however it will be a matter of time before all browsers support it. In addition, when doing calculations, care should be given to the precision qualifiers offered by GLSL (highp, mediump, lowp) which alter the precision of numerical representations, at a speed/accuracy tradeoff. Per the GLSL specification, highp (normally IEEE float) may not be available in the fragment shader, and integers may be implemented as floats in the hardware, which should be taken into account when implementing algorithms.

There are various examples on the web using WebGL to improve the efficiency of calculations in simulations, such as WebGL Water Wallace (n.d.) which calculates the water and caustics simulation in the shaders,

In addition, the trick of using of OpenGL Shaders to increase performance is employed by the GPUImage library Larson (n.d.) for image processing on iOS devices.

2.3 Computation in WebGL - concepts and example

We shall provide a gentle introduction to the concepts behind using WebGL to perform general purpose computation, walking the reader through the main steps required to implement a simple convolution shader which operates on images from the webcam. Although this is a somewhat simple image processing task, it introduces many of the techniques that will be essential for face detection. Indeed, the act of processing a stage of the face detection cascade within a window can be viewed as a sort of glorified convolution, since for a certain pixel location it consists of looking up the values within some surrounding neighbourhood, using them to determine what result to output.

We lean on the work of Tavares (2011c) in using WebGL for image processing, and the explanations of the WebGL API by Tavares (2012) and Zhenyao Mo (2012). We draw on Harris (2005) in order to explain computational concepts in terms of OpenGL, in particular we make use of the analogies presented between CPU techniques and their GPU counterparts. We shall also make reference to a JavaScript library, called WebCV, created by the author for the purpose of abstracting away some of the complications of WebGL, providing utility functions which facilitate tasks commonly needed for general purpose computation and computer vision applications.

Before we can start using WebGL, we must first make some preparations. WebGL is exposed through a special context of the HTML `<canvas>` element, so we insert a canvas in our document, specifying the width and height. To access the webcam, we also require a `<video>` element, which we set to the same dimensions as the canvas. The body of our HTML document then looks as follows.

```
<body>
<canvas id="glcanvas" width="400" height="300"></canvas>
<video id="webcamvideo" autoplay width="400" height="300"></video>
</body>
```

We must now use JavaScript to give some life to these elements. We want to initialise the WebGL context of our canvas, and make it so our video receives input from the webcam. For the former we can instantiate our WebCV library using `WebCV.create(canvas)`, passing in our canvas, which returns an object (which we shall call `cv`) through which we can access our utility functions. The bare WebGL API is also available through `cv.gl`. This level of indirection allows us to instantiate multiple copies of WebCV on different canvas elements, each with their own WebGL context. To receive video from the webcam we use the browser's `getUserMedia` API, for which we have a wrapper available in `cv.utils.getUserMedia` to abstract away from the browser- specific differences. This will cause a prompt to appear to the user, asking for permission to use their webcam, and call a specified function upon success. Within this function we then set the `src` of the `<video>` element, which causes it to start streaming video from the webcam.

```
// Get our canvas object
var canvas = document.getElementById("glcanvas");

// Initialise WebCV, which in turn initialises WebGL
// We make cv and gl global variables
cv = WebCV.create(canvas);
gl = cv.gl;

// Ask for the webcam
cv.utils.getUserMedia({video: true},
                      function(stream) {
                          var vid = document.getElementById("webcamvideo");
                          vid.src = stream;
                      },
                      function () { alert("Couldn't get webcam"); });
```

We now look at the roles of shader programs in WebGL. We have two types of shaders, the vertex shader and the fragment shader. If we were using WebGL to draw 3D geometry, the vertex shader would answer the question "Where should my vertices be placed?", by setting the value of `gl_Position`, and the fragment shader would answer the question "What colour should my fragments (pixels) be?", by setting `gl_FragColor`.

Recall we have three special types of variables in our shader programs. Uniform variables can be treated like global variables. They can be set from JavaScript, and are accessible from both the vertex and the fragment shader. They are the most straightforward way of setting variables which we use in the shader, and are used for values that do not vary according to the position in the image. For example, to convolve the image with a 3×3 matrix as the kernel, we would use a Uniform variable to hold the matrix, since we need the same matrix for every pixel, but can change the matrix for different uses of the shader program.

Attribute variables are used in the vertex shader for accessing vertex- specific data. They are specified by uploading arrays to buffers on the GPU. A primary use of attribute variables is to set the *vertex coordinates* used to determine the position of vertices, and the *texture coordinates* which define a mapping of vertices to positions within texture. Following Tavares (2011b), to draw a rectangle in 2D we need to draw two view- aligned triangles, as in Figure 9 (since the triangle is the only polygon primitive available in OpenGL ES and WebGL). This gives six coordinates for the vertices, each with two elements (x, y), so we would have to upload a buffer of 12 elements.

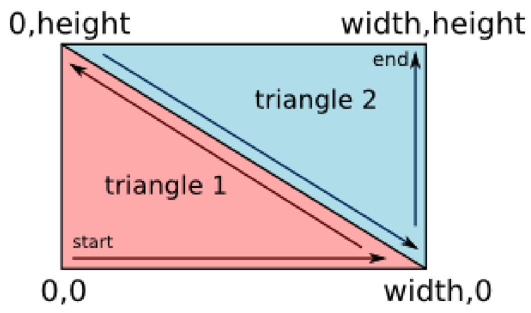
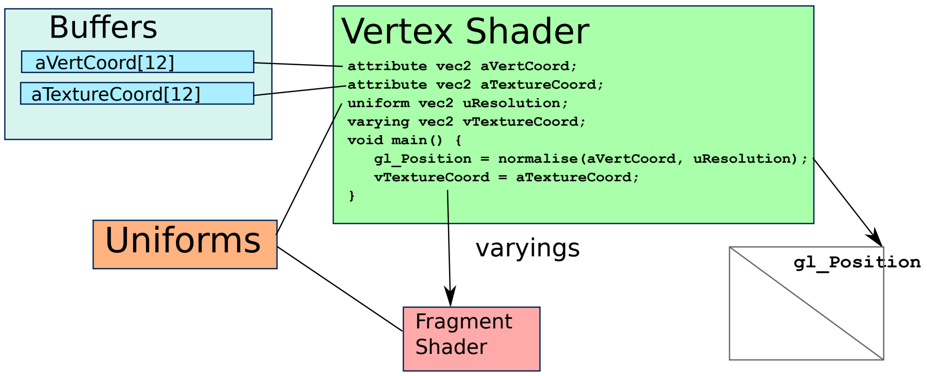


Figure 9: Drawing a quad using two triangles, showing the order of the vertices are specified.

The analogy given by Harris (2005, p.502) for the *vertex coordinates* is that they specify the *computational range*, by determining which pixels will be generated. We typically only wish to draw a rectangle (2 triangles), for which the task of the vertex shader is fairly simple, we just want to pass through the vertex coordinates, as if we were drawing pixel positions directly. A schematic of this simple vertex shader is shown in Figure 10, and it can be used whenever we just want to draw things in 2D and forget we are dealing with a 3D graphics API. Its only job is to draw the 2D geometry specified by our vertex coordinates, as if by orthogonal projection, and pass our texture coordinates to the fragment shader. The details of the normalisation of the input vertex coordinates in pixels to output vertex positions have been omitted, and require some understanding with OpenGL's coordinate systems, explained in the appendix.



```
attribute vec2 aVertCoord;
attribute vec2 aTextureCoord;
uniform vec2 uResolution;
varying vec2 vTextureCoord;
void main() {
   gl_Position = normalise(aVertCoord, uResolution);
   vTextureCoord = aTextureCoord;
}
```

Figure 10: A simple vertex shader to draw in 2D

While the *vertex coordinates* represent the *computational range*, the *texture coordinates* can be considered our *computational domain.* Texture coordinates in WebGL are 2-element floating point vectors varying from 0.0 to 1.0 in each dimension, and similarly to the vertex coordinates are passed as an attribute to the vertex shader using a buffer. However it is the fragment shader which needs to access the texture, not the vertex shader. This is where the third special type of variable, the varying, comes in to play. Varying variables output from the vertex shader at each vertex are linearly interpolated across the fragments between the vertices (this can be seen graphically in Figure 12). So while we only specify the texture coordinates at each vertex, by using varyings the texture coordinates passed to the fragment shader can span the entire texture. In many cases we will want the computational domain and range to be equal, in which case we can use a texture the same size as our output drawbuffer. However texture coordinates give us the flexibility to have a domain and range of different sizes, for example we may do some data minification that consumes two pixels in order to output a result. In the case that the domain and range are equal (or the domain can be expressed as offsets of the fragment position), we can avoid using texture coordinates through the use of the special `gl_FragCoord` variable in the fragment shader, which contains the screen- space position of the current fragment (already offset by 0.5 to give the centre of the fragment). We can then divide this by the texture size to get a 0.0-1.0 texture coordinate. An alternative way to modify the computational range is to use the `gl.viewport` and `gl.scissor` comands. The first modifies the size of the viewport, and the second lets us specify a box such that only the pixels within the box are written.

We now come to the main powerhouse in our inventory of WebGL tools, the fragment shader itself. The fragment shader is useful because it runs for every pixel (or fragment), thus making it amenable to parallel computation on a grid, where the output at one pixel does not require any intermediate information computed by its neighbours. The analogy with the CPU given by Harris (2005) is that fragment shader programs can be treated like the inner loops when iterating

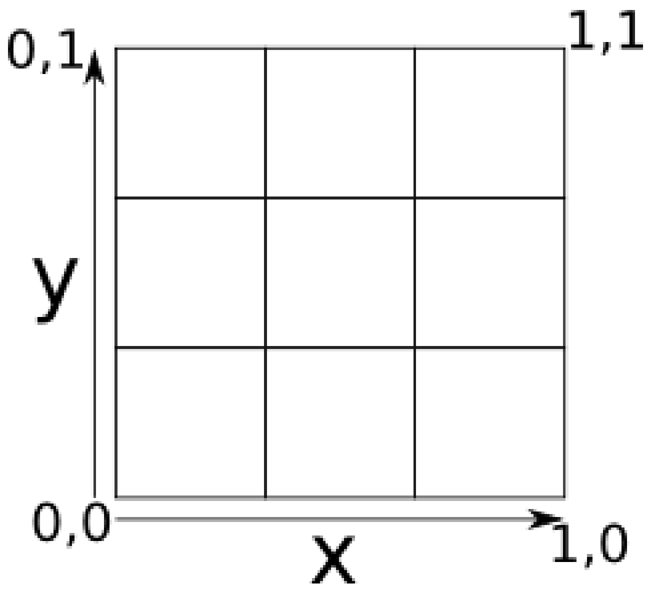


Figure 11: A texture of 3x3 texels. Texture coordinates in OpenGL vary from 0,0 to 1,1 regardless of size.

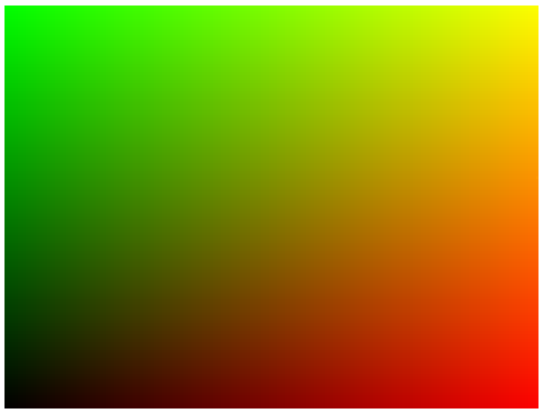
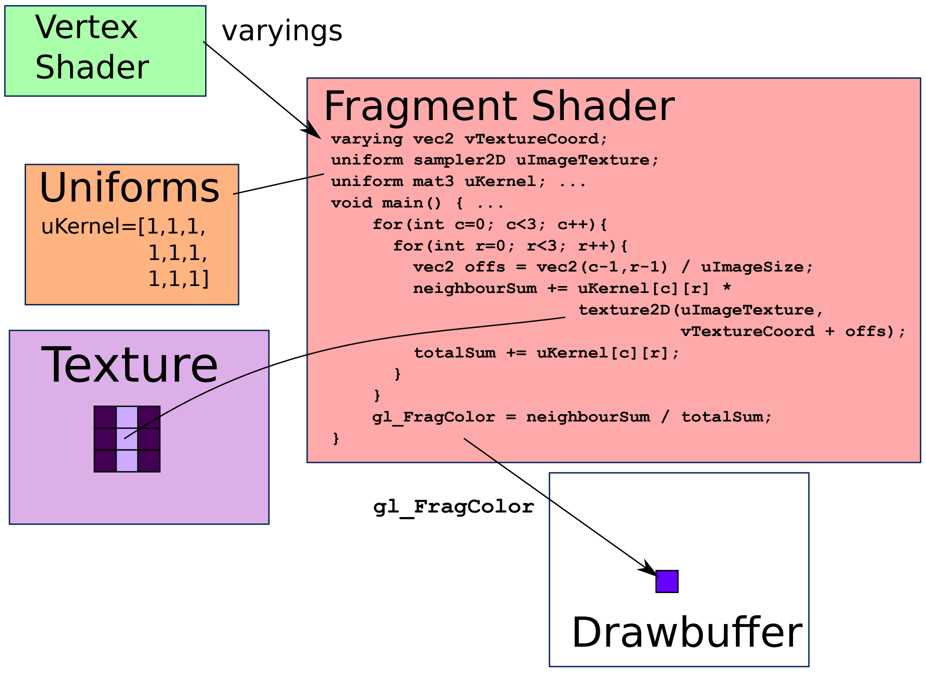


Figure 12: Interpolation of texture coordinates, shown by outputting the x and y texture coordinate in the red and green colour channel, such that we get a gradient that varies linearly in each channel



```
varying vec2 vTextureCoord;
uniform sampler2D uImageTexture;
uniform mat3 uKernel; ...
void main() { ...
    for(int c=0; c<3; c++){
      for(int r=0; r<3; r++){
        vec2 offs = vec2(c-1,r-1) / uImageSize;
        neighbourSum += uKernel[c][r] *
                        texture2D(uImageTexture,
                                  vTextureCoord + offs);
        totalSum += uKernel[c][r];
      }
    }
    gl_FragColor = neighbourSum / totalSum;
}
```

Figure 13: Fragment shader to process our convolution kernel. The example convolution kernel given would simply compute the average of the nine texels in the neighbourhood, giving a blur effect.

over our elements. So while on the CPU to process the elements of a grid we would have to loop over the X and Y indices, and have some core processing in the middle, it is this core that would be well suited to the fragment shader when converting code to run on the GPU. Woolley (2005) extends this analogy to explain that many of the performance considerations we would typically have in mind when writing the inner loop of an algorithm on the CPU are equally valid when considering the code in our fragment shader. Primarily, we want to put as little code as possible in the fragment shader itself, and pre-compute variables outside the fragment shader where possible, to avoid re- calculating redundant values for every fragment. The expense of inner-loop branching is likewise a pitfall on the GPU, even more so because it prevents efficient instruction-level parallelisation by the GPU hardware. However, this said, often the shader compiler in modern graphics drivers is clever enough to optimise away apparent inefficiencies, and especially compared to JavaScript, putting redundant computation in the fragment shader may be faster, so the only real way to know is to profile and test.

