GPU Computing in the browser, with WebGL

Emmanuel Leroy

{“Santa Cruz JS Meetup”: “August 2017”}

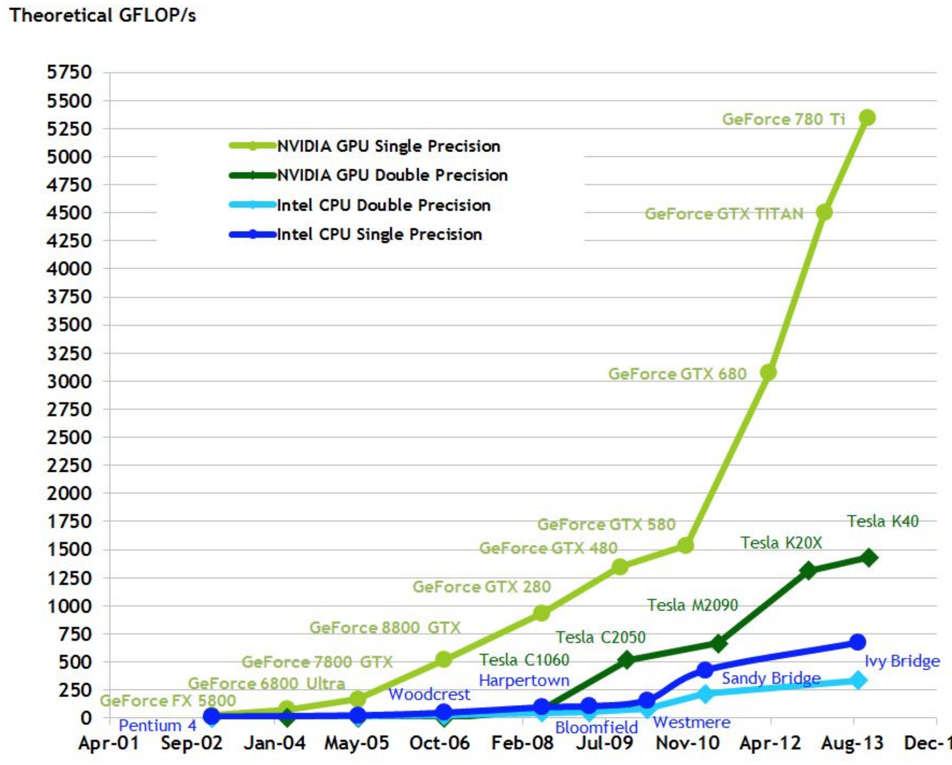
Overview

- Why GPU Computing?
- GPU pipeline basics.
- GPU Computing with limited H/W capabilities: how to.
- Implementing in the browser with WebGL.

Why GPU computing?

CPU vs. GPU

Performance over the years



Floating-Point Operations per Second - Nvidia CUDA C Programming Guide
Version 6.5 - 24/9/2014 - copyright Nvidia Corporation 2014

Multi-core CPU vs. many-core GPU

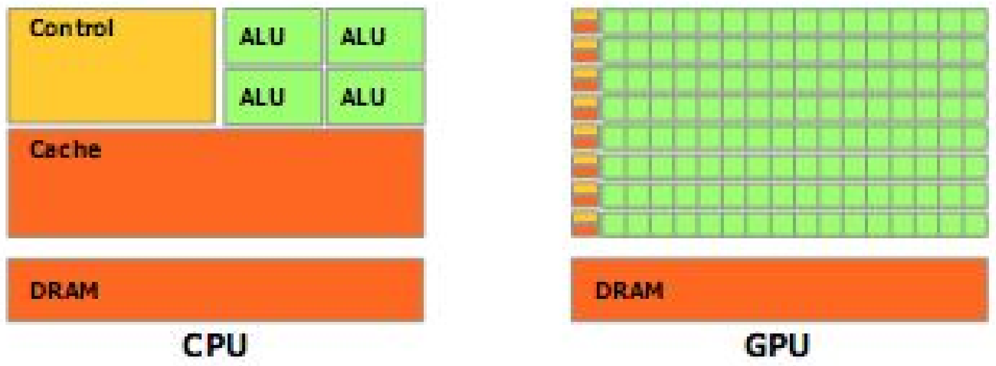
High-end modern CPU: **8-12** cores

High-end modern GPU: **2000-3000** cores

GPU trade-offs:

- higher parallelism, less flexibility.
- memory is fragmented:
- overhead: get data through the CP

GPUs are great for parallel computation

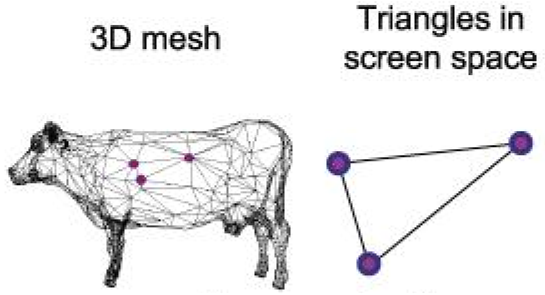


Relative Allocation of Transitors for Computation, Data Caching, and Flow Control - NVIDIA CUDA C Programming Guide - Version 4.2 - 4/5/2012 - copyright NVIDIA Corporation 2012

GPU Pipeline basics

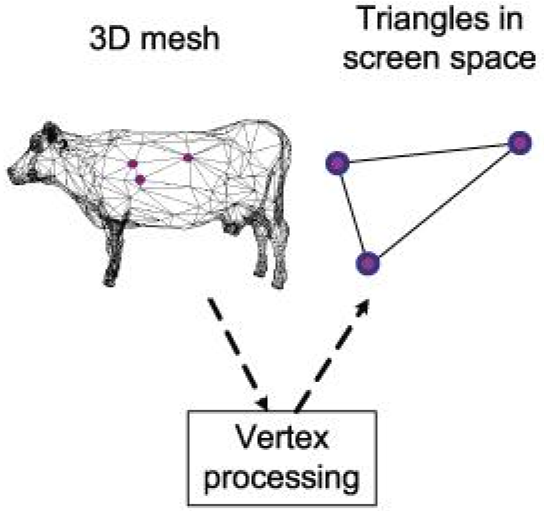
GPU Pipeline basics

- 3D Vertices coordinates define a model.
- A Model is made of triangles.



