Introduction to WebCL And Influence Map Demonstration

OpenCL is a framework allowing programmers the ability to write code that executes across a number of heterogeneous devices including CPUs, GPUs, DSPs, and other specialized hardware. Written in OpenCL C, a language based on the C99 standard, OpenCL programs utilize parallel devices, such as your multi-core CPUs and GPUs, to accelerate parallelizable work. OpenCL is well suited for data parallelism but can also perform task parallel work.

WebCL allows web developers to access hardware devices in which they may use to accelerate their work by allowing OpenCL functions to be called into through Javascript. No browsers currently natively support WebCL, but a number of extensions exist for developers who wish to make use of OpenCL on the web. Nokia developed the extension used in this demonstration for use with Mozilla’s Firefox browser. It is yet to be seen if WebCL will take off and eventually become natively supported across all browsers, but as of now it is looking unlikely. Compute shaders, a similar framework to OpenCL (though more coupled with OpenGL) are currently receiving more attention and will likely be the first GPU compute framework to be supported on the web. Thankfully, OpenCL and Compute Shaders are very similar, and becoming familiar with one will help one learn other compute frameworks that much more quickly.

To make use of OpenCL, one must create a host application in a language that has access to the various data structures of OpenCL. There are five main structures, devices, contexts, command queues, kernels, and OpenCL programs. A device is directly related with a hardware device on the hosts’ platform, such as a CPU, GPU, or DSP. A device can be scheduled tasks, or kernels, by the host application via a command queue. Each command queue is unique to a device. Kernels are instantiated from kernel definitions that are found within a compiled OpenCL C program. Finally, a context is a collection of devices. Contexts are the bridge between the host application and the devices, as well as devices with one another. The host application accesses devices and their command queues through a context, and devices can transfer data between one another through contexts via a global memory space. This allows devices to work together to solve a tasks. Knowing which devices are best suited for a task, how to structure those tasks to make the best use of the hardware devices being targeted, and how to schedule tasks in way that is as non-blocking and scalable as possible will make all the difference when trying to squeeze the most performance out of one’s hardware.

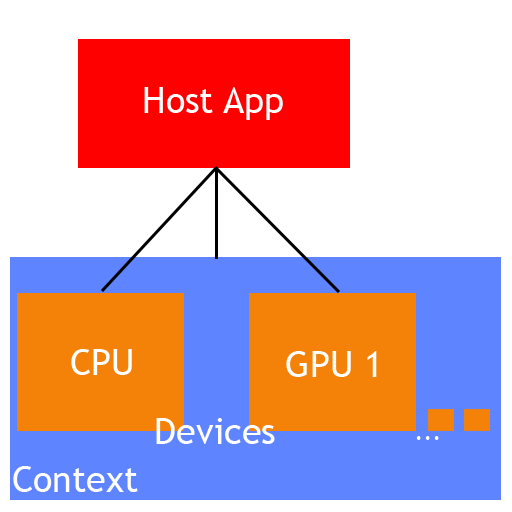


Figure : Devices exist on context(s) and the host app has access to them through the context.

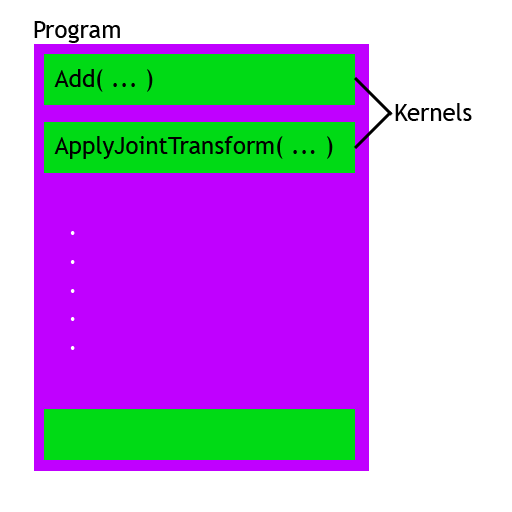


Figure : The layout of an OpenCL program and the kernel functions contained within.