For our convolution example, the fragment shader is where we do the main work, shown in Figure 13. We have our kernel to convolve with passed in as a uniform 3×3 matrix (bearing in mind the OpenGL matrices are column-major), and our texture coordinates passed from the vertex shader give our centre texel. Convolving is then a matter of accessing the 9 texels in the neighbourhood and computing a weighted average using the values from the kernel, which is easiest to do with a couple of `for` loops. (We might think it faster to unwrap these loops and specify the texture coordinates offsets individually, but the compiler likely does this already.) We output our weighted average by assigning it to `gl_FragColor`, which causes this pixel colour to be output during rasterisation. We can observe the effect of several convolution kernels in Figure 14, along with an application for edge detection.

Convolution can be seen as a *gather* operation, since we are obtaining data from nearby locations in our grid of pixels, looking up values in texture memory. We prefer that our computations be structured as a gather, since fragment shaders are well suited to gathering, but it is impossible to perform *scattering*, which is when we write out to different locations in memory, distributing values to other elements in the grid. The fragment processed can only write to precisely one location. As described in Buck (2005), if we require a scattering operation, the first step should be to see if it can be converted into a gather. Techniques for dealing with scattering include adding a layer of indirection, by outputting an address along with our output value, which can then be processed by later passes to give a contiguous array.

Once we have the shaders set up, launching our computation is then just a matter of drawing geometry with the `gl.drawArrays(gl.TRIANGLES, 0, 6)` function, which will draw the six vertices of our triangles. To run in real time using images from the webcam, we use the browser's `requestAnimationFrame` to call our drawing code at appropriate intervals. This relatively new function

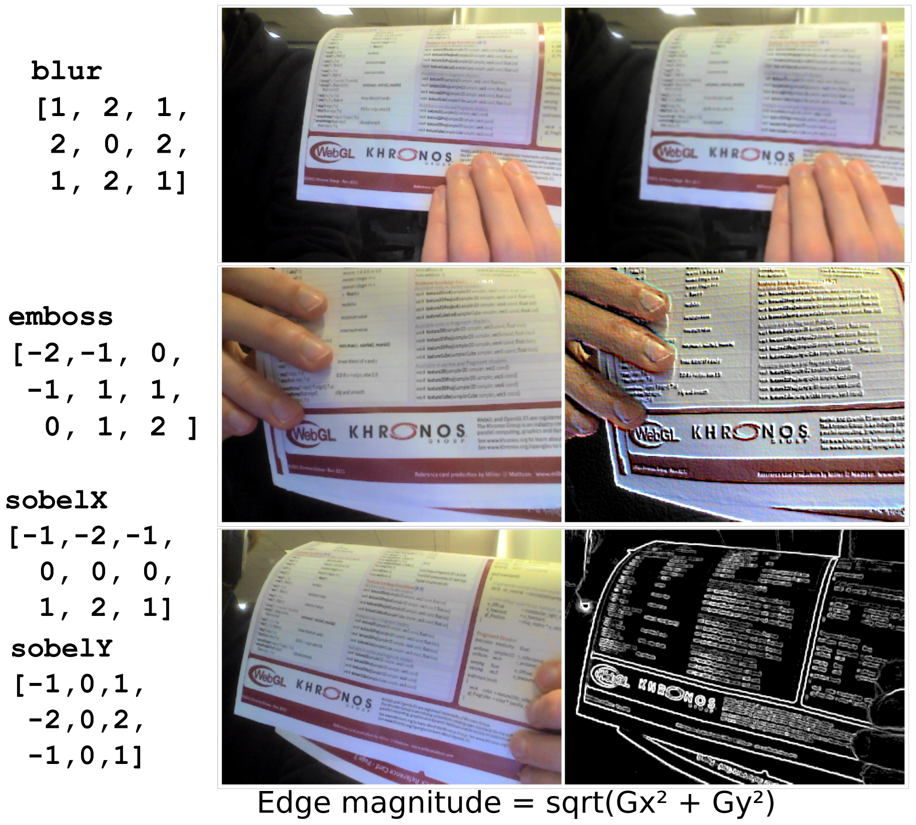


Figure 14: The effects of convolution using different kernels. We also show an application of convolution for edge detection, where we use the two sobel kernels to find the edge gradient in the X and Y direction, Gx and Gy respectively, and then output the magnitude of this gradient, showing edges in the image.

avoids the problems of JavaScript's `setInterval` function, which can be used for executing a function periodically, but has no knowledge of framerate, and will keep on running even when the window is not visible, wasting resources. Nevertheless, for browsers that do not support `requestAnimationFrame`, we offer a wrapper function that falls back to `setInterval`. In order to make all our shaders available through JavaScript, we have a Python script that constructs a JavaScript object containing all the shader source code, and a function `cv.getNamedShader` which deals with the task of compiling and linking our shader programs. We also provide functions in our WebCV library to deal with setting uniforms, uploading buffers, creating textures etc., allowing us to specify an associative array containing the uniform or attribute names with values as native JavaScript arrays or numbers, automatically dealing with the messy task of calling the correct version of many related functions, such as `uniform1f(), uniform2f(), ... uniform4iv() etc` which set uniforms of different types and sizes.

By now we are able to show the results of the convolution in our canvas on the screen, but we would also like to be able to use the results of previous computations in subsequent stages. For example, we might like to chain together multiple convolution passes. For this we use a framebuffer, which allows us to render our output to a texture. A framebuffer itself is just a lightweight structure, containing multiple attachment points, to which we attach objects containing storage, that we can render into. The attachment point used for storing the pixel colours drawn is `COLOR_ATTACHMENT0`, to which we can attach a texture whose values can then be accessed in subsequent fragment shaders. Another type of attachment is the `DEPTH_ATTACHMENT`, to which we can attach a depthbuffer, which will turn out useful later. Because we cannot read from the same texture that we are rendering to, if we want to use a pipeline of shaders it is necessary to use the "ping-pong" technique, alternating between two different framebuffers, such that one is used as the render target and one has its texture read by the shader, then swapping roles so that the output from the previous pass can be fed into the next one. Lastly, if we want to read back data into JavaScript, we can use the `gl.readPixels` command, which copies the contents of the current framebuffer's texture into a JavaScript array. Because this can be slow, it is best to do all computation possible on the GPU before reading back at the end.

3 Implementation

3.1 Cascade

3.1.1 Javascript Implementation

The core of the face detection method used is the cascade structure described previously, which subjects each window to progressively harder tests, each test being a stage in the cascade which specifies a number of weak classifiers with corresponding Local Binary Pattern features within the window.

Although the precise nature of these weak classifiers is crucial when constructing a cascade from training images (using the statistical method of Boosting to construct a strong classifier from individual classifiers performing only slightly better than chance) for the purposes of detection we need not be overly concerned with this. From a more abstract point of view, the weak classifiers simply tell us which points we need to look up in the integral image and which values should be used in the subsequent arithmetic in order to determine whether a window passes a stage. The main challenge then is to do this as fast as possible.

We initially implement the cascade using only JavaScript, running on the CPU, to give us a reference implementation which can then be used to assess the correctness of a WebGL version running on the GPU. For the moment we only consider the base scale of the cascade, 24×24 pixels, meaning we can only detect faces which occupy a window of these dimensions. We use the XML cascade file `lbpcascade_frontalface.xml` from the @OpenCV project, however we first use a Python script to convert this to a format more suitable for use with JavaScript, JSON (JavaScript Object Notation) which allows us to treat the cascade as a native JavaScript data structure, made up of JavaScript objects (associative maps with string keys, of the form `{"key": value}`) and arrays (eg. `[v1,v2,v3]` where values can be any type, not necessarily homogenous). An example of this data structure is given below, showing just one stage with one of its weak classifiers

```
var lbpcascade_frontalface = {
    "width": 24,
    "height": 24,
    "stages": [
        // 1st Stage
        {
            "stageThreshold": -0.7520892024040222,
            "weakClassifiers": [
                // 1st Weak classifier
                {
                    "featureRectangle": [6, 5, 4, 3],
                    "leafValues": [-0.654321014881134, 0.8888888955116272],
                    "categoryBitVector": [
                        -67130709,
                        -21569,
                        -1426120013,
                        -1275125205,
                        -21585,
                        -16385,
                        587145899,
                        -24005
                    ]
                }
                // ...2 more weak classifiers in this stage
            ]
        },
    // ...19 more stages (having up to 10 weak classifiers each)
    ]
}
```

Essentially, we have an array of stages, where each stage has a `stageThreshold` and its own array of `weakClassifiers`. The elements of each weak classifier merit further explanation:

- `featureRectangle`: Gives the position and dimensions of the weak classifier's Local Binary Pattern feature as a tuple $(x, y, width, height)$. The (x, y) coordinates give the top left corner of the feature, and the width and height are those of a single block of the feature, as shown in Figure 15.

- `leafValues`: The contribution of the weak classifier to the stage total in the case that it evaluates as negative (first value) or positive (second value). The stage total is the sum of these results for all its weak classifiers, and a stage passes if this total exceeds `stageThreshold`.

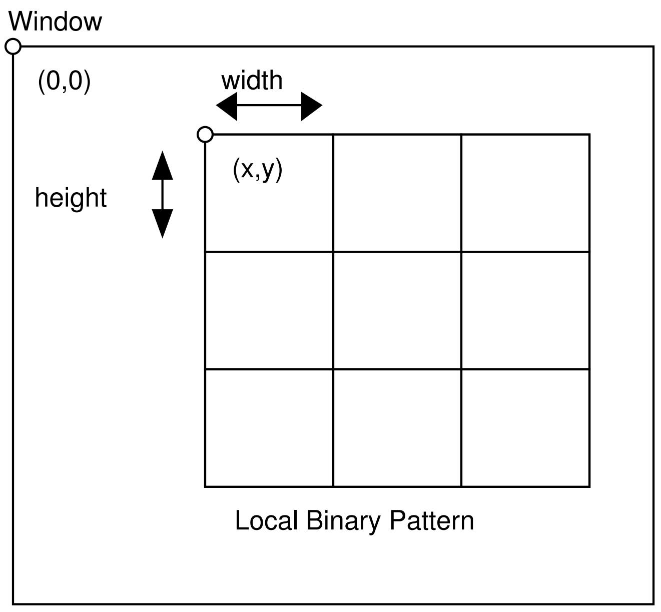


Figure 15: Interpretation of `featureRectangle` values

- `categoryBitVector`: Perhaps the most cryptic of the elements, this is a compact representation of which of the 256 possible Local Binary Patterns should be considered a positive result for the weak classifier, and which should be negative. It should be interpreted as eight 32-bit signed integers, giving 256 bits overall. This gives us a bit vector, where the i^{th} bit (counting from 0) is 1 if pattern i is negative and 0 if it is positive (using OpenCV's somewhat counterintuitive convention). For example, pattern 10101010_{bin} (which would represent alternating lighter and darker blocks than the centre) is 170 in decimal, and if this pattern were indicative of a face, then the 170^{th} bit in the bit vector would be 0. To check if bit i is zero or one, we can use the bitwise formula:
  `bitvec[i >> 5] & (1 << (i & 31))`
  which will be non-zero if bit i is set. This uses a right shift by 5 to select the three highest bits of i as an index to one of the 8 integers, then ANDs this integer with a number whose j^{th} bit (only) is 1, hence being non-zero if bit j of the integer is set. j is the lowest 5 bits of i and is obtained by masking i with 31 (11111_{bin}), and then to get a number whose j^{th} bit is 1 we left shift 1 by j.

This representation of the weak classifiers is in fact a simplification over that used in the original OpenCV XML file, which uses a rather vaguely named `<internalNodes>` element containing, in order, two dummy pointers to child nodes (unused, since we are dealing with a stump based classifier, containing only two leaf nodes, rather than a tree), the index of the feature rectangle (number 46 in example below) which is used to look up the actual rectangle specified elsewhere in the XML file, and then the eight elements of the bit vector.

```xml
<weakClassifiers>
  <!-- tree 0 -->
  <_>
    <internalNodes>
      0 -1 46 -67130709 -21569 -1426120013 -1275125205 -21585
      -16385 587145899 -24005</internalNodes>
    <leafValues>
      -0.6543210148811340 0.8888888955116272</leafValues></_>
  ...
```

Given this specification of the weak classifiers, we now look at how to compute the corresponding Local Binary Pattern values. From the `featureRectangle` associated with each weak classifier we can find the position of the blocks of our Local Binary Pattern within the window. In order to compute the Local Binary Pattern value we need to know the total intensity of each of these blocks (ie, the sum of all the grayscale 0-255 pixel values). By using the integral image, we can find the intensity of a block of any area with just four integral image values, those at the corners. We have nine blocks in a pattern, but some share corners, so we require 16 points in total, shown in Figure 16.