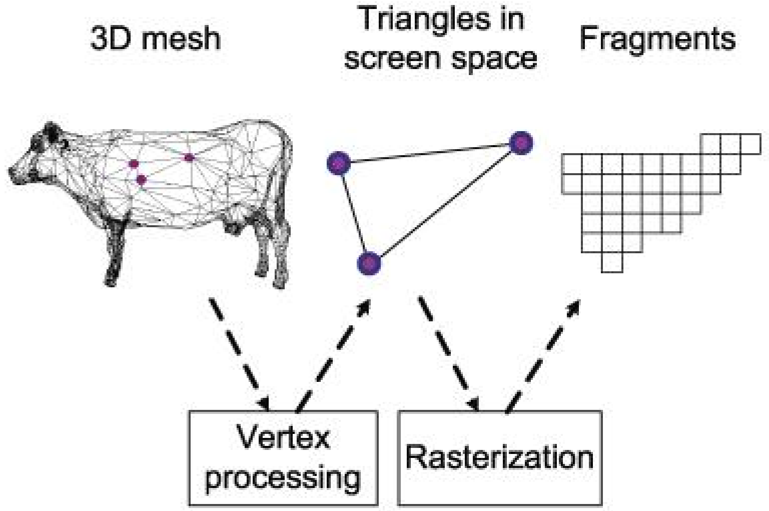
GPU Pipeline basics

- Vertex processing transforms 3D coordinates to 2D screen coordinates.
- The code processing *vertices* is called a **vertex shader**
- Vertex shaders can also alter the model shape (bump, animate…)



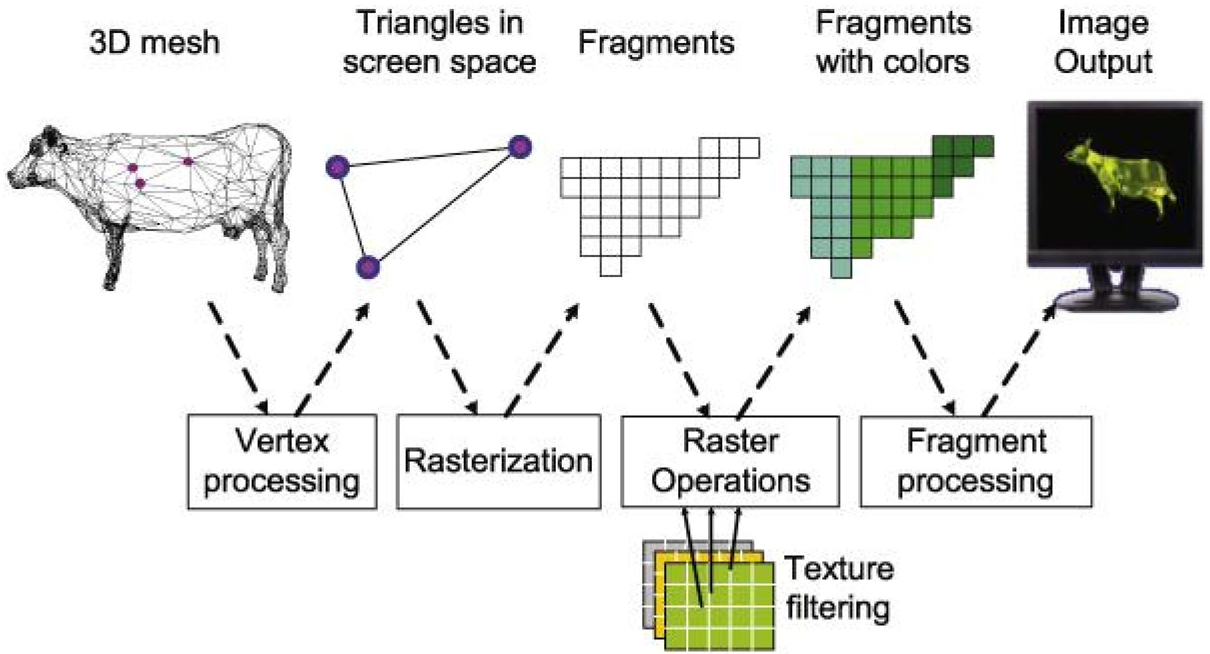
GPU Pipeline basics

- Triangles are rasterized into a framebuffer as **fragments**.



GPU Pipeline basics

- Each pixel is processed by a **fragment shader**.
- Fragment shaders are usually used to apply texture color, calculate lighting, and/or transparency.
- There may be many textures applied to the same model (multiple layers).
- The same pixel may be processed by multiple shaders in series.

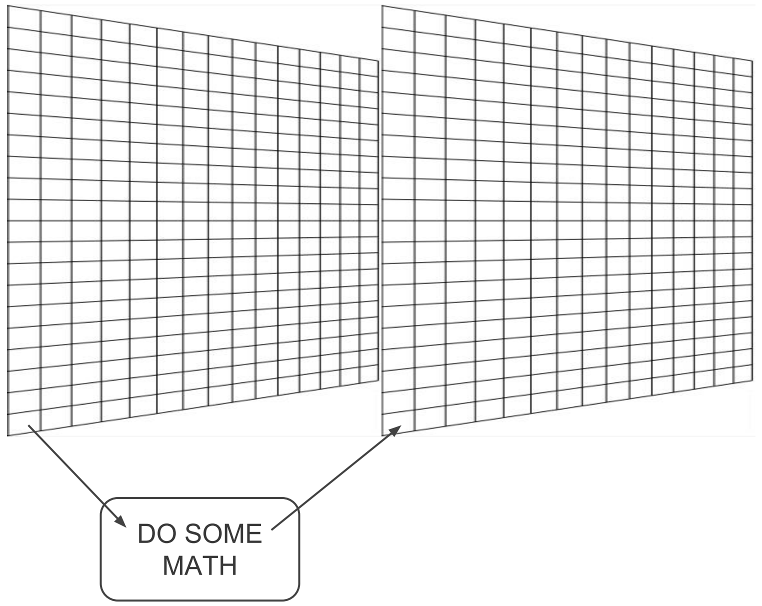


GPU Computing

Or how to use the GPU to do other things than rendering...