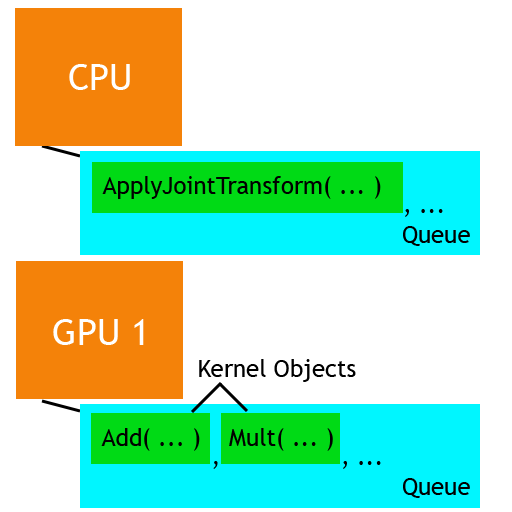


Figure : Devices with multiple kernel objects scheduled to them via command queues.

After creating a context, a queue for each device may be created. These queues are what the application uses to schedule kernel objects for the device to perform. Once a queue has been created, kernel objects can be inserted into the queue. kernel objects are sorted in the order they were entered into the queue. These tasks will be executed either in-order, or out of order. In the case of out of order execution, OpenCL chooses which task to execute based on the current resources available for a particular device, among other variables. In Figure 3, the Add kernel object was entered into GPU 1’s command queue prior to the Mult kernel object. If the queue was set to execute in order, Add will also be executed before Mult on GPU 1.

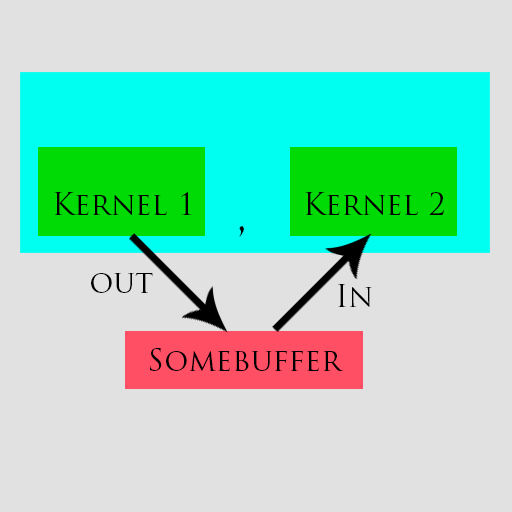


Figure : A dependency between tasks.

When an OpenCL queue is performing tasks out of order, the developer must be wary of any kernels that depend on the output of other kernels. If left unchecked, tasks whose kernels have dependencies may execute prior to the tasks whose output they depend on. For example, in Figure 4 Kernel 2 has input that is the output of Kernel 1. Thus, a dependency exists and one would need to be careful when queueing these two kernel objects on a queue that executes out of order. To resolve this issue, OpenCL offers a robust event system. Tasks can be made to wait on a number of other tasks to complete, and tasks may fire events when completed.

Kernel functions are C-like functions that perform their computation across a range of data in parallel. Most kernels have at least one parameter that is a readable or writable buffer that the kernel will be performing its computations on. Buffers can be one, two, or three-dimensional, and the user must specify the dimensions the kernel will execute on when enqueuing a task. In Figure 3, the parameter global\_dim is an example of a two-dimensional domain. This means the kernel is executing on a two dimensional (like an image) buffer. In Figure 5, imageWidth and imageHeight refer to range that the kernel executes on per dimension. With an imageWidth of 1920 and an imageHeight of 1080, the kernel will perform 1920 x 1080 computations.