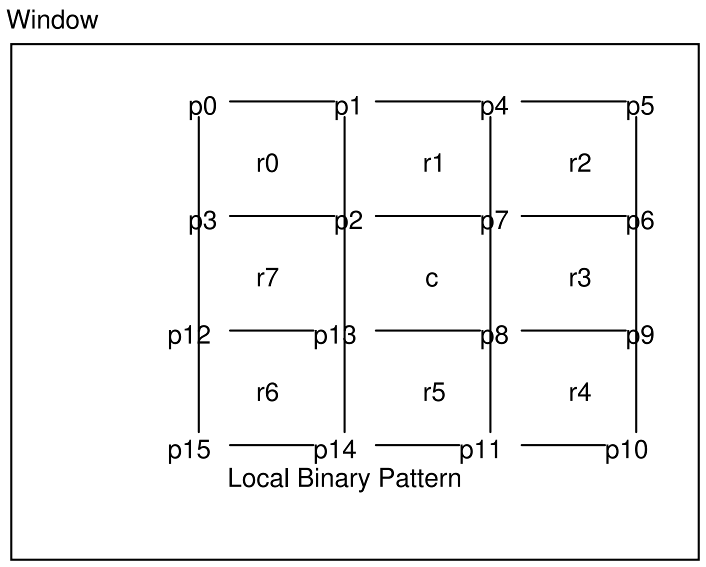


Figure 16: Points looked up for LBP (p0-15) and regions whose intensity we require (r0-7 and c)

Computing the total intensities is then just a matter of using the integral image trick described previously, which takes advantage of the fact that the value of the integral image at each point is the sum of all pixels to the top- left of it in the original image. For example, for the centre and first block we have:

```
c = p8 - p6 - p13 + p3;
r0 = p3 - p2 - p1 + p0;
```

Then, to find the value of our Local Binary Pattern as an 8-bit number, we must ask whether the value of r0 through to r7 is greater than the centre value c. If we define bit b_i as the result of the expression $r_i \geq c$ (where true=1, false=0) then our binary value will be $b_0b_1b_2b_3b_4b_5b_6b_7$, which can be computed in JavaScript using bitwise shifts and sums.

With this JavaScript implementation of the cascade we can now observe graphically the effect of each stage, by drawing masks (Figure 17) of which windows are accepted or rejected, plotting a white or black pixel respectively at the top-left position of each window. This lets us see the early termination effect which the cascade provides, whereby only a small number of windows pass through to the end, saving computational effort.

3.1.2 Adapting to WebGL

Having completed a reference implementation in JavaScript, it is now time to consider how best to take advantage of the capabilities of WebGL in order to speed things up. We have seen how WebGL can be used for computation on a grid, and this method adapts itself quite naturally to our need to compute many windows, whose results are independent of each other.

The main strategy for the initial implementation of face detection in WebGL is to offload the computation of each stage, involving the lookup operations on the integral image, the calculation of the Local Binary Pattern values, and the subsequent window evaluation (our “inner loop”) to the fragment shader. This allows the “sliding window” to be parallelised so that we are evaluating multiple window positions at once. We can consider our window to be “anchored” to the fragment position in the top left (Figure 18)

Since the classification of a window requires the evaluation of a number of stages, one choice we must make is whether to evaluate all these stages at once, looping over each stage within the fragment shader, which would require a single shader program able to handle every stage, or whether we should break up the evaluation into a different draw call per stage, keeping track of which windows have passed the previous stages using a texture.

Although having a single monolithic shader would avoid the overhead of a separate draw call for each stage, there are reasons not to prefer this approach. Firstly, we are limited in the amount of Uniform variables we can transfer to the

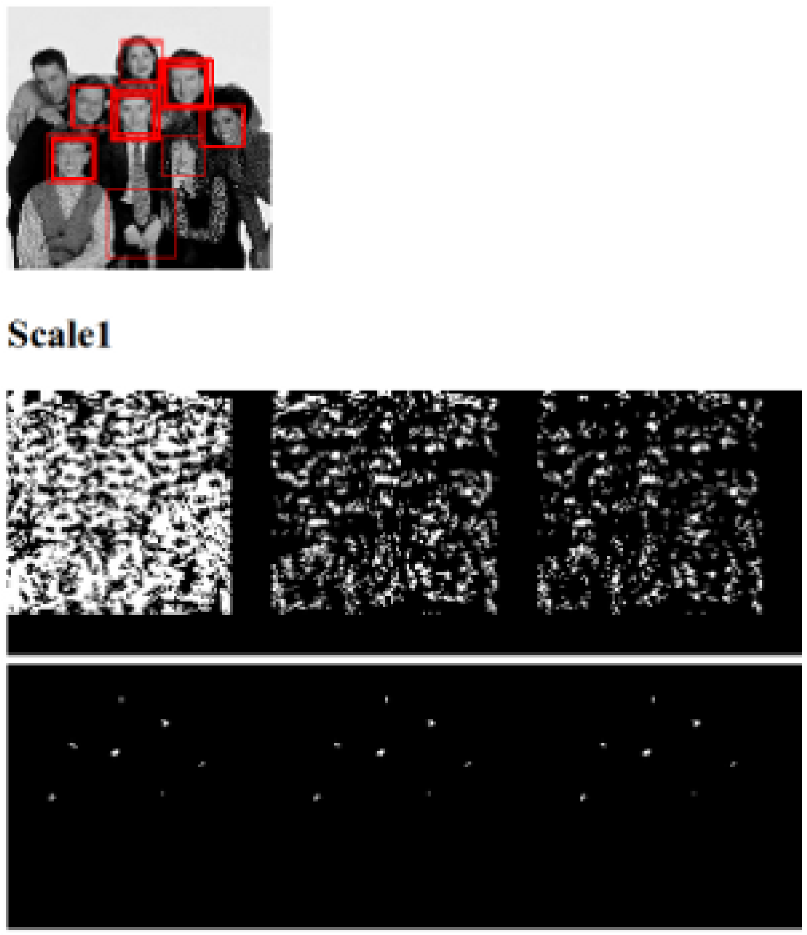


Figure 17: The output from the first and last three stages of our javascript cascade

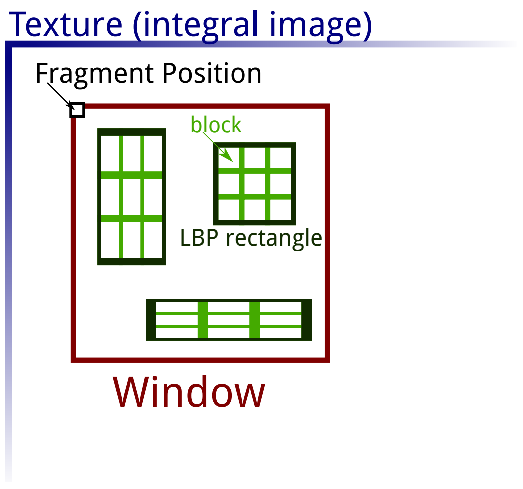


Figure 18: Window for a stage with 3 weak classifiers, processed by the fragment shader. The corners of the blocks are the positions which must be fetched from the integral image texture.

shader. According to WebGLStats (2013) 81% of users have a maximum of 221 Uniform 4-component variables available in the fragment shader, although for 9% of users it is as low as 29. (These limits, which can be queried at runtime, give the number of vectors of 4 floats which can be used. Quite how this corresponds to limits on other types and arrays does not seem to be documented, but they would be expected to share the same registers so be similarly limited, and there is some (desktop OpenGL) discussion in Krumlinde (2011)) At 20 stages, each having up to 10 rectangles, we would struggle with these limits. We could resort to packing the data into textures, however another limitation is that loop conditions and array sizes in GLSL must be based on constant expressions, available at compile time. Since each stage has a different number of weak classifiers, we would want to loop over a different number at each stage. We could solve this with a branch or `break;` within the loop, however branching carries with it performance penalties. For more information see the appendix on branching.

By instead having a shader program for each stage, we can use the same source code for each stage, but inject compiler `#define`s specifying certain constants, such as the stage number and the number of weak classifiers in the stage, allowing us to use constant expressions for loop conditions. We can then also make use of preprocessor directives so that different code is compiled for different stages. Since GLSL has no array literals, we still need to upload the arrays of `leafValues` and `featureRectangles` as Uniform arrays, but because their size will be determined by the number of weak classifiers in a single stage (maximum 10), we avoid pushing the limits of Uniforms allowed. One disadvantage of having many shaders is that we must compile them all, which gives a small delay upon initialising the face detector (around a second for the 20 stages), however this is a one-time cost upon startup so not too important for most applications.

Each stage writes out a texture with a white pixel for each window accepted, a black pixel otherwise. The texture from the previous stage is used as an input to the next stage, using the framebuffer "ping pong" technique, to avoid computing windows which have already been rejected. An additional advantage of splitting the computation into multiple draw calls is that we can inspect these intermediate textures in order to debug our code.

The tasks we must do in our fragment shader are:

1. Check if the window has been rejected
2. Compute the Local Binary Pattern value for the various weak classifiers within the window
3. Depending on which pattern we get, determine if we have a positive or a negative weighting towards the window being a face
4. Sum the weights contributed by all the classifiers, and compare against an overall threshold for the stage
5. Write out whether the window passes as the fragment colour

Some of these tasks are straightforward, however one complication is that we have no bitwise operators in the fragment shader (since GPUs typically work using floating point hardware for all numbers, what we might like to think of as a bit pattern has no relation to the value stored in memory). This means that we cannot compute the Local Binary Pattern value (a byte where each bit indicates if an outer block is greater than the centre) in the natural way. However we can get around this by forgetting about bits and thinking purely arithmetically, in powers of two, which gives us the following formula.

```
lbp = (int(r0 >= c) * 128)
      + (int(r1 >= c) * 64)
      + (int(r2 >= c) * 32)
      + (int(r3 >= c) * 16)
      + (int(r4 >= c) * 8)
      + (int(r5 >= c) * 4);
```

The lack of bitwise operators also means we cannot use a bitvector to specify which Local Binary Patterns should be considered positive or negative. Instead, this data is accessed from the shader by using a grayscale texture as a lookup table (Figure 19). There is one row for each stage in the cascade, so the height is the number of stages, and for each LBP rectangle we have 256 possible patterns.

A black pixel is used to indicate a positive pattern, a white pixel a negative pattern (OpenCV's strange convention). The width of the texture is then 256 $\times$ the maximum number of weak classifiers. For the default cascade used, we have 20 stages and a maximum of 10 LBP rectangles, giving a 2560×20 texture. This differs from the more compact representation used by the OpenCV and JavaScript implementation which packs the data for one rectangle into 256 bits (8 32-bit ints) per rectangle, since we use up an entire byte for each possible pattern, but the texture only needs to be uploaded to the GPU once and is fairly cheap to lookup.

Figure 20 gives an overview of the main components of our shader so far, labelled with the five tasks above.

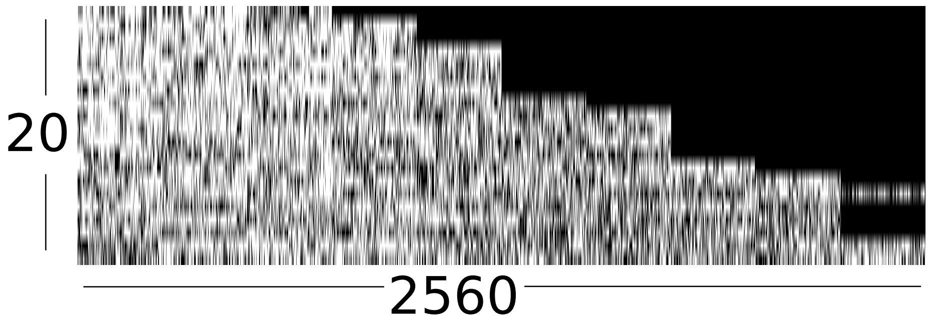
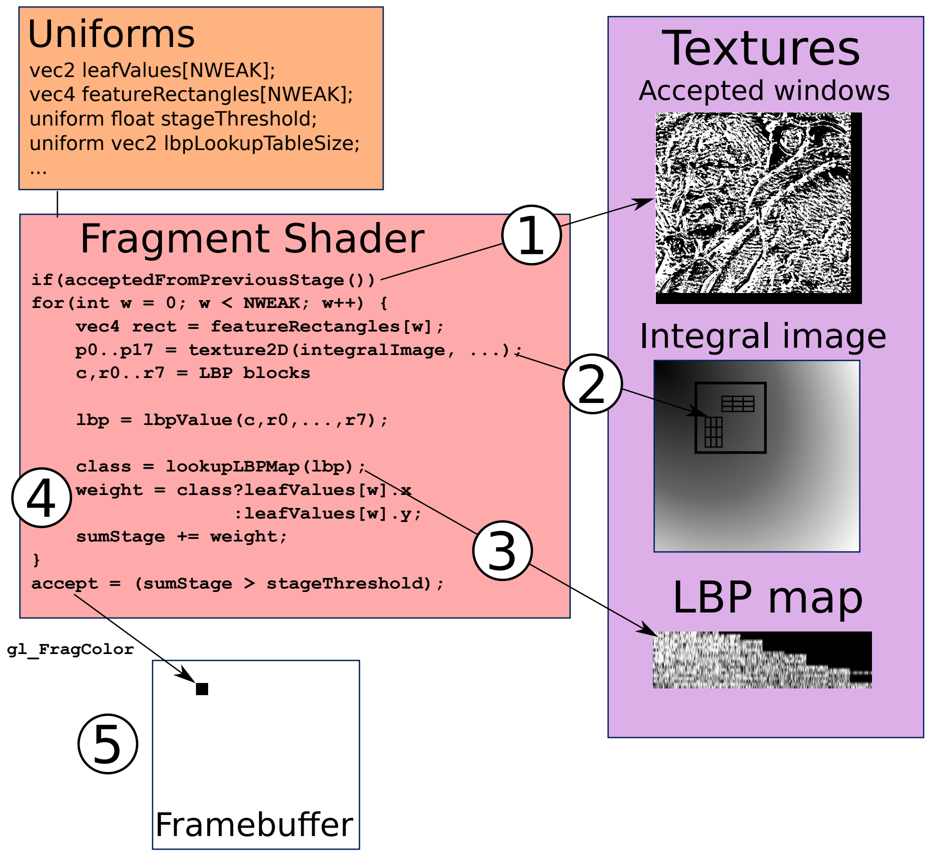


Figure 19: A (heavily resized) version of our lookup texture for Local Binary Pattern values, with a row for each stage, and 256 horizontal pixels for each weak classifier, where a black pixel is positive and white is negative. This gives a nice visualisation of how the number of weak classifiers increases as we get to later stages.