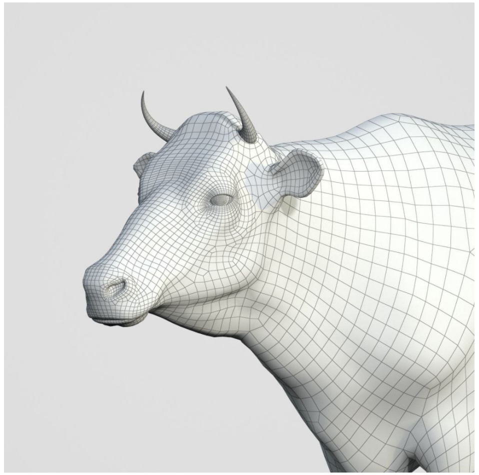
What do we have? Data.

What do we want to do? Compute



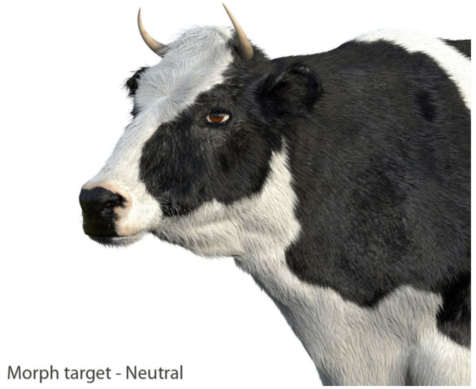
GPU Computing

GPU uses ***textures*** (images) to apply colors to the model.



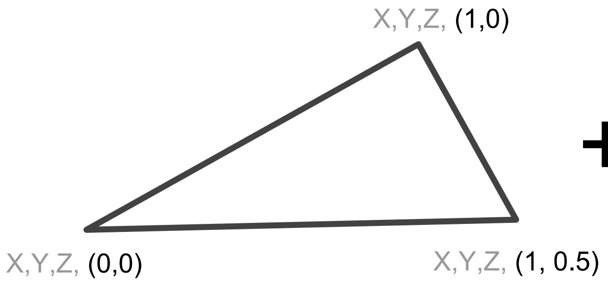
+

=

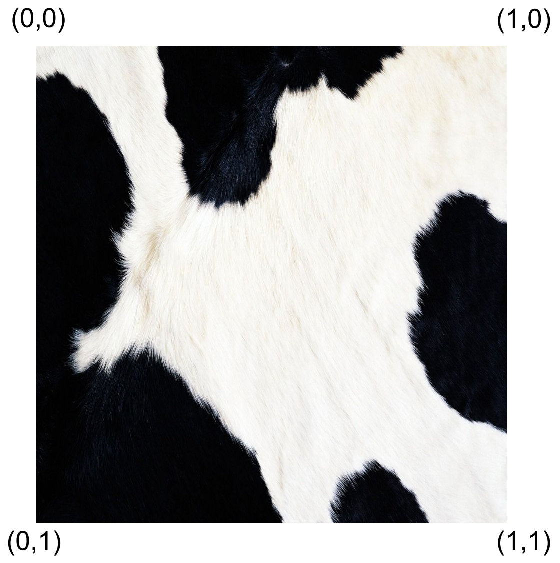


GPU Computing

Models define ***texture coordinates***, on top of the X,Y,Z coordinates, to indicate how a texture should be mapped to it.