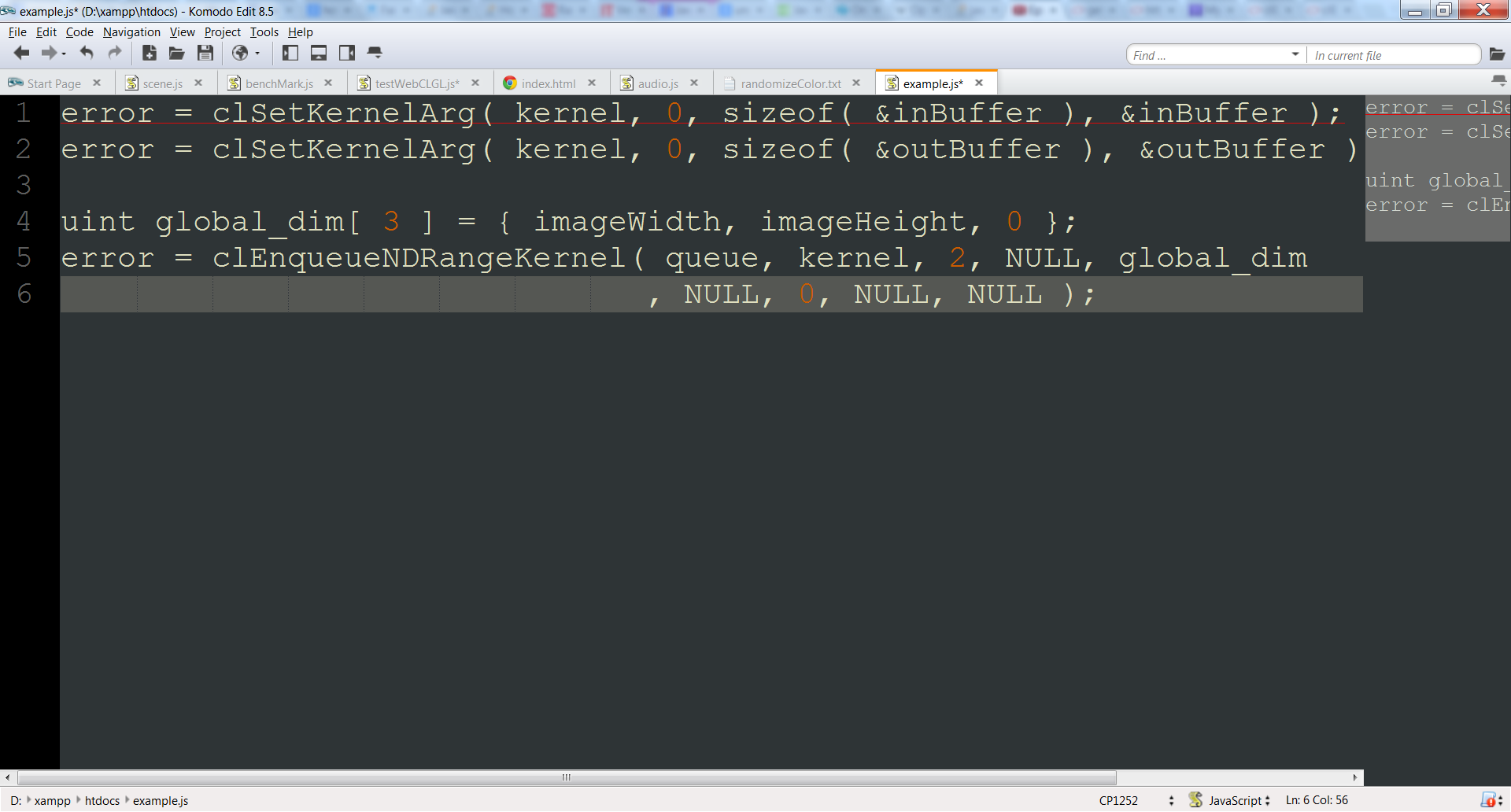


Figure : Setting the parameters of a kernel object and enqueueing a new task onto a queue.

Each computation in a kernel is called a work-item. Work-items can retrieve a global\_id value that refers to the indices to the buffer it is executing on. Work-items can access global memory, but this memory is not synchronized. This means that work-items cannot reliably share data with other work-items at a global level. Work-items can be part of a local work-group, however. In a local work-group, memory can be synchronized and shared by other work-items in that work-group. Work-items also have memory local to the scope of the work-item itself, and that memory is private to the work-item.