3.2 Scaling

On top of this loop over stages, we also need to consider different scales, to be able to detect faces of different sizes in the image. There are two potential ways to implement scaling of our classifier. The most obvious way is simply to rescale our image, and this is the approach used by OpenCV. However this would be expensive since it requires recomputation of the integral image, which is done in JavaScript, so we would have to upload many images to the GPU. The second way is to scale the classifier itself. This is done by setting a scaling factor (we use 1.2) which we successively multiply the window size and rectangle offsets by. We run the detection for each scale, starting from the base 24x24 pixel window size, until some maximum where the window would be too big to fit in the image. However one problem with this approach, mentioned by Obukhov (2011), is that when we are not scaling by an integer amount, our features are no longer aligned with pixel locations. The rounded coordinates mean that the



```
vec2 leafValues[NWEAK];
vec4 featureRectangles[NWEAK];
uniform float stageThreshold;
uniform vec2 lbpLookupTableSize;
...
```

```
if(acceptedFromPreviousStage())
for(int w = 0; w < NWEAK; w++) {
    vec4 rect = featureRectangles[w];
    p0..p17 = texture2D(integralImage, ...);
    c,r0..r7 = LBP blocks

    lbp = lbpValue(c,r0,...,r7);

    class = lookupLBPMap(lbp);
    weight = class?leafValues[w].x
                  :leafValues[w].y;
    sumStage += weight;
}
accept = (sumStage > stageThreshold);
```

Figure 20: Diagram and pseudocode of our Stage fragment shader

areas fetched using the integral image no longer correspond with those specified by the features, which can give inaccurate results. It is possible to re-weight the features to fit the scale, however for our purposes we will simply live with the slightly inaccurate values.

After detection is run on each scale, the accepted window texture is read back to a JavaScript array using the WebGL `gl.readPixels` command, and used to draw appropriately sized rectangles at the locations where faces have been found.

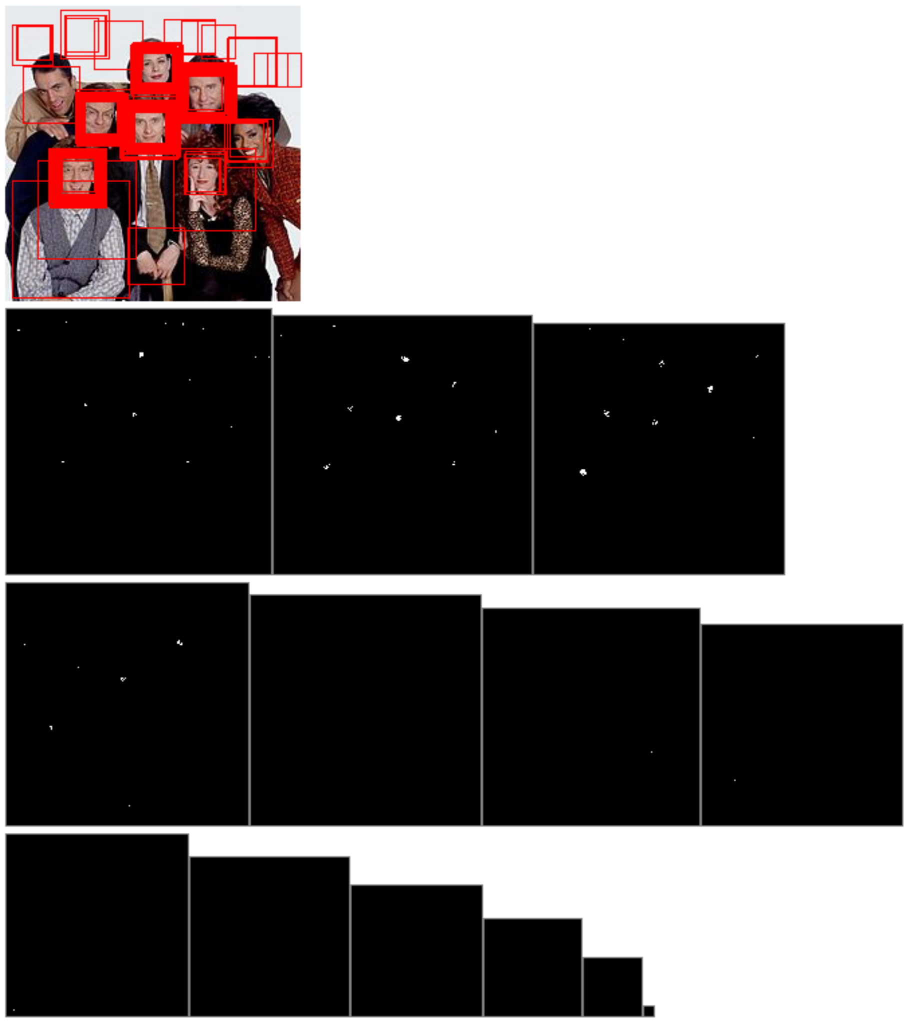
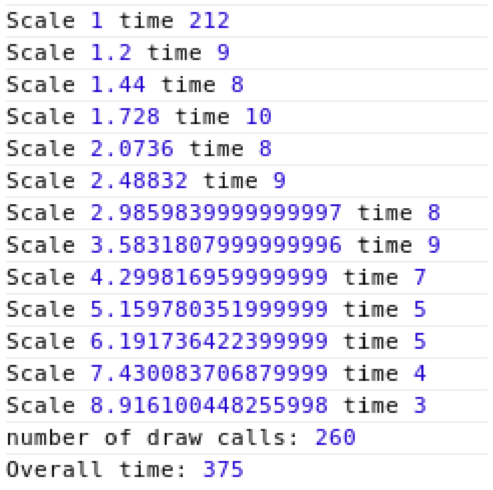


Figure 21: Output of detection at different scales

3.3 Optimising

3.3.1 Profiling

In order to test how fast this initial implementation is we can insert some timer calls. We measure the time for each scale as well as the overall time for the detection call (after the inital setup of shaders and textures), on an image of dimensions 320x240 containing three faces of different scales. This gives the output shown below.



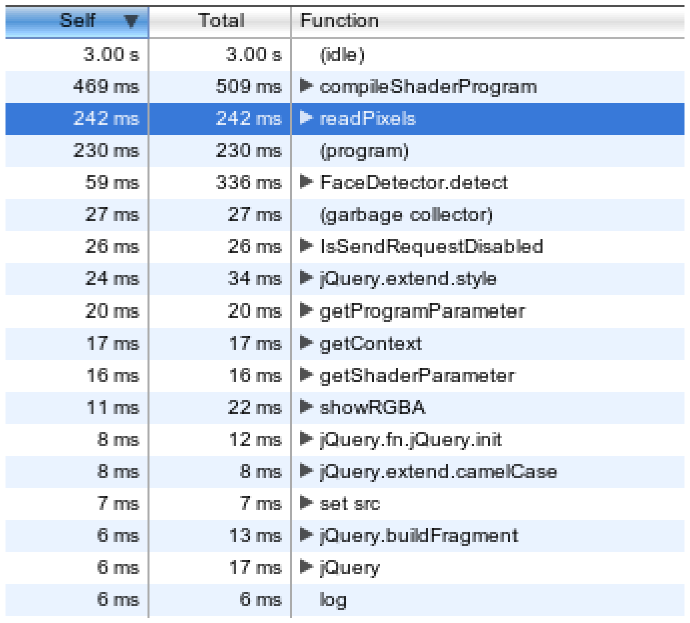
```
Scale 1 time 212
Scale 1.2 time 9
Scale 1.44 time 8
Scale 1.728 time 10
Scale 2.0736 time 8
Scale 2.48832 time 9
Scale 2.9859839999999997 time 8
Scale 3.5831807999999996 time 9
Scale 4.299816959999999 time 7
Scale 5.159780351999999 time 5
Scale 6.191736422399999 time 5
Scale 7.430083706879999 time 4
Scale 8.916100448255998 time 3
number of draw calls: 260
Overall time: 375
```

Figure 22: Initial timing

This gives an overall time of 375 milliseconds, obviously not good enough for real time detection. Looking more closely, the majority of time seems to be spent on the first scale, which takes 212 ms, whereas the other scales take 10ms or less. Using the Chrome Javascript Profiler tool we can investigate further by checking which functions are taking up the most time.

This shows that (besides the initial overhead of setting up the shaders) most of the time is spent in the `gl.readPixels` function, responsible for transferring image data from GPU memory back to JavaScript. Reading back pixels from the GPU can certainly be slow, however, because most WebGL calls are non-blocking, when we call a function that depends on the drawn output, it will wait until all the previous commands have finished their work. This means that slowness in other areas gets "blamed" on later commands.

The previous results were timed using a single image, running the detection once after the page loads. In a real scenario we would want to be detecting continually on each frame. This leads us to investigate the result of running the detection on two different images, one after the other, without refreshing the page. (In



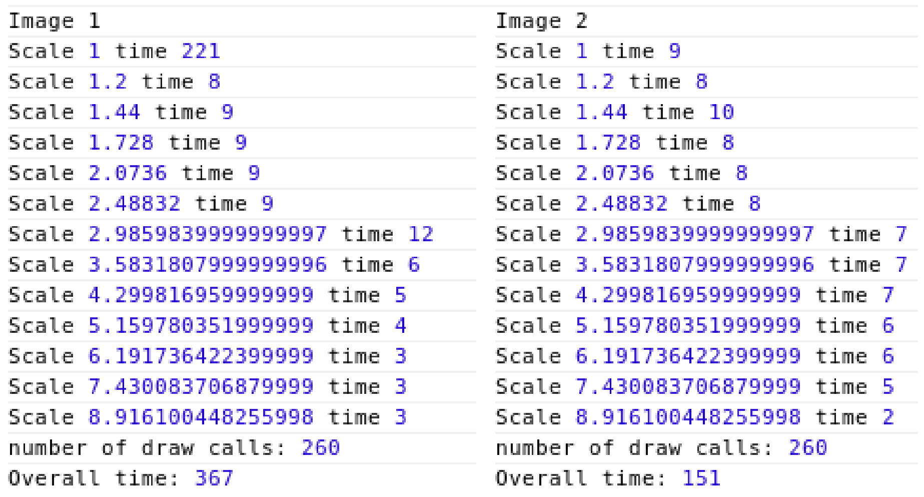
Self	Total	Function
3.00 s	3.00 s	(idle)
469 ms	509 ms	▶ compileShaderProgram
242 ms	242 ms	▶ readPixels
230 ms	230 ms	(program)
59 ms	336 ms	▶ FaceDetector.detect
27 ms	27 ms	(garbage collector)
26 ms	26 ms	▶ IsSendRequestDisabled
24 ms	34 ms	▶ jQuery.extend.style
20 ms	20 ms	▶ getProgramParameter
17 ms	17 ms	▶ getContext
16 ms	16 ms	▶ getShaderParameter
11 ms	22 ms	▶ showRGBA
8 ms	12 ms	▶ jQuery.fn.jQuery.init
8 ms	8 ms	▶ jQuery.extend.camelCase
7 ms	7 ms	▶ set src
6 ms	13 ms	▶ jQuery.buildFragment
6 ms	17 ms	▶ jQuery
6 ms	6 ms	log

Figure 23: Javascript Profile

fact the same image, but flipped horizontally, so we would expect similar face detection results, but avoid any clever caching by the browser).

This gives the surprising result that, while the first run of the detection takes a long time, the second is considerably shorter, with times between 2 and 10 ms for each scale. While we cannot determine the exact cause of this, it seems that from a “cold start”, the initial readPixels has some overhead which is not experienced on subsequent calls. So while readPixels is still the slowest factor, once the detection gets going we need not worry about reads taking over 100ms. From here, the best strategy to improve overall time seems to be to minimise the number of readPixels calls needed, ideally with just one at the end of detection rather than intermediate calls for each scale.

While refactoring the code to “pingpong” by flipping between multiple framebuffers, rather than the more expensive technique of using one framebuffer and attaching different textures in turn (as recommended by Tavares (2011a) at 37m20s), it was discovered that the slowdown on the first readPixels seemed to disappear. However, after some work to narrow down the exact conditions which would produce the slowdown, it was determined that this optimisation alone was not responsible for the difference, but rather that it was determined by the ordering of the calls to attach textures to the framebuffers, relative to the code setting up the shaders. It turned out that, if at least one `gl.framebufferTexture2D` call was before the shader setup, the initial readPixels call took 10ms, whereas otherwise it took over 200ms. The initial setup which includes compiling the



```
Image 1                                   Image 2
Scale 1 time 221                          Scale 1 time 9
Scale 1.2 time 8                          Scale 1.2 time 8
Scale 1.44 time 9                         Scale 1.44 time 10
Scale 1.728 time 9                        Scale 1.728 time 8
Scale 2.0736 time 9                       Scale 2.0736 time 8
Scale 2.48832 time 9                      Scale 2.48832 time 8
Scale 2.9859839999999997 time 12          Scale 2.9859839999999997 time 7
Scale 3.5831807999999996 time 6           Scale 3.5831807999999996 time 7
Scale 4.299816959999999 time 5            Scale 4.299816959999999 time 7
Scale 5.159780351999999 time 4            Scale 5.159780351999999 time 6
Scale 6.191736422399999 time 3            Scale 6.191736422399999 time 6
Scale 7.430083706879999 time 3            Scale 7.430083706879999 time 5
Scale 8.916100448255998 time 3            Scale 8.916100448255998 time 2
number of draw calls: 260                 number of draw calls: 260
Overall time: 367                         Overall time: 151
```

Figure 24: Timing on two images

shaders always takes around a second, so while the order of calls does not change the initial setup time, it allows the "warm up" time required before readPixels to effectively be hidden behind the time needed to compile the shaders. This is likely because the shader compilation is mostly CPU-bound, allowing other tasks to be done in parallel on the GPU.