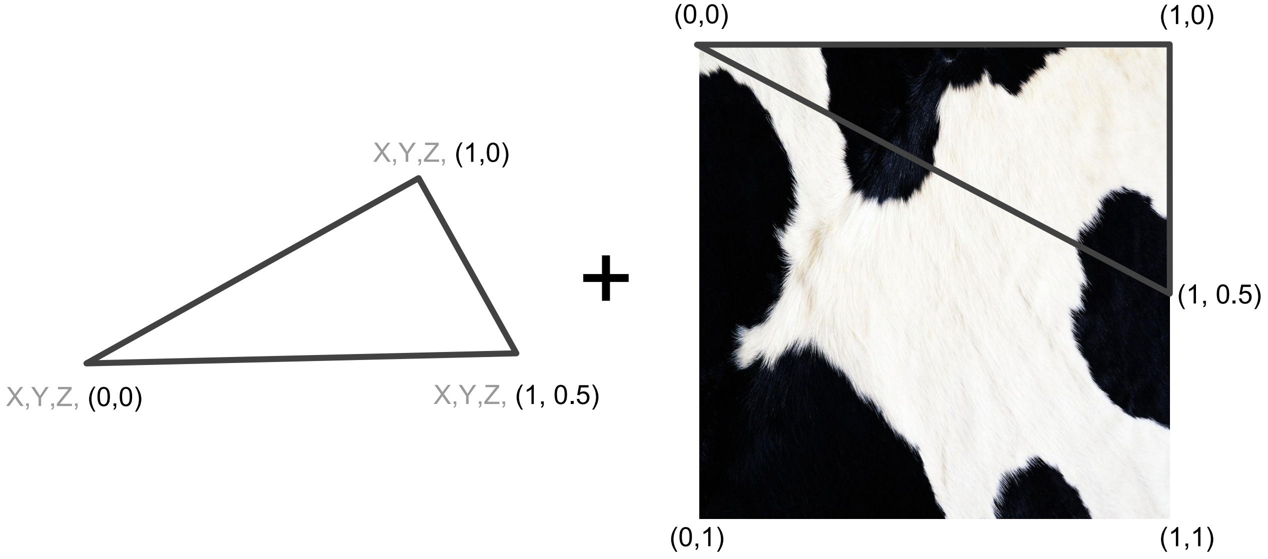


+



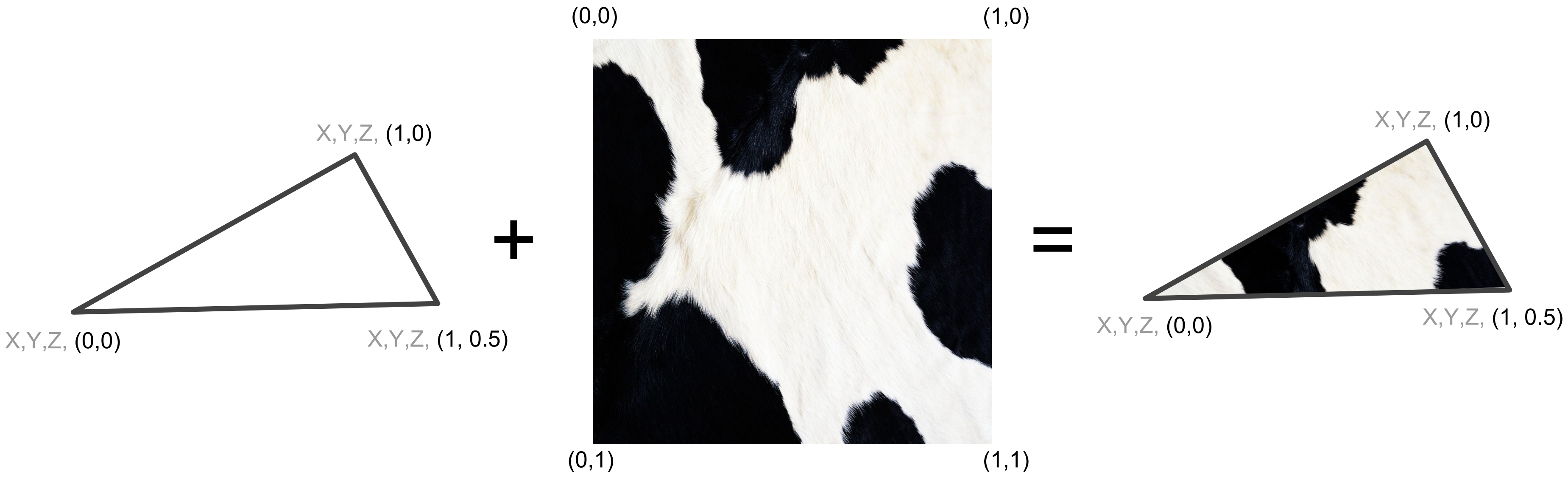
GPU Computing

Models define ***texture coordinates***, on top of the X,Y,Z coordinates, to indicate how a texture should be mapped to it.



GPU Computing

Models define ***texture coordinates***, on top of the X,Y,Z coordinates, to indicate how a texture should be mapped to it.

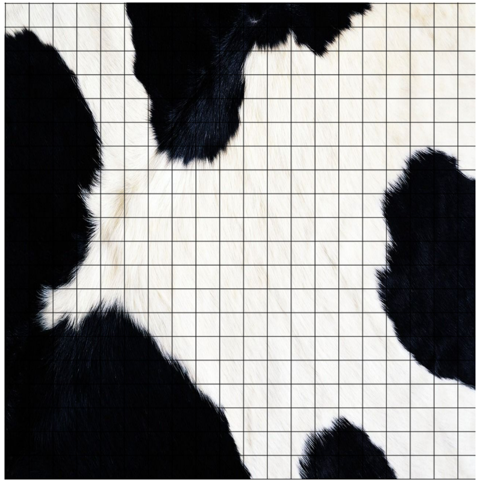


GPU Computing

Texture pixels are data, encoded in RGBA format (i.e. 4 numbers)

1 texture pixel = 4 values
x
width x height

-> Data for computing can be stored in a texture



One caveat: GPU textures often need to be sized as a power of 2, some GPU require textures to be square: 2x2, 4x4, 8x8, 16x16…

Max size is usually 2048x2048

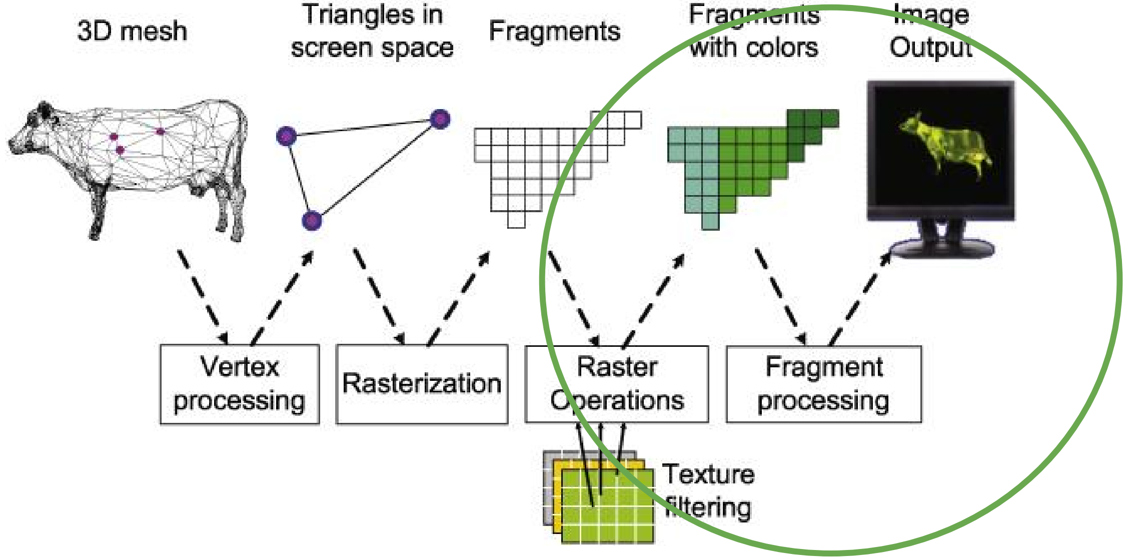
Textures are Data
-> We have a place for input data

Texture Pixels are processed by Fragment Shaders
-> We have a Compute Unit

How do we go from input data to output?