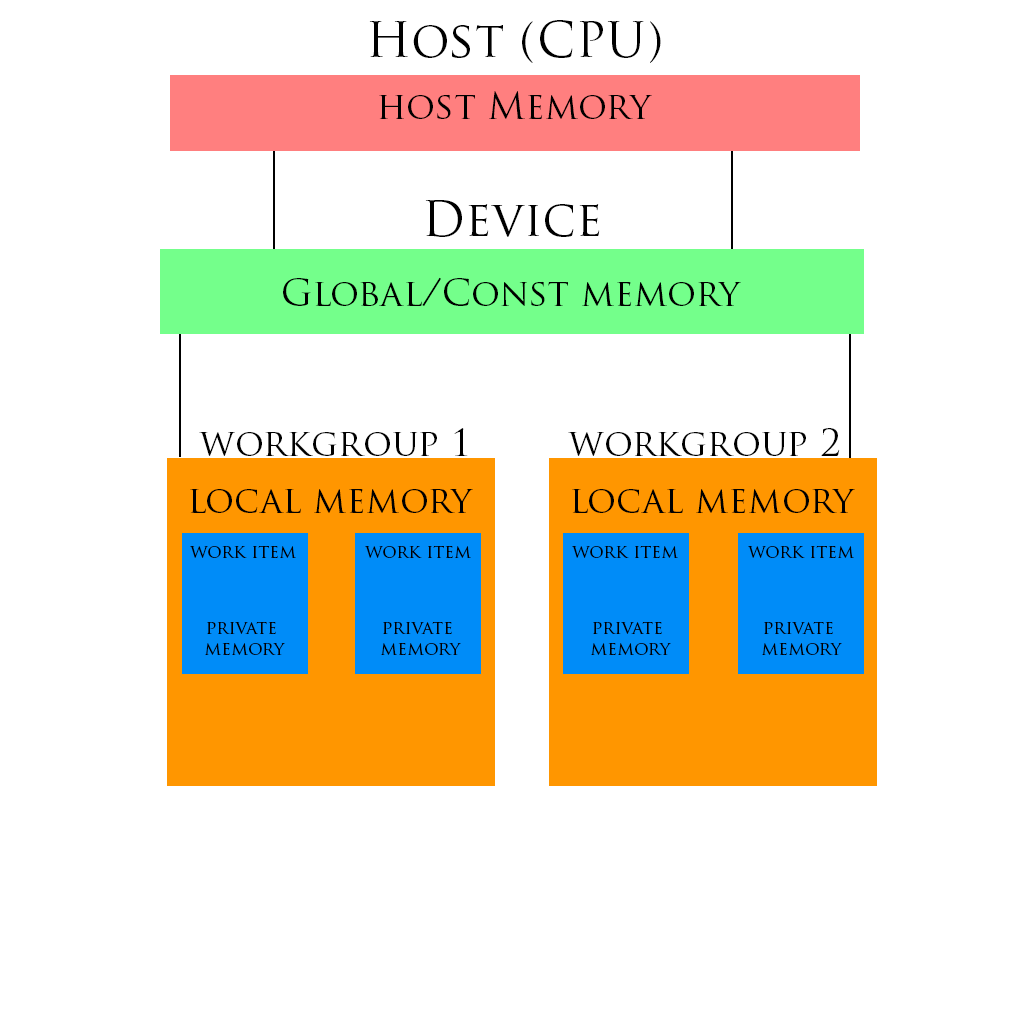


Figure : The OpenCL memory model.

To demonstrate WebCL, I have created an application that computes a very simple influence map for two competing factions. The computation for the influence map is done both in pure Javascript and as a kernel object to display the difference in performance. After completing this tutorial, one should be capable of creating their own WebCL applications with ease.

To help me break down the task at hand, I created a number of classes/modules. These classes mostly help me make the various parts of the application more modular and extendible, and do not need to be detailed fully for one to code this project. However, some names particular to these classes will pop up. I will try to make things as clear as possible while not straying from the core of our task.

Creating a simple canvas to display and debug our work is the first thing we’ll do.