3.3.2 Sharing the same texture across scales

Although the slowness attributed to readPixels also includes the expense of previous commands, we would still like to minimise the number of times we have to copy data from the GPU. Ideally we want to only do it once, after all the computation is finished, instead of once for every scale. In order to eliminate the intermediate readPixel calls, we need to write the output from each scale to the same texture, preserving the pixels output from the previous scale, and encoding the scale in the pixel value. To indicate the scale of an accepted window we can simply write out the ordinal number (1,2,3. . .) of the scale as a colour value, or 0 if the window is rejected. The size to multiply the rectangle by is then $scaleFactor^{(scaleNumber-1)}$. The use of two textures to "pingpong" the results between each stage in a scale remains as before, except that on the final stage we write to a shared final output texture. One limitation is that, if we have two detections of different scales at exactly the same position, the later (larger) scale will overwrite the previous one. However, this should be a relatively rare occurence, and should not make too much difference when all the rectangles are grouped to find the final face positions. Another issue is that we want to keep the previous written pixels, instead of writing a black pixel for a rejected window in a subsequent scale. The simplest way to prevent output of any pixel at all is to use the `discard;` statement in the fragment shader. However, in certain cases (discussed in Jave (2011)) this may invoke a performance penalty, particularly on mobile GPUs. An alternative is to use OpenGL's blend modes, which specify how pixels written should be blended with the pixels already present.

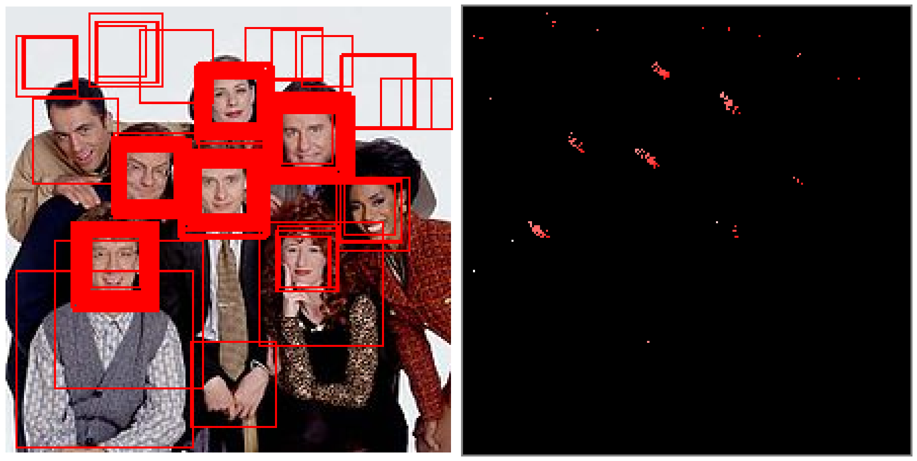


Figure 25: Writing out pixels for each scale. A lighter colour pixel indicates a larger detection

First an implementation is created using `discard;`, giving an average time of 71ms per detection run (for a 320x240 image over 20 runs), compared to 110ms using `readPixels` for each scale under equivalent conditions.

The implementation is then adapted to use blending, in order to test which would give the best performance. As explained in Thomas (2009), the `gl.blendFunc(sfactor, dfactor)` function sets the factors which the source (being drawn) and destination (already in the framebuffer) should be multiplied by, where `sfactor` and `dfactor` are symbolic constants determining where the factors should come from. The output for each colour channel is given by $Result = SourceVal \times SourceFactor + DestVal \times DestFactor$. We set `sfactor` to `SRC_ALPHA` and `dfactor` to `ONE_MINUS_SRC_ALPHA`, which means that when outputting `gl_FragColor` we can set the alpha value to 0.0 to completely preserve the existing pixel.

Comparing the timing of the two techniques over 100 iterations, there turned out to be almost no difference in the mean time, at least on a laptop Intel GPU, although as shown in the box plot the Blend version had a slightly greater variance (possibly due to sporadic browser pauses caused by garbage collection). For the moment we shall prefer the Blend version, to avoid potential slowness with other GPUs and because it allows the shader code to be simplified, eliminating a branching condition to explicitly check if the window was rejected. However, knowing that `discard` can be used (at least on non-mobile GPUs) with little penalty shall prove useful in further optimisation.

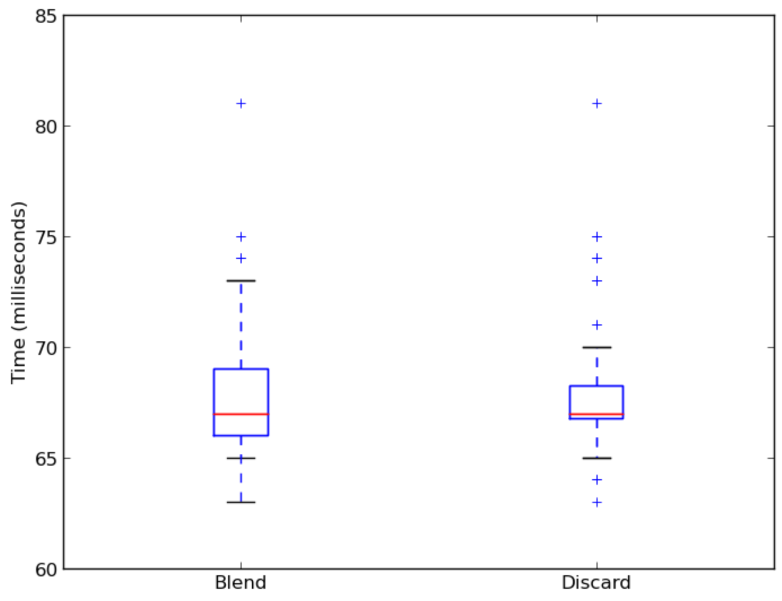


Figure 26: Box plot of timings using Discard vs Blend, for 100 iterations

3.3.3 Timing stages

In order to analyse the times of operations at a finer granularity, we want to time each draw call individually. However, because the CPU and GPU operate asynchronously, each draw call will in fact return immediately, and the CPU will only wait for the GPU to finish when some operation requiring information from a framebuffer is performed. Therefore, we insert a dummy readPixels operation, reading only 1 pixel, after each draw. Because some times are very small (below 1ms) and difficult to measure accurately, we also artificially repeat each draw operation 10 times, and divide the total time by 10. In this way, we can obtain a detailed profile of how much time is spent running the shader for each stage and scale (Figure 27).

We observe that, as expected more time is spent in the early stages, because the first stage must run on all windows, whereas for laters stages some windows are rejected. Increasing the scale also shows a decrease in time, since less window positions need to be evaluated, although this is only really noticeable in the first two stages, the subsequent stages showing around the same time regardless of scale (Figure 28)

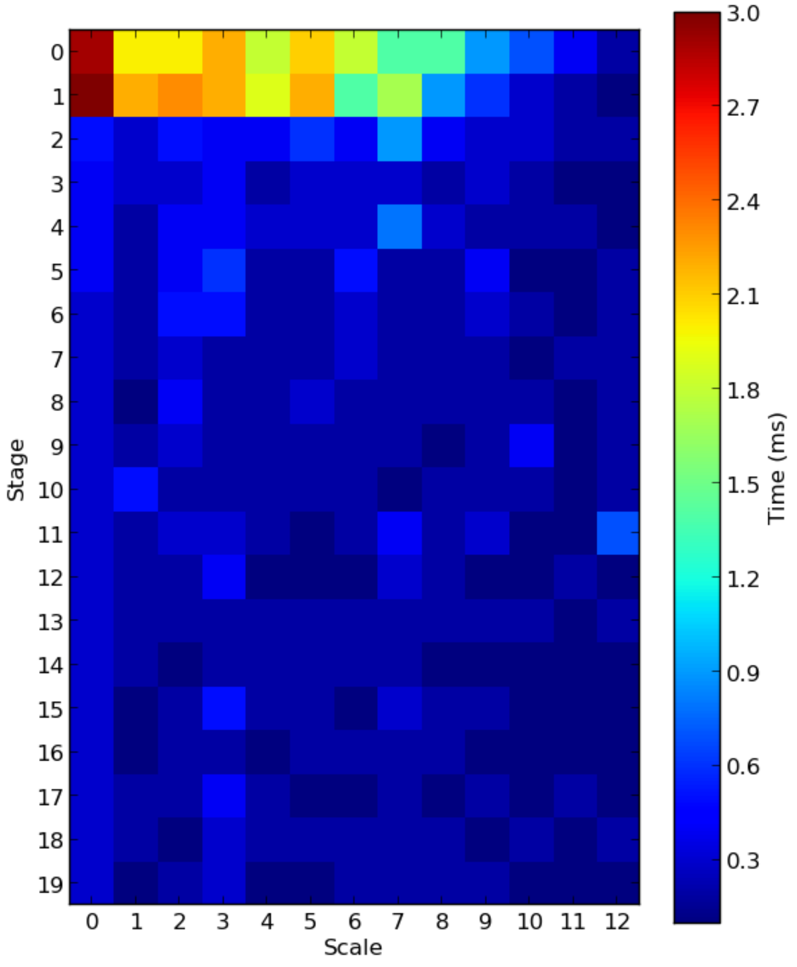


Figure 27: Times of stages and scales

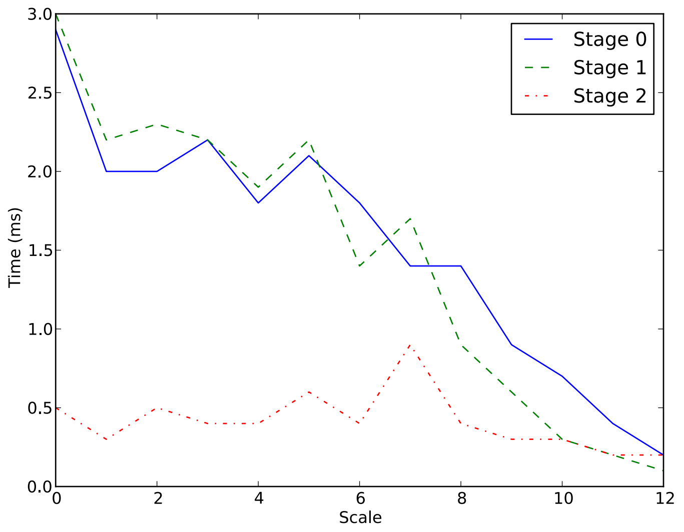


Figure 28: Times for the first 3 stages at different scales

What is interesting to note is that the first three stages take up 48% of the time, while the remaining 17 stages take up 52% of the time. So while it is tempting to try to chip away at the above-2ms times in the early stages, we have a "long tail" effect where the sub-0.5ms times of later stages add up to a significant proportion of the overall time. Therefore, treating the early stages as special cases (such as manually fine-tuning the shader code for these specific stages) is unlikely to provide much of an advantage, compared to general techniques that apply equally to the later stages.

To gain further insight we must identify precisely where the bottleneck is. At an abstract level, all the fragment shaders are doing is

1. Looking up some values in the integral image and LBP lookup textures
2. Doing some maths to determine what value to output

Now, GPUs are typically very fast at carrying out floating point calculations, so we wouldn't expect the "maths" portion to be overly challenging. Harris (2005) explains this using the concept of "arithmetic intensity", the ratio of computation to bandwidth.

```
arithmeticIntensity = operations/wordsTransferred
```

According to Harris (2005), applications that benefit most from GPU acceleration are those with high arithmetic intensity, where "The data communication required to compute each element of the output is small and coherent". So ideally, the amount of data fetched from textures would be small, and would be spatially localised, in order to take advantage of caching. Unfortunately, in order to calculate the 9 blocks of the rectangle for each classifier, we require 16 texture lookups, and the positions fetched for a window are not guaranteed to be close together. Since the number of weak classifier rectangles can vary from 3 in the first stage to 10 in the later stages, we are talking about $3 \times 16 = 48$ at best and $10 \times 16 = 160$ at worst texture lookups. For the base scale they will at least be within the same 24×24 area, but when the window is scaled we will be fetching values locations more spread out over the image. Texture caches are typically optimised for some 2D neighbourhood of a few texels, which great for applications such as convolution where we just need to look up adjacent texels, but is not ideal for more general purpose approaches.

To test the theory that the texture fetches are responsible for most of the slowdown, we create a test shader which performs the same texture fetches as our face detection shader but does not do anything useful with the result (instead just outputting the sum of the values, to ensure the fetches are not optimised out). Performing the same texture fetches as the 1st stage of the cascade (48 fetches), and timing over 1000 iterations, we get an average time of 3.1 ms per draw call, which is pretty much identical to the full shader. Further, commenting out half the fetches reduces the time to 1.3ms, clearly showing the impact of texture fetching on the time. We can't really do much about the fact that we need to access so many values, since they are all required to give a correct result. However, one source of inefficiency is that we are using a texture merely to keep track of which locations have been rejected, meaning that every fragment needs to read at least one texture value, if only to determine that it has nothing else to do.

3.3.4 Z-Culling

Z-culling can be used to remedy the aforementioned inefficiency, by ensuring that certain fragments are never processed in the first place. Harris & Buck (2005) explain how a GPU feature called the depth buffer, intended to be used to avoid shading pixels of 3D objects that will never be seen due to being hidden behind other objects, can be used in general purpose computation to mask out values tht we don't want to be processed. When the depth buffer is enabled, rasterising 3D geometry at a position also causes a normalised version of its Z coordinate to be written as a value between 0.0 and 1.0 in the depth buffer, where 0.0 represents a near location and 1.0 far. Upon drawing further geometry, the Z coordinate of incoming fragment positions that would be shaded is checked against the value in the depth buffer. When using the default `gl.LESS` depth test, if the incoming value is less than the existing one, the fragment is taken to be "in front" of the existing geometry. Otherwise the incoming fragment is not processed. If we set our rectangle to be at depth position 0.5, then depth buffer values of 0.0 and 1.0 will cause elements in our grid to be turned on or off respectively.

We would like to use a depth buffer to control which windows have been accepted or rejected in our cascade. That way we could eliminate the need for a texture to pass around this information, and remove a branch from the fragment shader, since the shader would not run at all for rejected windows. Unfortunately, unlike in desktop OpenGL, in WebGL we cannot set the `gl_FragDepth` variable in the shader. The only depth that can be written to the depth buffer is that determined by the underlying geometry. However we have another option, not writing any depth at all, by using the `discard` keyword described previously, which prevents any output at all for the fragment.