GPU Computing: Model

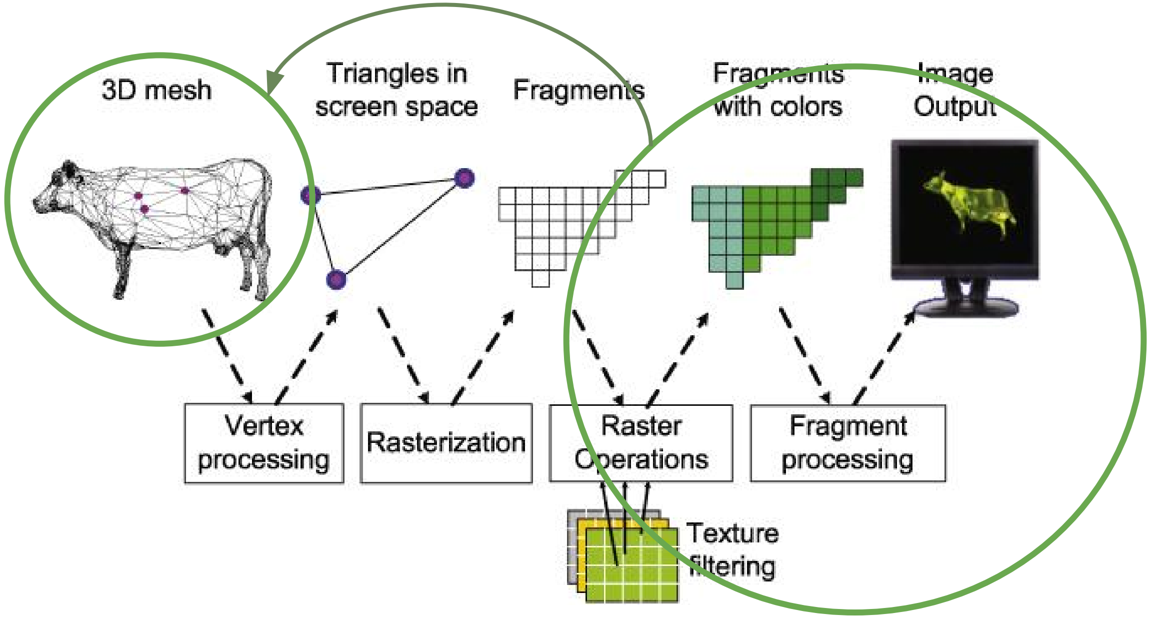
We need a way to get the fragment shader to compute our texture data:



GPU Computing: Model

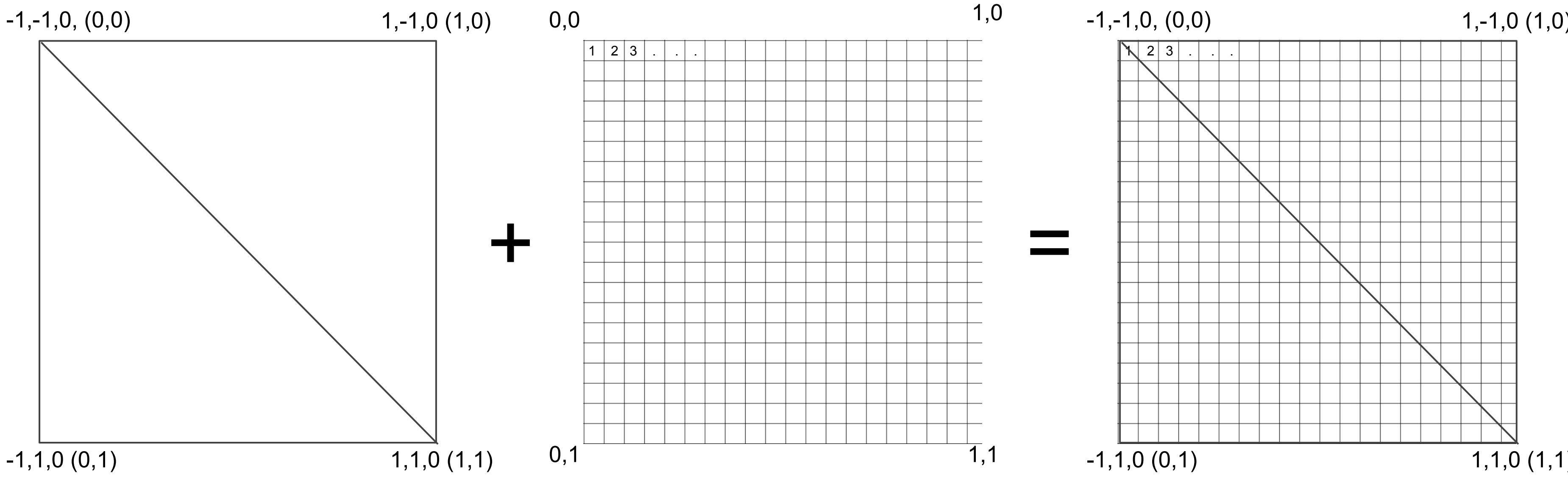
We need a way to get the fragment shader to compute our texture data:

We need a 3D model to map our (data) texture to the problem space.



GPU Computing: Model

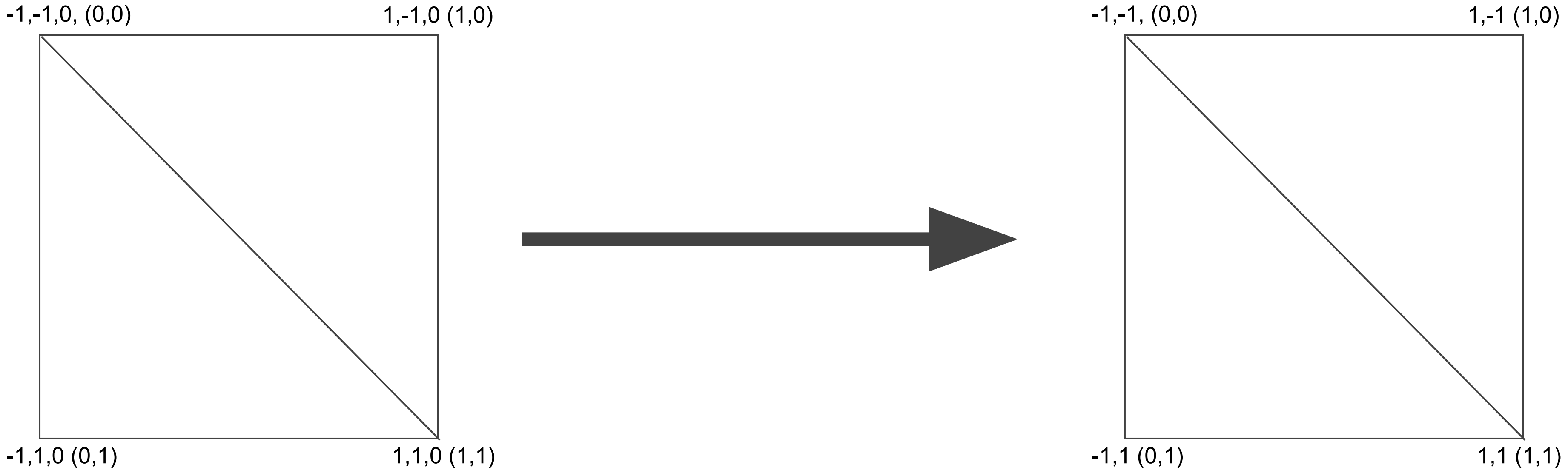
The simplest 3D model is a square, in the Z=0 plane.
It is made of just 2 triangles, and covers the full viewport.



Texture coordinates are such that the texture will cover the full square.

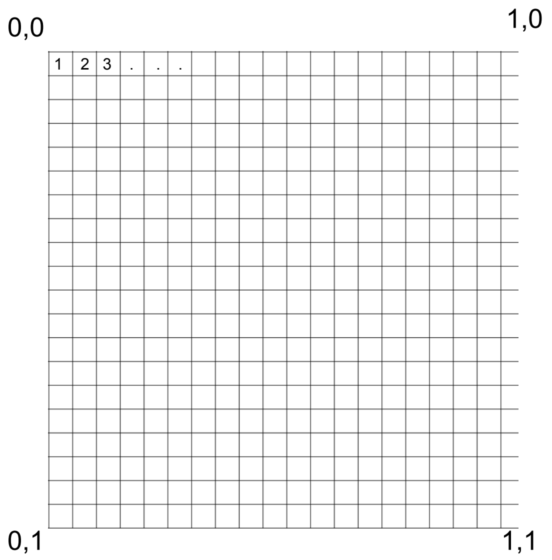
GPU Computing: vertex shader

The ***vertex shader***, which is usually in charge of 3D -> 2D transforms, will just map XY coordinates 1 to 1.

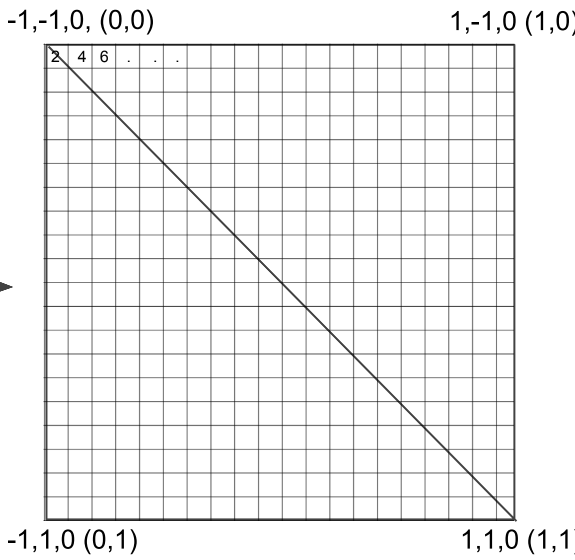


GPU Computing: fragment shader

The ***fragment shader***, which applies/transform the texture, does the computations

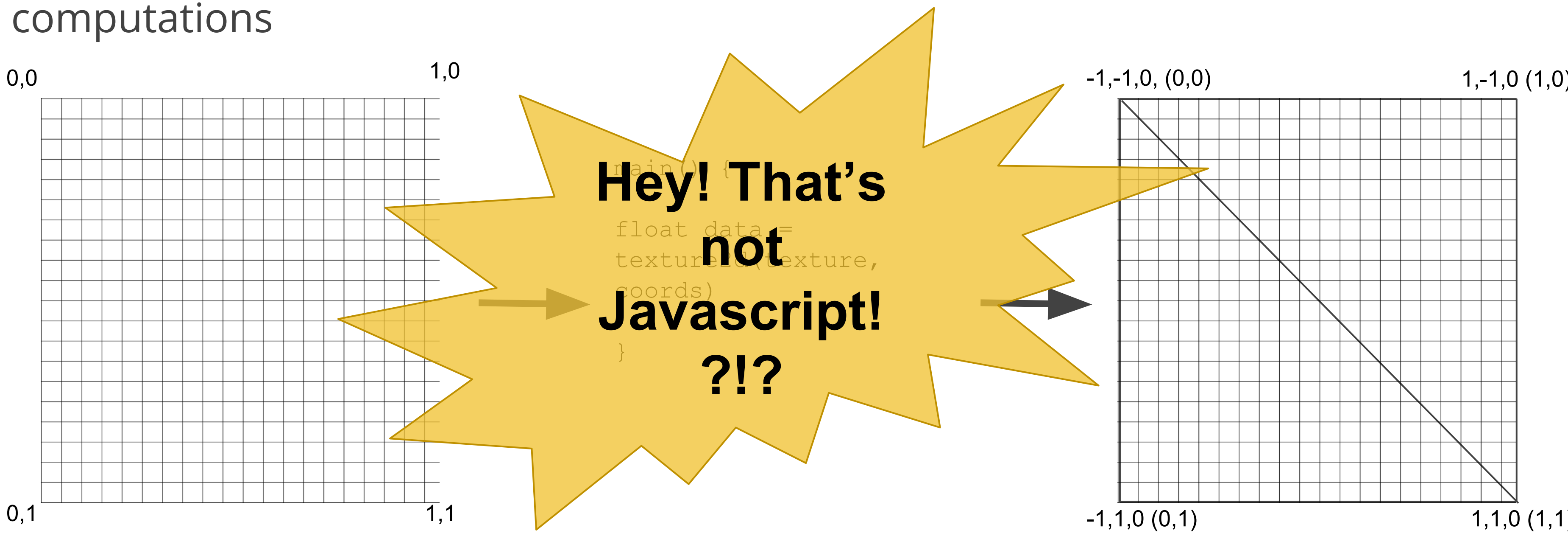


```
void main() {
  vec4 data = texture2d(texture, coords.st)
  gl_FragColor = data * 2.0
}
```