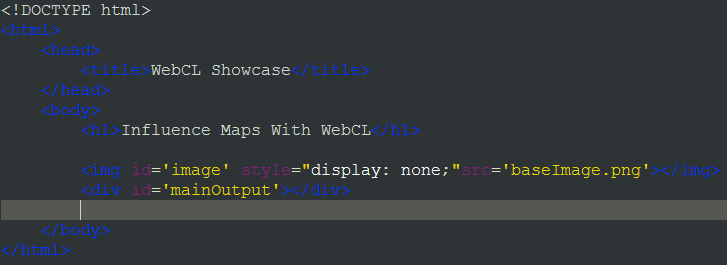


Figure : Creating a simple canvas for us to display our influence map onto.

In our html file we’ll create a div tag for our canvas to reside in. We’ll also load in a pure black image to use as a blank slate. I’m sure there is an easy way to create canvas and fill it with black pixels, but this worked fine for me. My image was 512 x 512 pixels.

Now that we have a place to plug our canvas into, we should create our canvas object in javascript and append it to our mainOutput divider object.

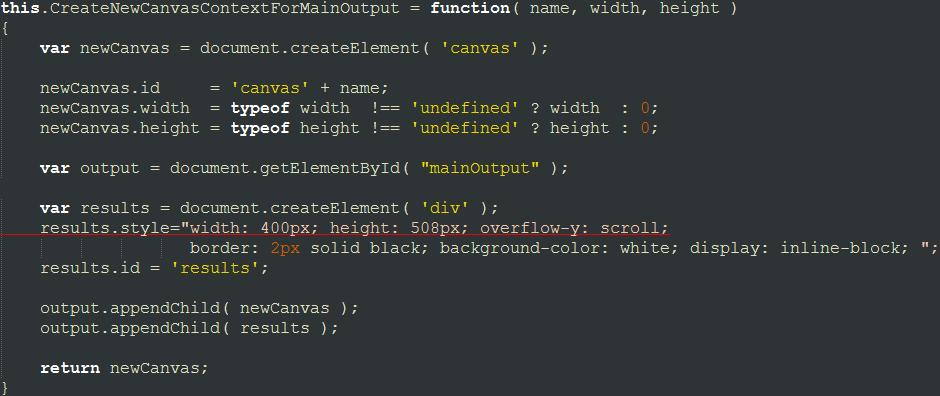


Figure : A canvas object creation function for displaying Influence Map.

I wrapped my canvas object creation into a function that would allow me to add multiple canvases easily if I wanted to. Note that the line that starts with results.style and the line after it should be one line. I just needed to make room so that it could fit in a screenshot. The results element is a divider that I have alongside my main canvas. It displays useful debug and performance information. Note that is should probably by separated into its own function, as we may find it annoying to have a large output box display to the right of each additional canvas we create.

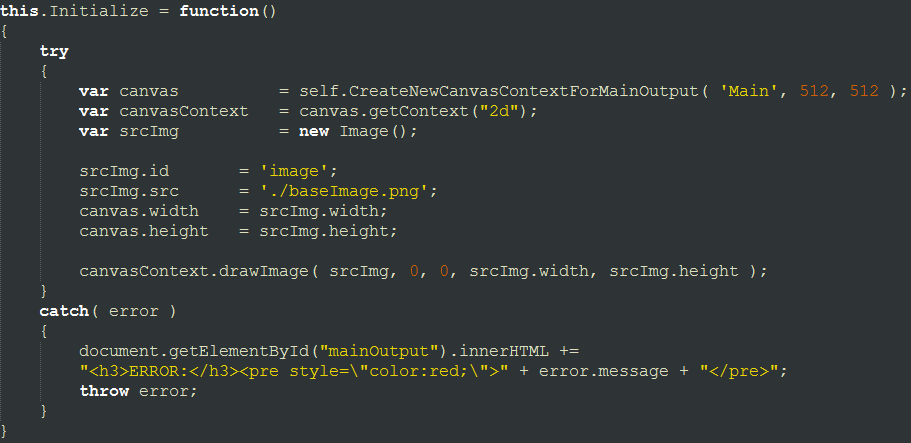


Figure : Creating main canvas and drawing blank slate image.

After creating our canvas object, we should set it up to render our blank slate image. In my initialize function, I actually invoke the function seen in Figure 8 to create my canvas object and then setup an image for it to display.

Next, we’ll want to set up our environment, or world, to accommodate the entities used for our influence map.