We can use this to turn the problem on its head somewhat. If we initialise the depth buffer to the far value, 1.0, and then draw pixels on our quad at 0.0, these pixels will fail subsequent depth tests, because the test `0.0 < 0.0` fails. However if we discard a fragment then the previous value of 1.0 in the depth buffer will remain, meaning that the fragment will still be processed in subsequent draws. This means that, somewhat paradoxically, we can set up a situation where if we want to accept a window then we must discard it, and if we want to reject it we must output some arbitrary value in order to update the depth buffer. Then for the final stage, as before, we write out a value indicating the scale. Understanding how the depth buffer interacts with the fragment shader output is best done visually, as in Figure 29. We also give a sample of Z-culling output on an actual image, Figure 30.

Z-culling lets us eliminate the accepted windows texture and associated checks in the fragment shader. But it also means that we can go a step further and disable writing colour buffer values altogether for the intermediate stages, since everything is handled by the depth buffer. We no longer need any textures to "ping pong", so we can simply not attach any texture to our framebuffer and make do with just a depth buffer. This saves on bandwidth, since we no longer have to be writing RGBA values just to indicate a binary result.

The Z-culling optimisation shaved off around 6 to 10ms for a 320x240 image, which is fairly significant when dealing with real time detection, although the savings vary according to how many candidate faces the image has. An alternative to the depth buffer is the stencil buffer, an 8 bit buffer which can be used for more general comparison functions. We tested using the stencil buffer for the same purpose, but it gave no difference in performance.

3.3.5 Non-optimisations

We briefly examine some attempted optimisations which in fact made no difference, or made things worse. If an optimisation has no effect, it doesn't necessarily

We start with a depth buffer that is cleared to 1, the far value, meaning that all pixels on our quad (which is at the near z value, 0) will be processed.

Depth buffer

1	1	1	1
1	1	1	1
1	1	1	1

We then run the first stage, which will discard the output if the window is accepted, writing a 0 if it is rejected. This causes the depth buffer to be updated to the z value of the quad, 0, for those windows which have been rejected, whereas the previous depth of 1 is preserved for the accepted windows, since their output is discarded.

1st Stage Output

d	discarded
0	rejected

d	0	d	0
0	d	d	0
0	0	d	0

Updated depth buffer

1	0	1	0
0	1	1	0
1	1	0	1

On subsequent stages, those pixels with a depth value of 0 will not be processed, since we are using the "gl.LESS" depth test to compare the depth of the pixel on our quad with the depth in the depth buffer, and because zero is not less than zero, the test fails, so the pixel is not processed.

2nd Stage Output

x	not processed
d	discarded
0	rejected

d	x	0	x
x	0	d	x
x	x	0	x

Updated depth buffer

1	0	0	0
0	0	1	0
1	1	0	1

Figure 29: A visual explanation of Z-Culling

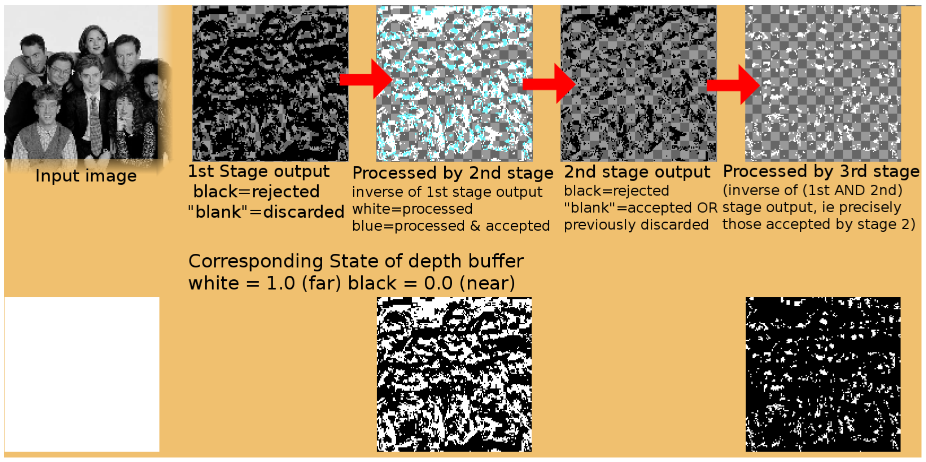

Input image
1st Stage output
black=rejected
"blank"=discarded
Processed by 2nd stage
inverse of 1st stage output
white=processed
blue=processed & accepted
2nd stage output
black=rejected
"blank"=accepted OR previously discarded
Processed by 3rd stage
(inverse of (1st AND 2nd) stage output, ie precisely those accepted by stage 2)
Corresponding State of depth buffer
white = 1.0 (far) black = 0.0 (near)

Figure 30: An illustration of Z-culling on an actual image. We use a checkerboard pattern to show the lack of a value

mean the principle is wrong, it can just be an indication that the bottleneck is in another place.

- Since we need to access various texture offsets in the fragment shader in order to calculate the Local Binary Patterns, and these offsets "move" along with the window position, the idea was to calculate these offsets in the vertex shader and use a varying to interpolate them linearly across the rectangle. According to Woolley (2005), "when multiple related texture coordinates are used…a common mistake is to compute the related values in the fragment program. This results in a possibly expensive computation being performed very frequently." However, testing a shader using this approach (Figure 31) gave no difference, perhaps indicating that moderns compilers are clever enough to "hoist" out these sort of values.

- In an attempt to take better advantage of locality of texture accesses, which gives values a better chance of being cached, we explored computing multiple windows at once in the fragment shader, which would involve looking up adjacent texture values, in theory better for caching. This shrinks the computational range, since we only output half as many fragments, and we need some way to reconstruct the original window locations. We tried packing up to four windows at once, using the four colour channels to output four different results, however in all cases it was slightly slower than computing a window per fragment. We also looked at iterating over scales within the fragment shader, but the "multiscale" shader took the same amount of time as the combined time for different scales. This seems to indicate that it is the sheer number of texture accesses made which accounts for the majority of the time used.

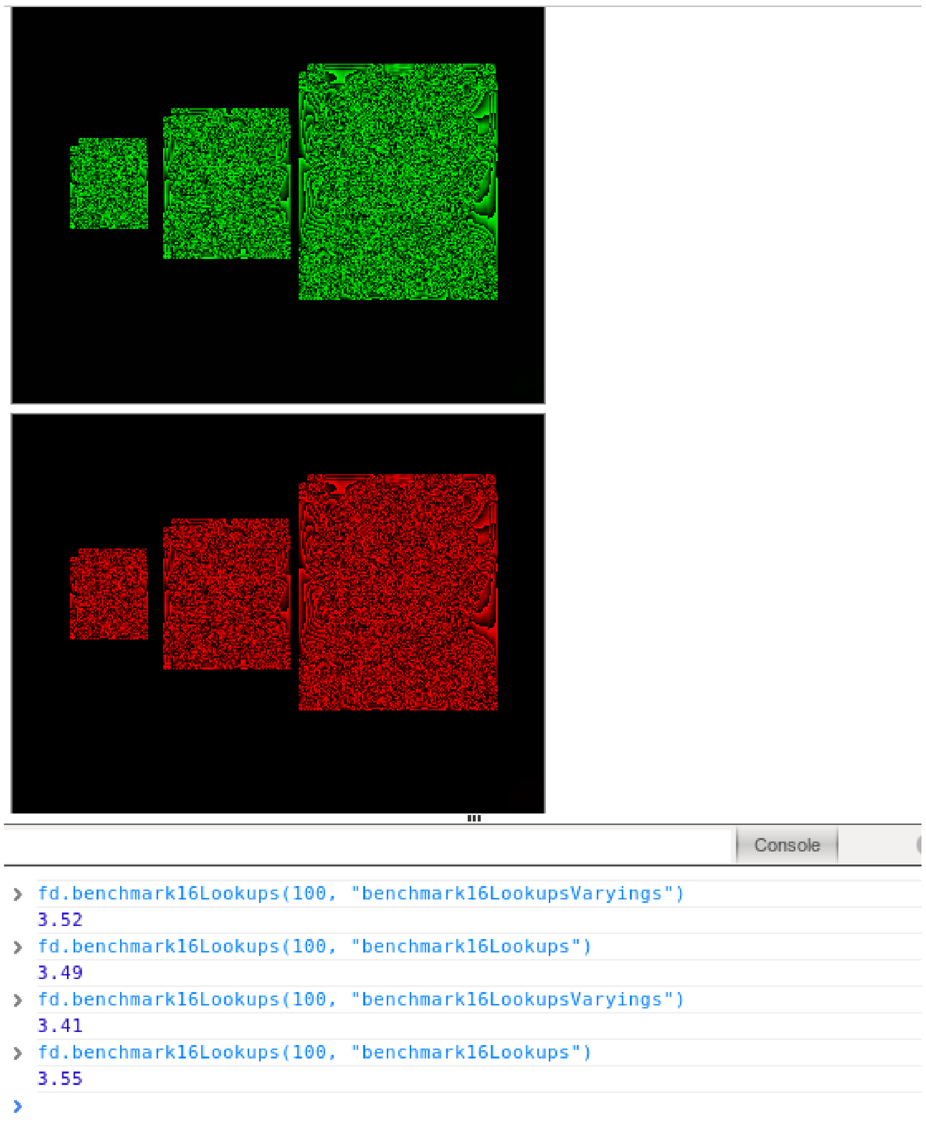
- Although we compute the integral image in JavaScript, it only takes around a millisecond. However, what is slower is first converting the original image to grayscale, for the sole reason that to get from an image to a JavaScript array we need to use a 2d `<canvas>` as an intermediary, and with Chrome the canvas `getImageData` method turns out to be quite slow (up to 8ms on a bad day). We looked at sidestepping the issue by computing the grayscale and integral image directly in WebGL, and we came up with a creative way of using blending and two passes of "shifting" the image down and then right in order to accumulate all the values for the integral, however this technique ended up taking longer. Nevertheless there are likely to be more efficient ways to calculate the integral image on the GPU, so this merits further investigation.

3.4 Evaluation

We are naturally interested in how good our face detector is at detecting faces, and how fast it is at doing so. Since we use the same cascade as OpenCV, we would expect to see somewhat similar detection success rates. However, there are certain differences in the implementations, with implications both for detection and speed. OpenCV employs a `ystep` optimisation, which causes every odd row and column to be skipped if the scale is less than 2, and to skip an additional step in the x direction when a face is not found, leading to more than a quartering of the work done for these scales, and altering which windows are detected. However, for the purposes of comparison, we patch OpenCV to remove this ystep, so that it then gives identical results to our implementation for the base scale, which gives us some assurance that our implementation is correct. However, as mentioned before, we do scaling in a different way from OpenCV, since we scale the detector and they scale the image. Hence, above the base scale, we get different output, and we would expect our detector to be less reliable than OpenCV for non-integer scales since we are not correcting for the imprecision caused by rounding non-integer coordinates.

In order to formally test our classifier's accuracy, we use the FDDB (Jain & Learned-Miller, 2010) benchmark, which contains 2845 images with 5171 faces. These are images from news articles containing all sorts of orientations of faces, so we would not expect a very high accuracy, however we do not expect to be drastically worse than OpenCV. We run both our and OpenCV's detectors using the LBP cascade, and employ OpenCV's rectangle clustering function (which has been ported to JavaScript in jsfeat) to group the subsequent detected rectangles. The number of neighbours (near-overlapping rectangles) at a certain position gives us a measure of confidence that a region is a face, since windows slightly off- centre from a face will still give a positive result.

We use this number of neighbours to plot a Region of Confidence with FDDB, Figure 32. This shows that indeed our accuracy is somewhat less than that of



```
> fd.benchmark16Lookups(100, "benchmark16LookupsVaryings")
  3.52
> fd.benchmark16Lookups(100, "benchmark16Lookups")
  3.49
> fd.benchmark16Lookups(100, "benchmark16LookupsVaryings")
  3.41
> fd.benchmark16Lookups(100, "benchmark16Lookups")
  3.55
>
```

Figure 31: Calculating offsets for texture lookup with varyings in the vertex shader (green) vs fragment shader (red) gives no difference in timing, around 3.5ms in each case. The output values are simply the sum of all texture values mod 255, giving not very meaningful output, but showing that identical values are computed

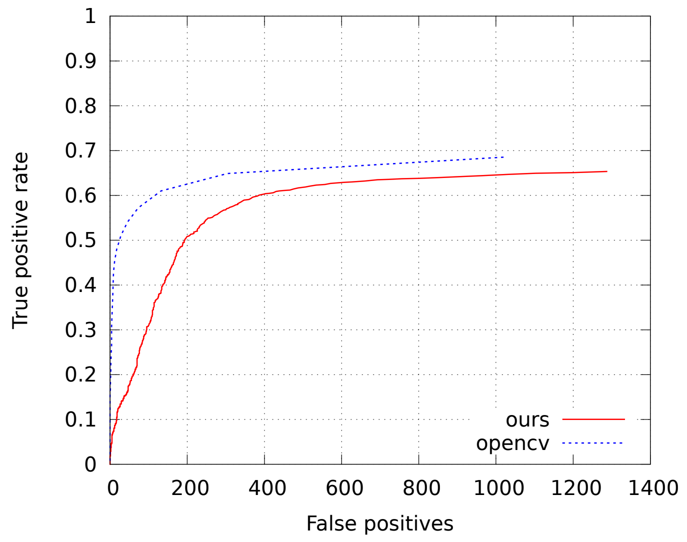


Figure 32: Region of confidence of our classifier vs OpenCV

OpenCV, in particular we have a relatively large number of false positives, so to get up to 50% detected in the dataset we have to put up with 200 false positives, and OpenCV outperforms us in number of true positives by quite a bit. We suspect that this is mainly due to the scaling method used, since it seems to lead to sporadic false positives rectangles. However, in practice this does not cause much problems, especially when using a webcam, since we can just set a relatively high threshold on the number of neighbours due to there being one face well centred in the scene.