GPU Computing: fragment shader

The ***fragment shader***, which applies/transform the texture, does the computations



```
main() {
 float data =
 texture2D(texture,
 coords)
}
```

GPU Computing: GLSL

It’s actually GLSL (Graphics Library Shader Language), which is based on C.

For more info / reference:

https://www.khronos.org/files/opengl43-quick-reference-card.pdf

WebGL is a Javascript wrapper for OpenGL, and uses GLSL, so it’s technically not all Javascript.

GPU Computing

A square ***model*** is rendered, applying a ***texture*** (data), which is processed by a ***fragment shader*** to produce a computation of interest, encoded as RGBA pixels.

The resulting rastered image is rendered in a ***framebuffer***, that contains the output data.

In the typical GPU rendering pipeline, the framebuffer would then be attached to a ***canvas***.

For GPU computing, the output is not made visible, but can simply be read out of the ***framebuffer***.

If the computation is used in any kind of visualization, the computed data can be fed back to the rendering pipeline as a ***texture***.

GPU Computing: Data Encoding

Each ‘pixel’ must be encoded in RGBA format. These can be **`INT32`** or **`FLOAT32`**

WebGL requires the data to be typed, and provides specific Array classes:

```
var data = new Float32Array(256 * 256 * 4) ;
```

Because RGBA means 4 values

To make it easy (although space inefficient), one can use R for value, and ignore the rest. Often times, a *data point* has multiple attributes (i.e X, Y, velocity X, velocity Y), and RGBA allow for up to 4 attributes.

GPU Computing: Data Indexing

Texture coordinates are always in [0,1] space, while the data is a 1D or 2D matrix.

Data coordinates need to be translated into Textures coordinates, and vice-versa.

The fragment shader will usually include code like:

```
float texture_to_1D_id(vec2 textCoord)
{
  float x = floor(width * textCoord.s) ;
  float y = floor(height * textCoord.t) ;
  return width * y + x;
}
```

```
vec2 id_to_texture(float id)
{
  float x = fract(id / width) ;
  float y = floor(id / width) ;
  return vec2(x, y / height) ;
}
```

GPU Computing: shader variables

The CPU program can interact with the GPU shaders using variables called **uniforms**

Uniforms are defined in the shader, and must also be referenced on the Javascript side.

more on this later...

Implementing in the browser with WebGL

WebGL

The process

- Create a canvas object (not attached to DOM, not visible).
- Get a GL context.
- Compile vertex and fragment shaders.
- Build program.
- Create a data texture, load data.
- Create model (load vertices).
- Set uniforms (variables).
- Create and attach output framebuffer.
- Trigger render of the model.
- Read output framebuffer.

WebGL GPU Computing: create canvas & get context

The Canvas element holds the WebGL context.

```
// create canvas, but don't attach it to an element
var canvas = document.createElement('canvas');

// default attributes: don't use alpha (transparency), depth, or smoothing
var attributes = {alpha: false, depth: false, antialias: false}

// get GL context
var gl = canvas.getContext("webgl", attributes) || canvas.getContext('experimental-webgl',
attributes);

// we want floating point textures so let's check for support
var FPTexture = gl.getExtension('OES_texture_float');
```

If we don't have floating point support, life gets a little harder: it's necessary to encode floats into integers (we'll assume floating point textures are available here, as they are available on most h/w nowadays)

WebGL GPU Computing: shader source

Shaders are imported as text.

They can be included with:

- `<script type="x-shader/x-fragment"></script>`
- loaded using `ajax`
- as a link with a `rel=”import”` attribute (browser support is sparse)

```
<link rel="import" href="vertex_shader.glsl" id="vertex-shader">
```

WebGL GPU Computing: compile shaders

```
// compile vertex shader
var vertexShader = gl.createShader(gl.VERTEX_SHADER);
gl.shaderSource(vertexShader, source); // ← source is a string extracted from
tag
gl.compileShader(vertexShader);
var success = gl.getShaderParameter(vertexShader, gl.COMPILE_STATUS);

// compile fragment shader
var fragmentShader = gl.createShader(gl.FRAGMENT_SHADER);
gl.shaderSource(fragmentShader, source);
gl.compileShader(fragmentShader);
var success = gl.getShaderParameter(fragmentShader, gl.COMPILE_STATUS);
```

WebGL GPU Computing: build the program

```
// Create program object
var program = gl.createProgram();
// attach compiled shaders
gl.attachShader(program, vertexShader);
gl.attachShader(program, fragmentShader);
// link the program
gl.linkProgram(program);
var success = gl.getProgramParameter(program, gl.LINK_STATUS);
```

WebGL GPU Computing: create model (vertexBuffer)

```
var geometry = new Float32Array(
  [   -1.0,  1.0, 0.0, 0.0, 1.0,  // upper left (X,Y,Z + texture coords (s,t)
      -1.0, -1.0, 0.0, 0.0, 0.0,  // lower left
       1.0,  1.0, 0.0, 1.0, 1.0,  // upper right
       1.0, -1.0, 0.0, 1.0, 0.0 ] // lower right
);

// Store this data into a gl buffer
var vertexBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
gl.bufferData(gl.ARRAY_BUFFER, geometry, gl.STATIC_DRAW);
```

WebGL GPU Computing: set uniforms

```
gl.useProgram(program);
// Get handles on our variables and uniforms:
// this makes the link between the CPU and GPU variables

// position handle will point to the X,Y,Z coordinates called ‘position’ in the shader
var positionAttributeHandle = gl.getAttribLocation(program, "position");

// texture handle will point to the s,t texture coordinates data called ‘textureCoord’
var textureAttributeHandle = gl.getAttribLocation(program, "textureCoord");

gl.enableVertexAttribArray(positionAttributeHandle);
gl.enableVertexAttribArray(textureAttributeHandle);
```

WebGL GPU Computing: set attribute pointers

```
// Tell the attribute how to get data out of the geometry buffer (ARRAY_BUFFER)
gl.vertexAttribPointer(positionAttributeHandle,
  3,            // x,y,z vertex coords = 3 FLOATs
  gl.FLOAT,     // type
  gl.FALSE,     // do not normalize
  5 * 4,        // stride (each element of the geometry is 5 floats of 4 bytes)
  0             // offset start at 0 for vertex positions
);

gl.vertexAttribPointer(textureAttributeHandle,
  2,            // s,t textures coordinates = 2 FLOATs
  gl.FLOAT,     // type
  gl.FALSE,     // do not normalize
  5 * 4,        // stride (each element of the geometry is 5 floats of 4 bytes)
  3 * 4         // offset start at byte 12 for texture positions
);
```

So far we have:

- Loaded, compiled and linked the shaders to build a GPU program.
- Defined the geometry of the model, and how the GPU should access this data

Let’s create the input data texture

WebGL GPU Computing: create data texture

```
// Create the input data texture
var texture = gl.createTexture();

// Bind the texture so following methods effect this texture.
gl.bindTexture(gl.TEXTURE_2D, texture);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);   // disable filters
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);   // (set to nearest)
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.CLAMP_TO_EDGE); // wrap = clamp to edge
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.CLAMP_TO_EDGE); // -> no wrap

// Pixel format and data for the texture
gl.texImage2D(gl.TEXTURE_2D, // Target is the current texture bound above.
  0,                  // Level of detail 0 = full res (no downsampling)
  gl.RGBA,            // Internal format RGBA.
  width,              // Width - normalized to s.
  height,             // Height - normalized to t.
  0,                  // Always 0 in OpenGL ES.
  gl.RGBA,            // Format for each pixel RGBA.
  gl.FLOAT,           // Data type for each channel = FLOAT.
  data);              // Image data in the described format (4 FLOATs * nb data points), or null.
```

WebGL GPU Computing: textures and framebuffer

```
// create a handle for the data texture that will be called ‘data’ in the program
var dataTextureHandle = gl.getUniformLocation(program, "data");
// bind input data texture to GPU TEXTURE0
gl.activeTexture(gl.TEXTURE0);
gl.bindTexture(gl.TEXTURE_2D, texture);
// set uniform data texture handle to point to TEXTURE0
gl.uniform1i(dataTextureHandle, 0);

// create an outputTexture to hold results: same code as previous page, with ‘null’ as data
[...]

// Create a framebuffer to render in
var frameBuffer = gl.createFramebuffer();
gl.bindFramebuffer(gl.FRAMEBUFFER, frameBuffer);
gl.framebufferTexture2D(gl.FRAMEBUFFER,
     gl.COLOR_ATTACHMENT0, // We are providing the color buffer.
     gl.TEXTURE_2D,        // This is a 2D image texture.
     outputTexture,        // The ‘null’ output texture.
     0);                   // 0, we aren't using MIPMAPs
```

WebGL GPU Computing: render

```
// Check that our buffer was created properly
var status = this.frameBufferIsComplete();

if (status.isComplete) {
     // draw the ‘square’ model
     gl.drawArrays(
                       gl.TRIANGLE_STRIP, //primitive type
                       0, //offset in ARRAY_BUFFER where we have the data
                       4  //count (4 points to build 2 triangles)
                     );
}
```

WebGL GPU Computing: read back rendered data

```
var buffer = new Float32Array(4 *width * height);

gl.activeTexture(gl.TEXTURE0);
gl.bindTexture(gl.TEXTURE_2D, outputTexture);

gl.readPixels(
             0,          // x-coord of lower left corner
             0,          // y-coord of lower left corner
             width,      // width of the block
             height,     // height of the block
             gl.RGBA,    // Format of pixel data.
             gl.FLOAT,   // Data type of the pixel data, must match makeTexture
             buffer      // Load pixel data into buffer
           );
```

Now the fun part...

WebGL GPU Computing: vertex shader code

```
attribute vec3 position;        // 3-values vector attribute for position
attribute vec2 textureCoord;    // 2-values vector attribute for texture coords

// internal variable to pass texture coords to fragment shader
varying highp vec2 vTextureCoord;
void main() {
   gl_Position = vec4(position, 1.0);  // pass X,Y,Z coords as-is (4th not
used)
   vTextureCoord = textureCoord;       // pass texture coords as-is
}
```

Where the magic happens: fragment shader code

```
uniform sampler2D data;         // handler on the data texture
varying vec2 vTextureCoord;    // textures coords transferred from vertex shader

// compute simple 2x for each data point
vec4 computeElement(vec2 texCoords)
{
   vec4 data = texture2D(data, texCoords.st); // lookup value in texture
   return data * 2; // multiply all 4 RGBA values by 2
}
// main: compute and output to color variable
void main() {
   // gl_FragColor is a special GL variable a fragment shader
   // is responsible for setting. It is a RGBA vector of the output color
   gl_FragColor = computeElement(vTextureCoord);
}
```

All this for **THAT?**

GPU Computing in WebGL

This is just a step by step example of how to get the GPU to work for you in the browser. It’s a bit of work but it’s very powerful.

There is obviously a lot more you can do, but there are also limitations:

While there can be many textures and they can be READ in any location, one can only WRITE to the pixel being processed: thing through your algorithms accordingly; not everything is parallelizable.

Further work

Network Graph Force Layout

Resources

http://www.shaderific.com/glsl-functions/

https://www.khronos.org/files/webgl/webgl-reference-card-1_0.pdf

https://www.shadertoy.com/

http://learningwebgl.com/blog/?page_id=1217