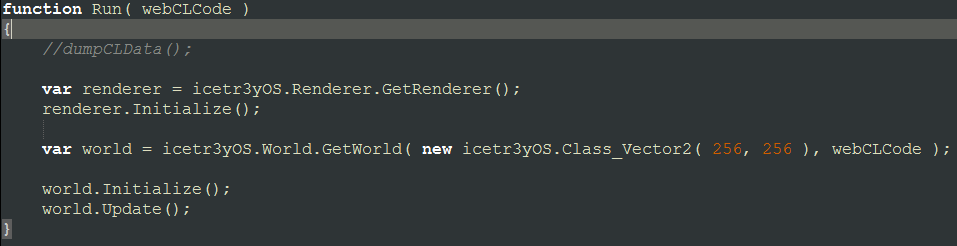


Figure : My "main" function.

This is actually my entry function into the application. world is the environment I’m creating for my entities to exist in. It is basically a two-dimensional grid, 256 units by 256 units, and hosts two factions of entities. The second parameter for my GetWorld function is the webCLCode for the WebCL program we’ll use for this tutorial. It will be covered later. The only other thing to note here is the Update function. This tutorial will not cover how to compute and render the influence map in real-time, but note that it is not much additional work to do so.

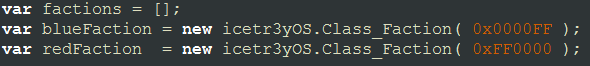
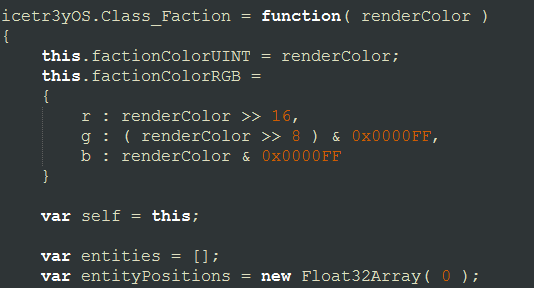


Figure : Faction creation

One of the first things I do in my world is create two factions of entities. All a faction is an array of entities with color that is used to for rendering purposes.



This is basically the faction class. There are a few convenience variables not listed, as well as functions for adding new entities to a faction and rendering factions separate from the influence map. Note that I have an array both for entities (which have position data already), as well as a float array for entity positions. This is redundant and should not be implemented as such, but as we’ll learn later WebCL wants typed arrays for its buffer data and this was an easy way to get entity positions onto my devices.

Entities themselves are simply position data (an x and y position within the confines of the world) and nothing more. In my implementation, I also have movement information and functionality incorporated into the entities, but it is not needed to compute an influence map and will be left out of this tutorial. If you wish you could forgo the idea of an entity entirely and simply keep a list of random positions per faction. The idea of a faction could also be stripped away. You could simply have two separate arrays of positions and use those as parameters to the following functions. I would suggest doing so for a quicker implementation and then once you have gotten something up and running you can abstract the data structures how you see fit.

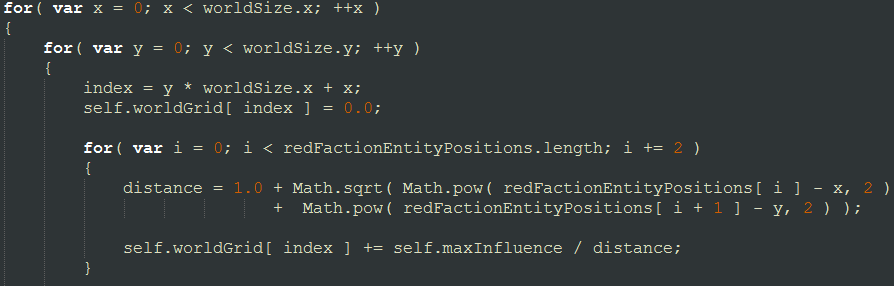


Figure :Influence Map calculation

Now for the meat and potatoes. We will first calculate our influence map sequentially in Javascript. We’ll do this by simply calculating the distance between each tile in our world and each entity, each time adding influence for that particular faction. For the red faction I am adding influence, but for the blue faction (not shown but it’s the same for loop as red’s but we subtract at the end instead of add), I am subtracting. Really, tiles with a net negative value are just more influenced by the blue faction than the red, and you can get the actual influence value of a the blue faction on a tile by taking its absolute value ( if it is negative to begin with ).