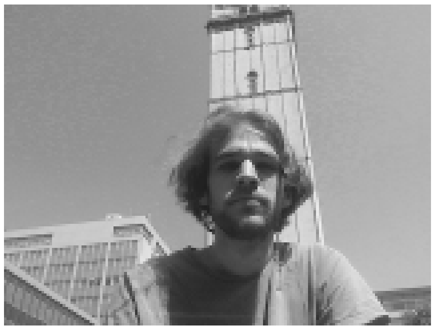


Figure 33: A typical webcam image

We use jsfeat (Zatepyakin, n.d.) as our baseline JavaScript implementation, for the purpose of comparing speed, which uses a Haar feature cascade. jsfeat also employs a (slightly different) step-skipping optimisation, so we time with and without this optimisation (we could of course modify our implementation to use this sort of optimisation, but it is easiest to compare if we know each detector is processing every window). We also show the times for OpenCV, which is a native C++ library. OpenCV can be compiled with Intel's Thread Building Blocks (TBB), which offers CPU parallel processing primitives which OpenCV uses to split the window computations up onto multiple CPU cores. We also show speed for a regular Release build of OpenCV without TBB, and a Release build with the `ystep` disabled. We run our implementation on an integrated Intel laptop graphics card, so the numbers are about what the average user could expect.

We take the average over 20 iterations, and with a scale factor of 1.2, using an image with one face, Figure 33, since we are mainly interested in the type of speeds we can get for tracking one face with a webcam. We run the detection for two resized versions of the image, at 320x240 and 160x120.

Detector	Variant	320x240 (times in ms)	160x120
jsfeat	regular	149.66	25.75
jsfeat	no ystep	667.15	98.15
opencv	with TBB	6.55	2.06
opencv	without TBB	11.54	2.46
opencv	no ystep,no TBB	32.62	5.96
ours	no optimisation	74.36	16.53
ours	z-culling	71.93	15.15

We see that we do better than jsfeat, even when jsfeat is skipping pixels. In particular, the unoptimised jsfeat takes 667 ms for a 320x240 image, whereas we can do it in 72ms. For the 160x120 images, the times our detector gets would let us get 60 frames a second, or give us time to do other tasks and still reach an acceptable framerate. However OpenCV still comes out on top, showing that native code still has the upper hand.

4 Application for Head Tracking

4.1 Tracking

We can use the detected face from the webcam in order to control the virtual camera in a 3D scene, moving the camera opposite the head motion, so that it appears like we are looking through a window into a 3D world, as in figure 34. To get the most likely face out of multiple detections, we use the jsfeat clustering function, and select the rectangle with the most neighbours. We then use the x and y position of the rectangle to control our camera, mapping the motion relative to the video size to a user-specified range that we offset the camera by. To control the depth, we use the size of the window, normalised by the width of the webcam video.

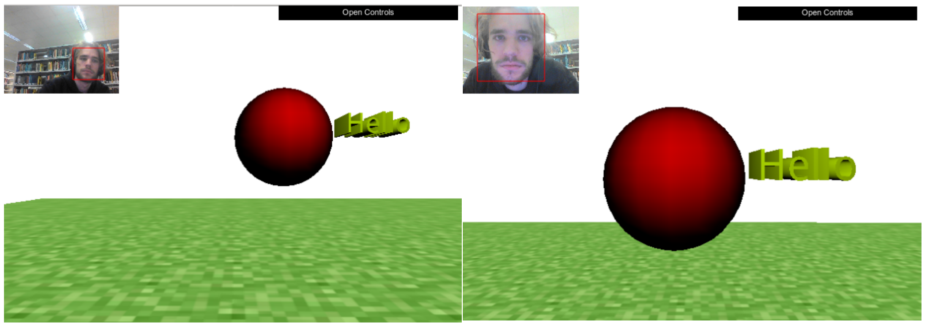


Figure 34: Using head motion to control a virtual camera, such that the scene acts like looking through a window

Our head tracking gives a fast, responsive result allowing the user to quickly observe the 3D scene from different angles. However it suffers from the problem of jitter in the camera position, due to small shifts in the precise pixel location of the detected face, even when the head remains fairly stationary. These noisy measurements are an unavoidable aspect of our detection, but we can employ some filtering to get a smoother result.

4.2 Kalman Filter

The Kalman filter is a Bayesian model which can be used to estimate the dynamics of a system, with a linear Gaussian transition. The typical example is that of tracking blips on a radar, estimating position and velocity from these noisy observations. The Kalman filter consists of determining the mean and covariance of a gaussian distribution through a cycle of measurements, updates, and predictions.

We shall start with a simple 1D example.

Assume we have a prior $P(x_t) = N(x_t; \mu_t, \sigma_t^2)$. The mean and variance for the prediction at the next time step, $P(x_{t+1})$ are given by the rules:

$\mu_{pred} = \mu_t + \mu_{motion}$

$\sigma_{pred}^2 = \sigma_t^2 + \sigma_{motion}^2$

Where μ_{motion} lets us specify some external motion in the system (but we can simply set it to zero) and σ_{motion}^2 (the motion noise, or transition variance) is some constant variance specifying our uncertainty in the motion, ie how unpredictable we expect it to be.

We then make a measurement, z and consider the posterior distribution

$P(x_{t+1}|z)$

The posterior mean and variance are given by the Kalman update step:

$\mu_{t+1} = \frac{\sigma^2_{pred} z + \sigma^2_{measure} \mu_{pred}}{\sigma^2_{pred} + \sigma^2_{measure}}$

$\sigma^2_{t+1} = \frac{\sigma^2_{measure} \sigma^2_{pred}}{\sigma^2_{measure} + \sigma^2_{pred}}$

Where $\sigma^2_{measure}$ is our measurement noise, a constant variance giving our uncertainty in the measurement.

Russel & Norvig (2010, chap.15 p. 587) explain how σ^2_{motion} and $\sigma^2_{measure}$ control the tradeoff between our predicted and measured values. The update μ_{t+1} can be seen as just a weighted average of the prediction μ_{pred} and the measurement z, with the two variances giving the weights. If we are not very confident in the measurement, then $\sigma^2_{measure}$ will be large and we will prefer the predicted value. However if we doubt the old mean (high σ^2_t) or our motion is unpredictable (high σ^2_{motion}) then σ^2_{pred} will be large so we prefer the measurement z. This effect can be seen graphically in Figures 35 and 36.

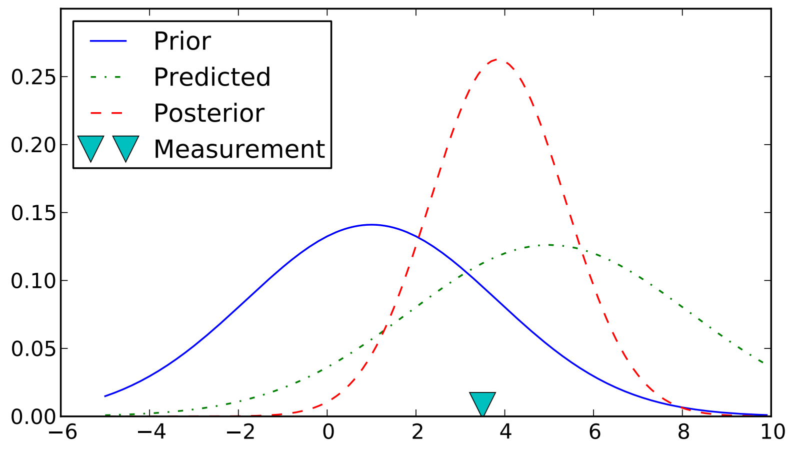


Figure 35: The update cycle for 1D Kalman with prior mean 1 and variance 8, and external motion shifting the prediction mean right by 4. Motion noise $\sigma^2_{motion} = 2$, measurement noise $\sigma^2_{measure} = 3$ and the measured value is 3.5. Notice how the width of the posterior decreases, indicating an increase in certainty. The mean of the posterior is 3.85, slightly to the right of the measured value, due to the weighted average effect between the prior and prediction.

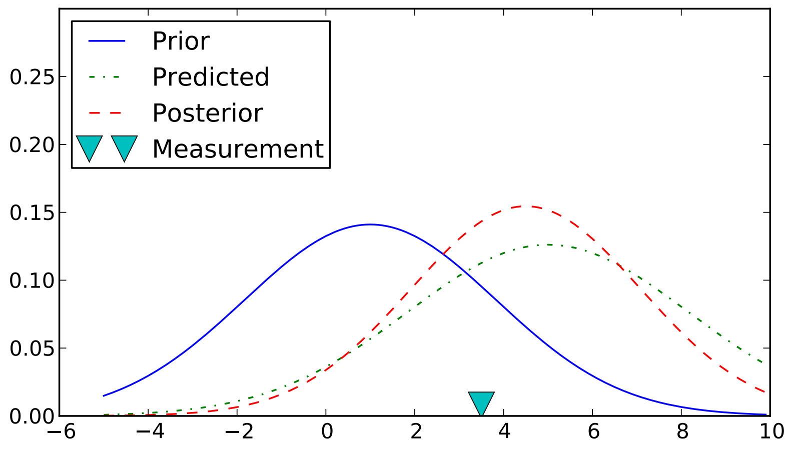


Figure 36: Changing the measure noise, $\sigma^2_{measure}$ from 3 to 20 has the effect of pulling the posterior towards the predicted value and away from the unreliable measurement.

The general, multivariate case of the Kalman filter is more complicated, involving some intimidating linear algebra, which we will not explain in detail, but only give a brief overview. For 2D motion, our mean now becomes a state containing the positions and velocities in each direction, $x = (X, Y, dX, dY)^T$, and our variance is now a covariance matrix P. The motion noise and measurement noise are now matrices Q and R, for which we can specify values along the diagonal giving the noise for each variable.

- The prediction step:

  New State $x' = Fx + u_{extmotion}$

  New Covariance $P' = FPF^T + Q$

- Update with measurement z:

  Innovation $y = z - Hx$

  Residual covariance $S = HPH^T + R$

  Kalman Gain $K = PH^T S^{-1}$

  New State $x' = x + Ky$

  New Covariance $P' = (I - KH)P$

The matrix F is our state transition function, giving a linear transformation of our state x. For calculating the 2D dynamics we want to set F to

```
F = 1 0 1 0
    0 1 0 1
    0 0 1 0
    0 0 1 0
```

Which means `x<-Fx` causes the update:

```
X  <- X + dX
Y  <- Y + dY
dX <- dX
dY <- dY
```

The matrix H is the measurement function, which "picks out" the measured values from the state.

```
H = 1 0 0 0
    0 1 0 0
```

Where $H \times (X, Y, dX, dY)^T = (X, Y)^T$, selecting the two positions.

In order to apply the Kalman filter for reducing the noise in measured face positions, we implement a Kalman filter in JavaScript, using the Sylvester matrix library. To demonstrate this we apply the filter to mouse clicks on a webpage, shown in Figure 37.

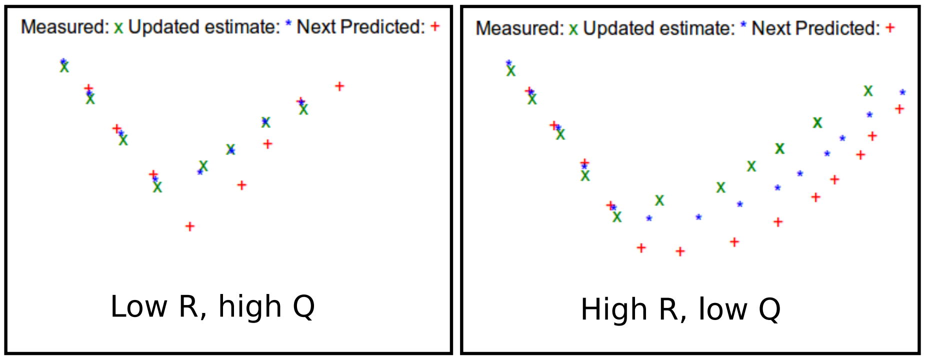


Figure 37: Kalman filter in Javascript, tracking mouse clicks. The lower R (measurement noise) and higher Q (motion noise) causes there to be more certainty in the measured clicks, so the updated states remain near the clicked position. For the opposite case, the estimates are "pulled" towards the prediction.

We use a Kalman filter to track the 3D dynamics of the face, where the Z coordinate is given by the width of the window. The result is that the camera movements in the 3D scene are a lot less subject to noise. We can tweak the parameters R and Q to determine the extent of this effect. Setting a low Q means that the motion is smoother, but the virtual camera reacts less quickly to sudden movements, and with a very low value it can feel like your head is "pulling" the camera around with a piece of elastic. A high Q means it is more responsive, but there is a perceptible "jiggle" in the camera position caused by the noise.

When we fail to detect the face in the image, we can skip the Kalman update from a measurement and just use the predicted position. This allows the camera to continue moving at the same constant speed. We do this for 10 frames before giving up, in case the face reappears, so that the camera motion is not interrupted if we lose the face for a couple of frames.

5 Conclusions and Further Work

We hope that this work shows the viability of WebGL as a platform for computer vision and other general purpose computational techniques. The performance gains achieved over plain JavaScript show that, despite the sometimes arduous task of transforming algorithms to fit within the WebGL framework, when speed is essential WebGL is the only widely available choice without abandoning the convenience of instant delivery through the web browser. However we believe there are opportunities for simplifying the experience of programming with WebGL, and we see interesting opportunities in providing abstraction libraries or cross-compilers that allow expressing computational concepts in a more natural way, for example a CUDA-like language that can be compiled to WebGL. One challenge is the inherent statefulness of WebGL. A big difficulty in writing modular components is the need to account for the global state, such as the currently enabled modes like depth testing and blending. A subroutine has no guarantee about what will happen when attempting to draw something, and the caller cannot rely on the state being the same after the subroutine is called. While it is tempting to hide all this messiness away through abstractions, there is a tradeoff between convenience and the ability to use the native API for WebGL specific optimisations.

With regards to the face detection implementation described, we see the possibility of making it more portable, such that it could be used on a wider range of devices. Firstly there are considerations which would need to be taken to make it run well on certain browsers in Windows, such as Chrome, since these use a wrapper layer called ANGLE which translates OpenGL calls into DirectX, which carries some overhead and is not completely compatible with OpenGL. Beyond that, many mobile devices and mini computers such as the Raspberry Pi have quite capable GPUs running OpenGL ES. To adapt our implementation to these we would have to work with less generous limits of numerical precision than on a full computer, and would be unlikely to have the convenience of floating point textures, so would have to adapt accordingly, such as packing the integral image values into the bytes of an RGBA texture. WebGL itself is available on mobile devices in the experimental version of the Chrome browser, and being able to perform computer vision within the browser on a mobile device would offer exciting possibilities, especially when combined with access to the motion sensors or GPS.

Our treatment of face detection has focused mainly on the use of a pre- existing cascade, essentially shoehorning it to fit our needs on the GPU. An alternative approach would be to create a purpose-built cascade, having characteristics that make it intentionally well suited to the GPU. In particular, given the bottleneck posed by the texture lookups, we would ideally like features that are "GPU friendly". In @ZhangZhang, one of the methods presented to speed up detection is the feature-centric (rather than window-centric) cascade, due to Schneiderman (2004). The idea is that, at least in the early stages, we prefer features that are easy to compute over features that are "best" for detecting faces, so that for the first stage all our features in the image can be computed as a grid, for example at the pixel level, and multiple windows then share the same features. We believe that similar features could be adapted quite well to the GPU, since we can do grid- style computation easily with the fragment shader, and it allows us to benefit from contiguous texture accesses.

Finally, we speculate on further applications for face detection in the browser. The tracking implemetation described would work well for the purposes of 3D games, for example allowing the player to move his head to glimpse around walls. Augmented reality games involving interacting with virtual objects would also be possible, such as keeping a virtual ball in the air. It could also be adapted to interactive media and photography, for example to observe the perspective-shift photography offered by light field cameras such as Lytro. With some additional work, it would be feasible to implement some (approximate) form of eye tracking, for example with gaussian processes as in Nel (2009), which could be used to track the user's gaze on the screen, for example to conduct tests that collect a heatmap of where participants look on a web page, or interactive HDR (High Dynamic Range) photography where the tone mapping is adjusted according to the intesity of the area the user is looking at. The use cases need not be limited to the user's webcam, since we can perform face detection on any image or video we can embed in a webpage, for example it could be used to zoom in on faces in a streaming video, to crop a video chat to a smaller region containing just the face, or to tag faces in photos in the browser before uploading to a photo website.

6 Appendix

6.1 Coordinate Considerations

Working with the OpenGL ecosystem invariably requires an understanding of the different coordinate systems used for the polygon vertices, textures and screen. This becomes even more important when using OpenGL for computation, as being off by one pixel (or a fraction of a pixel) can have much more serious consequences than mere graphical glitches. If a texture is used as a lookup table for arbitrary information, it is essential that the correct values are indexed, to avoid giving, at best, completely incorrect results, or at worst hard-to-detect bugs due to the limitations of floating point precision in the 0 to 1 range used to index textures.

Firstly we have the window (or screen) coordinates, which give the position in the viewport, in other words the final image output. However, since the output may be rendered to a texture, window coordinates don't have to be related to an image actually displayed on the screen. They are similar to pixel positions, however OpenGL itself does not have a concept of a pixel until rasterisation. Peers (2002) gives a detailed mathematical treatment of OpenGL coordinates, drawing from the OpenGL specification. In this way, the viewport can be treated as a Cartesian plane whose origin and unit vectors are given by the `gl.viewport(x,y,w,h)` command. This sets the x,y offset of the origin, which is at the bottom-left edge of the image, and determines the area of the scene which should be rasterised, so in a graphics sense can be considered a sort of cropping of the image. Two important points to note here are that the Y axis is effectively flipped relative to the coordinate system usually used in graphics, which has the origin at the top-left, and that integer coordinates will index the bottom left corners of pixels, so to index the centre of a pixel requires adding 0.5 to each dimension. For general purpose computation on a grid, modifying the viewport can be used to change the output range of the computation. For example, when doing face detection at different scales, the "sliding window" of the detector will change size, meaning less pixel positions need to be considered for larger windows, so the size of the output grid should be smaller.

The vertex positions of polygons are specified by setting the `gl_Position` variable in the vertex shader. This is a four dimensional `(x,y,z,w)` vector where x,y,z are in Normalised Device Coordinates, a resolution-independent coordinate system which varies from -1 to 1 in each dimension such that the origin is at the centre. These then undergo perspective division by the fourth `gl_Position.w` coordinate. For convenience we can use window coordinates when we supply the vertices as an attribute, then compute the normalised coordinates in the vertex shader by dividing by the image resolution. This will give a value in the range [0,1], which can be changed to the range [-1,1] by multiplying by 2 then subtracting 1. For the purposes of computation on a 2D grid, the only geometry we need is a rectangle aligned with the viewport, which we can get by drawing

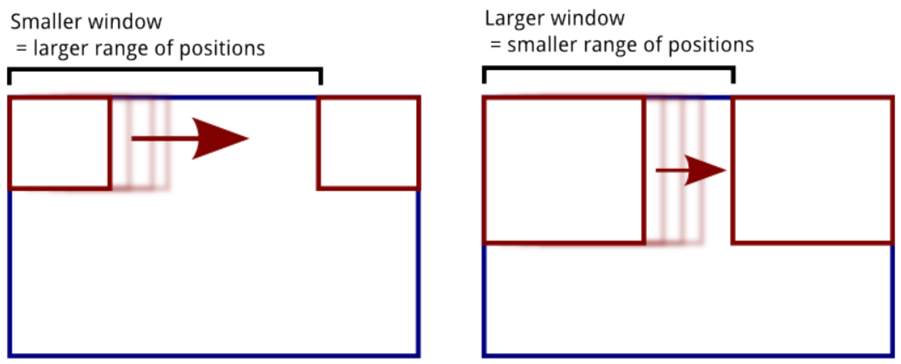


Figure 38: Decreased output range for larger window

two triangles. We do not want any perspective division, so z,w can be set to 0,1. This effectively "passes through" the vertex coordinates, allowing us to use them as if they were window coordinates.

The shader code to achieve this is: (where aPosition is the vertex position attribute and uResolution gives the image resolution)

```
vec2 normCoords = ((aPosition/uResolution) * 2.0) - 1.0;
gl_Position = vec4(normCoords, 0, 1);
```

Finally, we have to deal with the coordinates of texture maps, made up of texels (the texture equivalent of a pixel) which are sampled using texture2D() in the fragment shader. They have coordinates from 0,0 in the bottom left to 1,1 in the top right. Textures may be sampled using different filtering methods in order to interpolate between the discrete texels, the simplest being "NEAREST" which simply uses the closest texel value, and "LINEAR" which interpolates linearly based on the distance to surrounding texels. To sample at precisely the texel centre, with no filtering, it is necessary to offset by half a texel, since the "zero" of a texel is at the bottom left corner. So for the ith texel in a row we would use X coordinate `(i + 0.5)/width` to offset then normalise to the [0,1) range.

6.2 The Dangers of Branching

Branching within the shader, while possible through the use of if-else statements, carries with it numerous caveats, explained in Harris & Buck (2005). In the "olden days" (circa 2003) in order to emulate branching, GPUs would simply evaluate both sides of the condition, then determine which result to use before writing the output. This meant that the time would be proportional to the cumulative cost of both branches.

Things got better with the SIMD (Single Instruction, Multiple Data) model, which uses multiprocessors executing the same instruction on many data elements at once. In this case, the GPU will not be doing useless work evaluating both sides of the condition, but instead divergent branches will cause a stall, where the processors that do not take a branch have to wait for the branching processors to catch up. In the worst case this will still take as long as both branches combined, but in the case where all processors take the same branch (known as coherency) it will be more efficient, and since the allocation of fragments to processors is often done in a spacially localised manner, it allows for speedups when fragments in the same area of an image branch in the same way.

Finally, true dynamic branching may be available in the form of MIMD (Multiple Instructions, Multiple Data) where different processors may execute different instructions simultaneously. Most modern GPUs support dynamic branching to some extent (NVIDIA's GeForce 6 series, released in 2005, introduced MIMD branching in the fragment shader) however at an architectural level branching still presents a barrier to efficient parallel computation, since knowing that all fragments will follow the same instructions gives the GPU opportunities for optimisation.

For this reason, branching in the shader should be kept to a minimum, and it is preferred for algorithms to be structured such that fragments in the same neighbourhood take the same branches in order to maximise coherency. Especially in the case of WebGL, the programmer has no control over what graphics card capability the user will have, and is unable to query information about the graphics card due to security restrictions, so it is usually best to program for the lowest common denominator. However there will always be cases where it is faster to branch than not to branch, and there is no substitute for careful testing to determine if a branch in fact gives a penalty.

7 References

Arthur, H. (n.d.) *Kittydar*. [Online]. Available from: http://harthur.github.com/kittydar/ [Accessed: 16th January 2013].

Buck, I. (2005) Taking the Plunge into GPU Computing. In: Matt Pharr (ed.). *GPU Gems 2*. Addison Wesley. p. 509.

Consortium, T.C. (n.d.) *WebGL GPGPU experiment - reading a floating point texture*. [Online]. Available from: http://lab.dev.concord.org/experiments/webgl-gpgpu/webgl.html [Accessed: 16th January 2013].

Google (n.d.) *Google+ API*. [Online]. Available from: https://developers.google.com/+/hangouts/api/ [Accessed: 16th January 2013].

Harris, M. (2005) Mapping Computational Concepts to GPUs. In: Matt Pharr (ed.). *GPU Gems 2*. Addison Wesley. p. 493.

Harris, M. & Buck, I. (2005) GPU Flow-Control Idioms. In: Matt Pharr (ed.). *GPU Gems 2*. Addison Wesley. p. 547.

Jain, V. & Learned-Miller, E. (2010) *FDDB: A Benchmark for Face Detection in Unconstrained Settings*.

Jave (2011) *Is discard bad for program performance in OpenGL? - Stack Overflow*. [Online]. Available from: http://stackoverflow.com/questions/8509051/is-discard-bad-for-program-performance-in-opengl [Accessed: 27th May 2013].

Khronos (2012a) *WebGL_color_buffer_float Extension Draft Specification*. [Online]. Available from: http://www.khronos.org/registry/webgl/extensions/WEBGL_color_buffer_float/ [Accessed: 16th January 2013].

Khronos (2012b) *WebGL_color_buffer_half_float Extension Draft Specification*. [Online]. Available from: http://www.khronos.org/registry/webgl/extensions/EXT_color_buffer_half_floa [Accessed: 16th January 2013].

Krumlinde, V. (2011) *GLSL: passing a list of values to fragment shader*. [Online]. Available from: http://stackoverflow.com/questions/7954927/glsl-passing-a-list-of-values-to-fragment-shader [Accessed: 16th June 2013].

Larson, B. (n.d.) *GPUImage*. [Online]. Available from: https://github.com/BradLarson/GPUImage.

Lienhart, R. & Maydt, J. (2002) An extended set of Haar-like features for rapid object detection. *ICIP02*. 900–903.

Liu, L. (n.d.) *A Not-so-slow JavaScript Face Detector*. [Online]. Available from: http://liuliu.me/ccv/js/nss/ [Accessed: 16th January 2013].

Molchanov, A. (n.d.) *flashopencv*. [Online]. Available from: https://github.com/bonext/flash-opencv [Accessed: 16th January 2013].

Nel, E.-M. (2009) *Opengazer*. [Online]. Available from: http://www.inference.phy.cam.ac.uk/opengazer/.

Obukhov, A. (2011) Haar Classifiers for Object Detection with CUDA. In: Wen-Mei Hwu (ed.). *GPU Computing Gems*. Morgan Kaufmann. p. 517.

Ojala, T., Pietikhenl, M., Harwood, D. & Measures, L.T. (1994) Performance evaluation of texture measures with classification based on Kullback discrimination of distributions. *ICPR*. 582–585.

Peers, B. (2002) *OpenGL pixel and texel placement*. [Online] Available from: http://bpeers.com/articles/glpixel/.

Russel, S. & Norvig, P. (2010) *Artificial Intelligence A Modern Approach*. New Jersey, Pearson.

Schneiderman, H. (2004) Feature-Centric Evaluation for Efficient Cascaded Object Detection. *CVPR*.

Tavares, G. (2011a) *Google I/O 2011: WebGL Techniques and Performance*. [Online]. Available from: http://www.youtube.com/watch?v=rfQ8rKGTVlg [Accessed: 27 May 2013].

Tavares, G. (2011b) *WebGL Fundamentals (WebGL is a 2D API!)*. [Online]. Available from: http://games.greggman.com/game/webgl-fundamentals/ [Accessed: 19th May 2013].

Tavares, G. (2012) *WebGL How It Works*. [Online]. Available from: http://greggman.github.io/webgl-fundamentals/webgl/lessons/webgl-how-it-works.html [Accessed: 19th May 2013].

Tavares, G. (2011c) *WebGL Image Processing*. [Online]. Available from: http://games.greggman.com/game/webgl-image-processing/ [Accessed: 19th May 2013].

Thewlis, J. (2012) *Face Detection in Video for Digital Product Placement*. Industrial Placement at MirriAd Ltd.

Thomas, G. (2009) *WebGL Lesson 8 – the depth buffer, transparency and blending*. [Online]. Available from: http://learningwebgl.com/blog/?p=859 [Accessed: 27th May 2013].

Viola, P. & Jones, M. (2001) Rapid object detection using a boosted cascade of simple features. *CVPR*.

Wallace, E. (n.d.) *WebGL Water*. [Online] Available from: http://madebyevan.com/webgl-water/.

WebGLStats (2013) *WebGL Stats*. [Online]. Available from: http://webglstats.com/ [Accessed: 16th June 2013].

Woolley, C. (2005) GPU Program Optimization. In: Matt Pharr (ed.). *GPU Gems 2*. Addison Wesley. p. 557.

Zatepyakin, E. (n.d.) *jsfeat*. [Online]. Available from: http://inspirit.github.com/jsfeat/ [Accessed: 17th June 2013].

Zhang, L., Chu, R., Xiang, S., Liao, S., et al. (2007) Face Detection Based on Multi-Block LBP Representation. *Advances in Biometrics*. 11–18.

Zhenyao Mo, K.R. (2012) *Graphics Programming for the Web: WebGL*. [Online]. Available from: http://goo.gl/kQirL [Accessed: 19th May 2013].