Entities that are further away from a tile project less influence on it, and entities from different factions negate each others’ influence on a tile. These gradients of black, red, and blue are the data the influence maps provide us and we can use it how we see fit.

For the WebCL version, let’s first go over some setup that needs to be performed before issuing kernels to our graphics cards.

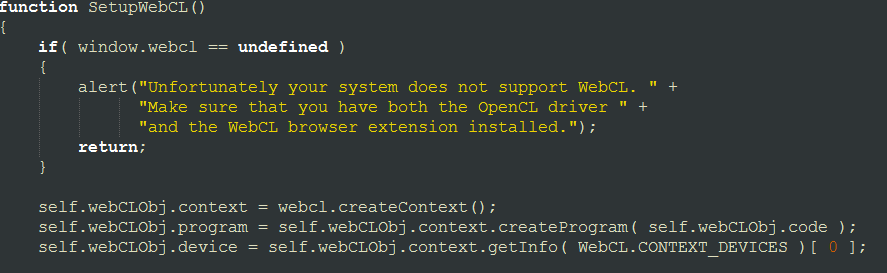


Figure : Creating context and program.

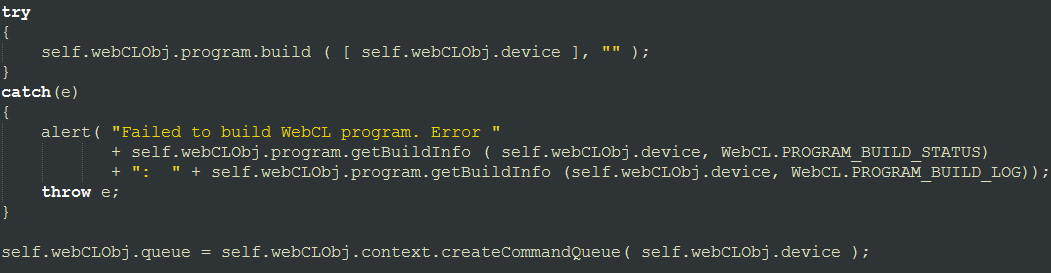


Figure : compiling program and creating command queue.

You can find some useful tutorials on Nokia’s own WebCL site: <http://webcl.nokiaresearch.com/> that sums up how to setup up WebCL for your application, but the basic gist is that we want to create a context with one or more devices in which we can queue kernel objects onto. We need to compile the program we’re interested in per device we want to use it on. If successful, we can then create a queue and start issuing tasks.

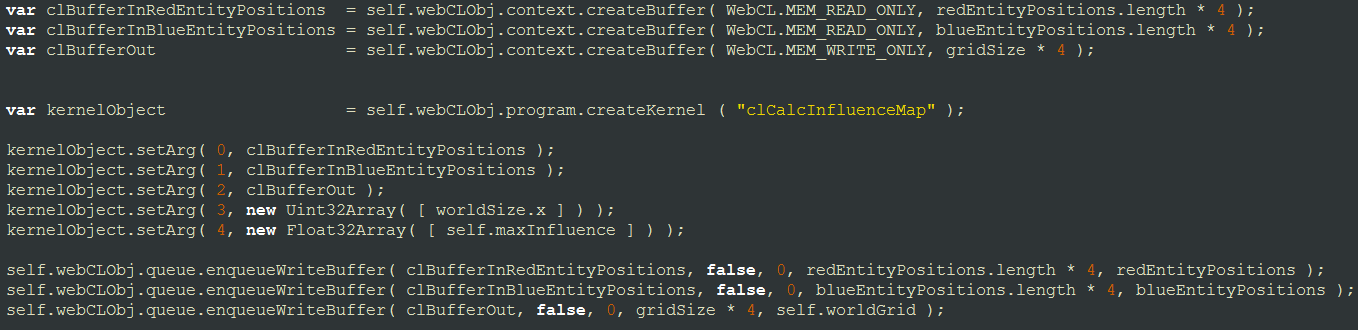


Figure : Creating our memory buffers for WebCL and kernel object

Now to actually calculating our influence map with WebCL. First, we will create buffers for our entity positions and our out buffer with the influence map data. We then create a kernel object from our WebCL program. After creating our kernel object, we set up its parameters. Then, we write our entity and grid data onto the WebCL buffers.

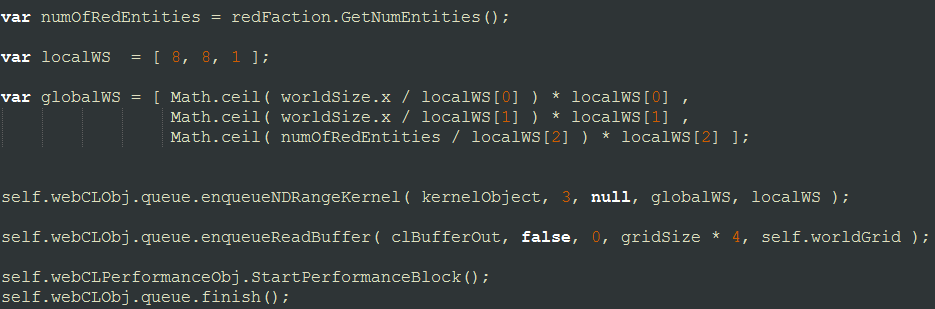


Figure : Queueing the Kernel Object and reading results into our out buffer.

We then create our local and group workspaces. Since this is a three dimensional problem ( two dimensional grid + entity positions ), we make three dimensional local and global workspaces. We then enqueue the kernel onto the device’s command queue and finally read the out buffer into our grid buffer.

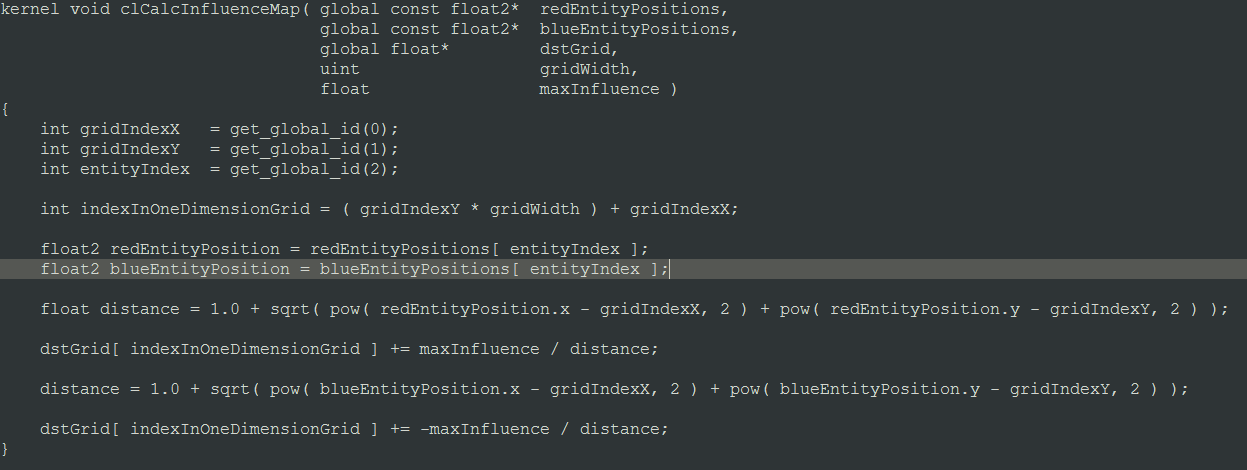


Figure : The WebCL kernel function used to calculate influence map

Finally, let’s look at the WebCL kernel function, clCalcInfluenceMap. We first get the indices for our particular work item. Each work item will execute in parallel, decreasing our computation time exponentially. We will calculate the same basic distance function here and add/subtract influence per tile.

You can then take the out buffer data and render it to a texture (the image I referenced in the beginning of this tutorial is what I used). If all goes well you should see something similar to this:

