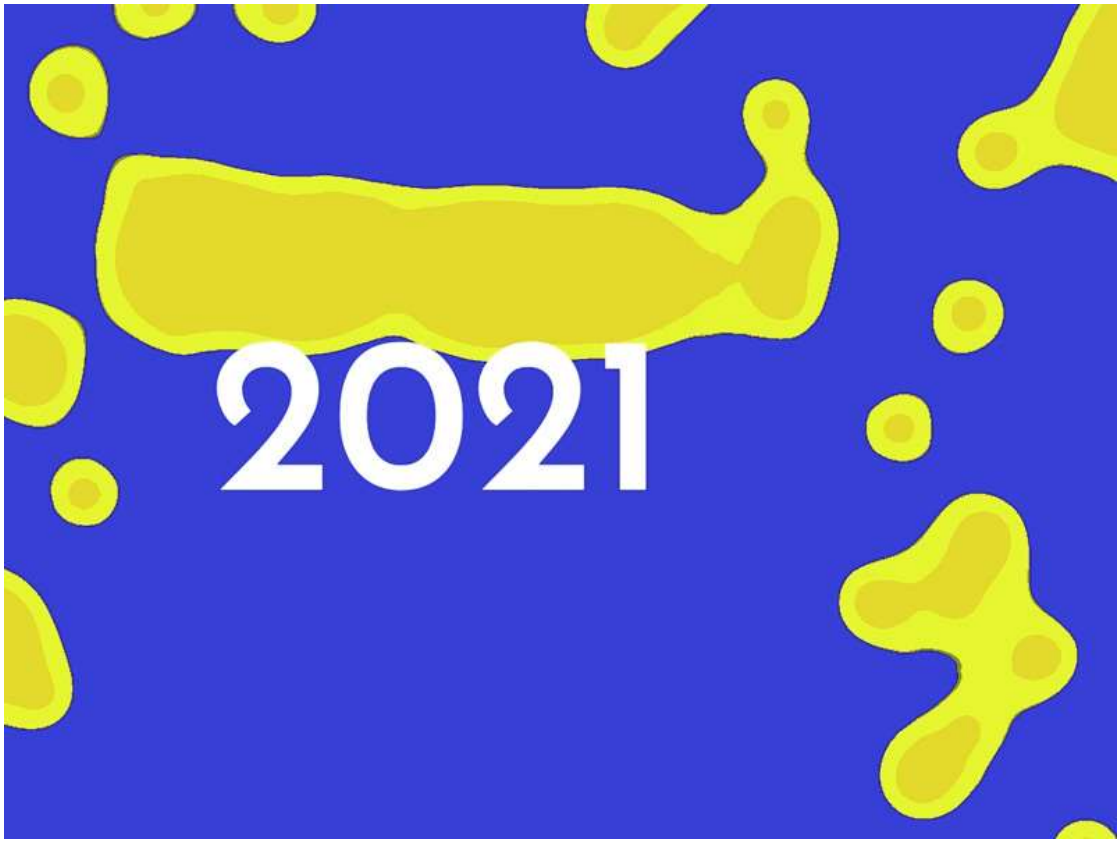
Drawing 2D Metaballs with WebGL2

An in-depth tutorial on how to code 2D visuals using WebGL2.

By Georgi Nikolov in Tutorials on January 19, 2021



View demo Download Source

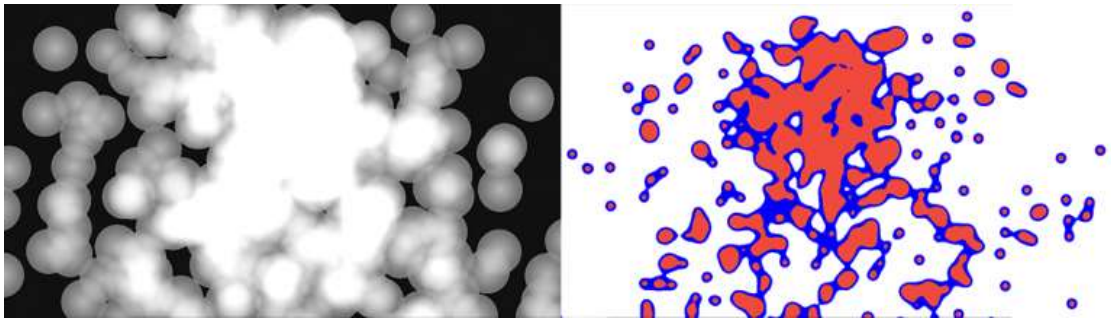
While many people shy away from writing vanilla WebGL and immediately jump to frameworks such as three.js or PixiJS, it is possible to achieve great visuals and complex animation with relatively small amounts of code. Today, I would like to present core WebGL concepts while programming some simple 2D visuals. This article assumes at least some higher-level knowledge of WebGL through a library.

Please note: WebGL2 has been around for years, yet Safari only recently enabled it behind a flag. It is a pretty significant upgrade from WebGL1 and brings tons of new useful features, some of which we will take advantage of in this tutorial.

What are we going to build

From a high level standpoint, to implement our 2D metaballs we need two steps:

- Draw a bunch of rectangles with radial linear gradient starting from their centers and expanding to their edges. Draw a lot of them and alpha blend them together in a separate framebuffer.
- Take the resulting image with the blended quads from step #1, scan its pixels one by one and decide the new color of the pixel depending on its opacity. For example – if the pixel has opacity smaller then 0.5, render it in red. Otherwise render it in yellow and so on.



Left: Multiple quads rendered with radial gradient, alpha blended and rendered to a texture. Right: Post-processing on the generated texture and rendering the result to the device screen. Conditional coloring of each pixel based on opacity.

Don't worry if these terms don't make a lot of sense just yet – we will go over each of the steps needed in detail. Let's jump into the code and start building!

Bootstrapping our program

We will start things by

- Creating a `HTMLCanvasElement`, sizing it to our device viewport and inserting it into the page DOM
- Obtaining a `WebGL2RenderingContext` to use for drawing stuff
- Setting the correct WebGL viewport and the background color for our scene
- Starting a `requestAnimationFrame` loop that will draw our scene as fast as the device allows. The speed is determined by various factors such as the hardware, current CPU / GPU workloads, battery levels, user preferences and so on. For smooth animation we are going to aim for 60FPS.

```
/* Create our canvas and obtain it's WebGL2RenderingContext */
const canvas = document.createElement('canvas')
const gl = canvas.getContext('webgl2')

/* Handle error somehow if no WebGL2 support */
if (!gl) {
  // ...
}

/* Size our canvas and listen for resize events */
resizeCanvas()
window.addEventListener('resize', resizeCanvas)

/* Append our canvas to the DOM and set its background-color with CSS
canvas.style.backgroundColor = 'black'
document.body.appendChild(canvas)

/* Issue first frame paint */
requestAnimationFrame(updateFrame)

function updateFrame (timestampMs) {
   /* Set our program viewport to fit the actual size of our monitor w
   gl.viewport(0, 0, canvas.width, canvas.height)
   /* Set the WebGL background colour to be transparent */
   gl.clearColor(0, 0, 0, 0)
   /* Clear the current canvas pixels */
   gl.clear(gl.COLOR_BUFFER_BIT)

   /* Issue next frame paint */
   requestAnimationFrame(updateFrame)
}

function resizeCanvas () {
   /*
      We need to account for devicePixelRatio when sizing our canvas.
      We will use it to obtain the actual pixel size of our viewport a
      We will then downscale it back to CSS units so it neatly fills o
      We also need to limit it because it can really slow our program.

      More info: https://webglfundamentals.org/webgl/lessons/webgl-res
```

```
        */
        const dpr = devicePixelRatio > 2 ? 2 : devicePixelRatio
        canvas.width = innerWidth * dpr
        canvas.height = innerHeight * dpr
        canvas.style.width = `${innerWidth}px`
        canvas.style.height = `${innerHeight}px`
    }
```

Drawing a quad

The next step is to actually draw a shape. WebGL has a rendering pipeline, which dictates how does the object you draw and its corresponding geometry and material end up on the device screen. WebGL is essentially just a rasterising engine, in the sense that you give it properly formatted data and it produces pixels for you.

The full rendering pipeline is out of the scope for this tutorial, but you can read more about it here. Let's break down what exactly we need for our program:

Defining our geometry and its attributes

Each object we draw in WebGL is represented as a `WebGLProgram` running on the device GPU. It consists of input variables and vertex and fragment shader to operate on these variables. The vertex shader responsibility is to position our geometry correctly on the device screen and fragment shader's responsibility is to control its appearance.

It's up to us as developers to write our vertex and fragment shaders, compile them on the device GPU and link them in a GLSL program. Once we have successfully done this, we must query this program's input variable locations that were allocated on the GPU for us, supply correctly formatted data to them, enable them and instruct them how to unpack and use our data.

To render our quad, we need 3 input variables:

1. `a_position` will dictate the position of each vertex of our quad geometry. We will pass it as an array of 12 floats, i.e. 2 triangles with 3 points per triangle, each represented by 2 floats (x, y). This variable is an `attribute`, i.e. it is obviously different for each of the points that make up our geometry.
2. `a_uv` will describe the texture offset for each point of our geometry. They too will be described as an array of 12 floats. We will use this data not to texture our quad with an image, but to dynamically create a radial linear gradient from the quad center. This variable is also an `attribute` and will too be different for each of our geometry points.
3. `u_projectionMatrix` will be an input variable represented as a 32bit float array of 16 items that will dictate how do we transform our geometry positions described in pixel values to the normalised WebGL coordinate system. This variable is a `uniform`, unlike the previous two, it will not change for each geometry position.

We can take advantage of Vertex Array Object to store the description of our GLSL program input variables, their locations on the GPU and how should they be unpacked and used.

`WebGLVertexArrayObject` s or `VAO` s are 1st class citizens in WebGL2, unlike in WebGL1 where they were hidden behind an optional extension and their support was not guaranteed. They let us type less, execute fewer WebGL bindings and keep our drawing state into a single, easy to manage object that is simpler to track. They essentially store the description of our geometry and we can reference them later.

We need to write the shaders in GLSL 3.00 ES, which WebGL2 supports. Our vertex shader will be pretty simple:

```
/*
  Pass in geometry position and tex coord from the CPU
*/
in vec4 a_position;
in vec2 a_uv;

/*
  Pass in global projection matrix for each vertex
*/
uniform mat4 u_projectionMatrix;

/*
  Specify varying variable to be passed to fragment shader
*/
out vec2 v_uv;

void main () {
  /*
   We need to convert our quad points positions from pixels to the nor
  */
  gl_Position = u_projectionMatrix * a_position;
  v_uv = a_uv;
}
```

At this point, after we have successfully executed our **vertex shader**, WebGL will fill in the pixels between the points that make up the geometry on the device screen. The way the space between the points is filled depends on what primitives are we using for drawing – WebGL supports points, lines and triangles.

We as developers do not have control over this step.

After it has rasterised our geometry, it will execute our **fragment shader** on each generated pixel. The fragment shader responsibility is the final

appearance of each generated pixel and wether it should even be rendered. Here is our fragment shader:

```
/*
  Set fragment shader float precision
*/
precision highp float;

/*
  Consume interpolated tex coord varying from vertex shader
*/
in vec2 v_uv;

/*
  Final color represented as a vector of 4 components - r, g, b, a
*/
out vec4 outColor;

void main () {
  /*
    This function will run on each each pixel generated by our quad ge
  */
  /*
    Calculate the distance for each pixel from the center of the quad
  */
  float dist = distance(v_uv, vec2(0.5)) * 2.0;
  /*
    Invert and clamp our distance from 0.0 to 1.0
  */
  float c = clamp(1.0 - dist, 0.0, 1.0);
  /*
    Use the distance to generate the pixel opacity. We have to explici
  */
  outColor = vec4(vec3(1.0), c);
}
```

Let's write two utility methods: `makeGLShader()` to create and compile our GLSL shaders and `makeGLProgram()` to link them into a GLSL program to be ran on the GPU:

```
/*
  Utility method to create a WebGLShader object and compile it on the
  https://developer.mozilla.org/en-US/docs/Web/API/WebGLShader
*/
function makeGLShader (shaderType, shaderSource) {
  /* Create a WebGLShader object with correct type */
  const shader = gl.createShader(shaderType)
  /* Attach the shaderSource string to the newly created shader */
  gl.shaderSource(shader, shaderSource)
  /* Compile our newly created shader */
  gl.compileShader(shader)
  const success = gl.getShaderParameter(shader, gl.COMPILE_STATUS)
  /* Return the WebGLShader if compilation was a success */
  if (success) {
    return shader
  }
  /* Otherwise log the error and delete the faulty shader */
  console.error(gl.getShaderInfoLog(shader))
  gl.deleteShader(shader)
}

/*
  Utility method to create a WebGLProgram object
  It will create both a vertex and fragment WebGLShader and link them
  https://developer.mozilla.org/en-US/docs/Web/API/WebGLProgram
*/
function makeGLProgram (vertexShaderSource, fragmentShaderSource) {
  /* Create and compile vertex WebGLShader */
  const vertexShader = makeGLShader(gl.VERTEX_SHADER, vertexShaderSour
  /* Create and compile fragment WebGLShader */
  const fragmentShader = makeGLShader(gl.FRAGMENT_SHADER, fragmentShad
  /* Create a WebGLProgram and attach our shaders to it */
  const program = gl.createProgram()
  gl.attachShader(program, vertexShader)
```

```
  gl.attachShader(program, fragmentShader)
  /* Link the newly created program on the device GPU */
  gl.linkProgram(program)
  /* Return the WebGLProgram if linking was successfull */
  const success = gl.getProgramParameter(program, gl.LINK_STATUS)
  if (success) {
    return program
  }
  /* Otherwise log errors to the console and delete fauly WebGLProgram
  console.error(gl.getProgramInfoLog(program))
  gl.deleteProgram(program)
}
```

And here is the complete code snippet we need to add to our previous code snippet to generate our geometry, compile our shaders and link them into a GLSL program:

```
const canvas = document.createElement('canvas')
/* rest of code */

/* Enable WebGL alpha blending */
gl.enable(gl.BLEND)
gl.blendFunc(gl.SRC_ALPHA, gl.ONE_MINUS_SRC_ALPHA)

/*
  Generate the Vertex Array Object and GLSL program
  we need to render our 2D quad
*/
const {
  quadProgram,
  quadVertexArrayObject,
} = makeQuad(innerWidth / 2, innerHeight / 2)

/* --------------- Utils ----------------- */

function makeQuad (positionX, positionY, width = 50, height = 50, draw
```

```
/*
  Write our vertex and fragment shader programs as simple JS strings

  !!! Important !!!!

  WebGL2 requires GLSL 3.00 ES
  We need to declare this version on the FIRST LINE OF OUR PROGRAM
  Otherwise it would not work!
*/
const vertexShaderSource = `#version 300 es
  /*
    Pass in geometry position and tex coord from the CPU
  */
  in vec4 a_position;
  in vec2 a_uv;

  /*
   Pass in global projection matrix for each vertex
  */
  uniform mat4 u_projectionMatrix;

  /*
    Specify varying variable to be passed to fragment shader
  */
  out vec2 v_uv;

  void main () {
    gl_Position = u_projectionMatrix * a_position;
    v_uv = a_uv;
  }
`
const fragmentShaderSource = `#version 300 es
  /*
    Set fragment shader float precision
  */
  precision highp float;

  /*
    Consume interpolated tex coord varying from vertex shader
  */
  in vec2 v_uv;
```

```
  /*
    Final color represented as a vector of 4 components - r, g, b, a
  */
  out vec4 outColor;

  void main () {
    float dist = distance(v_uv, vec2(0.5)) * 2.0;
    float c = clamp(1.0 - dist, 0.0, 1.0);
    outColor = vec4(vec3(1.0), c);
  }
`

/*
  Construct a WebGLProgram object out of our shader sources and link
*/
const quadProgram = makeGLProgram(vertexShaderSource, fragmentShader

/*
  Create a Vertex Array Object that will store a description of our g
  that we can reference later when rendering
*/
const quadVertexArrayObject = gl.createVertexArray()

/*
  1. Defining geometry positions

  Create the geometry points for our quad

  V6  _______ V5         V3
     |      /         /|
     |    /         /  |
     |  /         /    |
  V4 |/      V1 /_____| V2

   We need two triangles to form a single quad
   As you can see, we end up duplicating vertices:
   V5 & V3 and V4 & V1 end up occupying the same position.

   There are better ways to prepare our data so we don't end up with
   duplicates, but let's keep it simple for this demo and duplicate

   Unlike regular Javascript arrays, WebGL needs strongly typed data
   That's why we supply our positions as an array of 32 bit floating
```

```
*/
const vertexArray = new Float32Array([
  /*
    First set of 3 points are for our first triangle
  */
  positionX - width / 2,  positionY + height / 2, // Vertex 1 (X, Y)
  positionX + width / 2,  positionY + height / 2, // Vertex 2 (X, Y)
  positionX + width / 2,  positionY - height / 2, // Vertex 3 (X, Y)
  /*
    Second set of 3 points are for our second triangle
  */
  positionX - width / 2, positionY + height / 2, // Vertex 4 (X, Y)
  positionX + width / 2, positionY - height / 2, // Vertex 5 (X, Y)
  positionX - width / 2, positionY - height / 2  // Vertex 6 (X, Y)
])

/*
  Create a WebGLBuffer that will hold our triangles positions
*/
const vertexBuffer = gl.createBuffer()
/*
  Now that we've created a GLSL program on the GPU we need to supply
  We need to supply our 32bit float array to the a_position variable

  When you link a vertex shader with a fragment shader by calling gl
  WebGL (the driver/GPU/browser) decide on their own which index/loc

  Therefore we need to find the location of a_position from our prog
*/
const a_positionLocationOnGPU = gl.getAttribLocation(quadProgram, 'a

/*
  Bind the Vertex Array Object descriptior for this geometry
  Each geometry instruction from now on will be recorded under it

  To stop recording after we are done describing our geometry, we ne
*/
gl.bindVertexArray(quadVertexArrayObject)

/*
  Bind the active gl.ARRAY_BUFFER to our WebGLBuffer that describe t
*/
```

```
gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer)
/*
  Feed our 32bit float array that describes our quad to the vertexBu
  gl.ARRAY_BUFFER global handle
*/
gl.bufferData(gl.ARRAY_BUFFER, vertexArray, drawType)
/*
  We need to explicitly enable our the a_position variable on the GP
*/
gl.enableVertexAttribArray(a_positionLocationOnGPU)
/*
  Finally we need to instruct the GPU how to pull the data out of ou
  vertexBuffer and feed it into the a_position variable in the GLSL
*/
/*
  Tell the attribute how to get data out of positionBuffer (ARRAY_BU
*/
const size = 2          // 2 components per iteration
const type = gl.FLOAT    // the data is 32bit floats
const normalize = false  // don't normalize the data
const stride = 0         // 0 = move forward size * sizeof(type) eac
const offset = 0         // start at the beginning of the buffer
gl.vertexAttribPointer(a_positionLocationOnGPU, size, type, normaliz

/*
  2. Defining geometry UV texCoords

  V6  _______ V5         V3
     |      /         /|
     |    /         /  |
     |  /         /    |
  V4 |/      V1 /______| V2
*/
const uvsArray = new Float32Array([
  0, 0, // V1
  1, 0, // V2
  1, 1, // V3
  0, 0, // V4
  1, 1, // V5
  0, 1  // V6
])
/*
```

```
  The rest of the code is exactly like in the vertices step above.
  We need to put our data in a WebGLBuffer, look up the a_uv variabl
  in our GLSL program, enable it, supply data to it and instruct
  WebGL how to pull it out:
*/
const uvsBuffer = gl.createBuffer()
const a_uvLocationOnGPU = gl.getAttribLocation(quadProgram, 'a_uv')
gl.bindBuffer(gl.ARRAY_BUFFER, uvsBuffer)
gl.bufferData(gl.ARRAY_BUFFER, uvsArray, drawType)
gl.enableVertexAttribArray(a_uvLocationOnGPU)
gl.vertexAttribPointer(a_uvLocationOnGPU, 2, gl.FLOAT, false, 0, 0)

/*
  Stop recording and unbind the Vertex Array Object description for
*/
gl.bindVertexArray(null)

/*
  WebGL has a normalized viewport coordinate system which looks like

       Device Viewport
     ------- 1.0 ------
    |         |         |
    |         |         |
  -1.0 --------------- 1.0
    |         |         |
    |         |         |
     ------ -1.0 -------

   However as you can see, we pass the position and size of our quad
   To convert these pixels values to the normalized coordinate syste
   use the simplest 2D projection matrix.
   It will be represented as an array of 16 32bit floats

   You can read a gentle introduction to 2D matrices here
   https://webglfundamentals.org/webgl/lessons/webgl-2d-matrices.htm
*/
const projectionMatrix = new Float32Array([
  2 / innerWidth, 0, 0, 0,
  0, -2 / innerHeight, 0, 0,
  0, 0, 0, 0,
  -1, 1, 0, 1,
```

```
    ])

    /*
      In order to supply uniform data to our quad GLSL program, we first
    */
    gl.useProgram(quadProgram)
    /*
      Just like the a_position attribute variable earlier, we also need
      the location of uniform variables in the GLSL program in order to
    */
    const u_projectionMatrixLocation = gl.getUniformLocation(quadProgram
    /*
      Supply our projection matrix as a Float32Array of 16 items to the
    */
    gl.uniformMatrix4fv(u_projectionMatrixLocation, false, projectionMat
    /*
      We have set up our uniform variables correctly, stop using the qua
    */
    gl.useProgram(null)

    /*
      Return our GLSL program and the Vertex Array Object descriptor of
      We will need them to render our quad in our updateFrame method
    */
    return {
      quadProgram,
      quadVertexArrayObject,
    }
}

/* rest of code */
function makeGLShader (shaderType, shaderSource) {}
function makeGLProgram (vertexShaderSource, fragmentShaderSource) {}
function updateFrame (timestampMs) {}
```

We have successfully created a GLSL program `quadProgram`, which is running on the GPU, waiting to be drawn on the screen. We also have obtained a Vertex Array Object `quadVertexArrayObject`, which describes

our geometry and can be referenced before we draw. We can now draw our quad. Let's augment our `updateFrame()` method like so:

```
function updateFrame (timestampMs) {
   /* rest of our code */

  /*
    Bind the Vertex Array Object descriptor of our quad we generated e
  */
  gl.bindVertexArray(quadVertexArrayObject)
  /*
    Use our quad GLSL program
  */
  gl.useProgram(quadProgram)
  /*
    Issue a render command to paint our quad triangles
  */
  {
    const drawPrimitive = gl.TRIANGLES
    const vertexArrayOffset = 0
    const numberOfVertices = 6 // 6 vertices = 2 triangles = 1 quad
    gl.drawArrays(drawPrimitive, vertexArrayOffset, numberOfVertices)
  }
  /*
    After a successful render, it is good practice to unbind our
GLSL program and Vertex Array Object so we keep WebGL state clean.
    We will bind them again anyway on the next render
  */
  gl.useProgram(null)
  gl.bindVertexArray(null)

  /* Issue next frame paint */
  requestAnimationFrame(updateFrame)
}
```

And here is our result:

We can use the great SpectorJS Chrome extension to capture our WebGL operations on each frame. We can look at the entire command list with their associated visual states and context information. Here is what it takes to render a single frame with our `updateFrame()` call:

A screenshot of all the steps we implemented to render a single quad. (Click to see a larger version)

Some gotchas:

1. We declare the vertices positions of our triangles in a counter clockwise order. This is important.
2. We need to explicitly enable blending in WebGL and specify it's blend operation. For our demo we will use `gl.ONE_MINUS_SRC_ALPHA` as a blend function (multiplies all colors by 1 minus the source alpha value).
3. In our vertex shader you can see we expect the input variable `a_position` to be vector with 4 components (`vec4`), while in Javascript we specify only 2 items per vertex. That's because the default attribute value is 0, 0, 0, 1. It doesn't matter that you're only supplying x and y from your attributes. `z` defaults to 0 and `w` defaults to 1.
4. As you can see, WebGL is a state machine, where you have to constantly bind stuff before you are able to work on it and you always have to make sure you unbind it afterwards. Consider how in the code snippet above we supplied a Float32Array with out positions to the `vertexBuffer`:

```
const vertexArray = new Float32Array([/* ... */])
const vertexBuffer = gl.createBuffer()
/* Bind our vertexBuffer to the global binding WebGL bind point gl.ARR
gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer)
/* At this point, gl.ARRAY_BUFFER represents vertexBuffer */
/* Supply data to our vertexBuffer using the gl.ARRAY_BUFFER binding p
gl.bufferData(gl.ARRAY_BUFFER, vertexArray, gl.STATIC_DRAW)
/* Do a bunch of other stuff with the active gl.ARRAY_BUFFER (vertexBu
// ...

/* After you have done your work, unbind it */
gl.bindBuffer(gl.ARRAY_BUFFER, null)
```

This is totally opposite of Javascript, where this same operation would be expressed like this for example (pseudocode):

```
const vertexBuffer = gl.createBuffer()
vertexBuffer.addData(vertexArray)
vertexBuffer.setDrawOperation(gl.STATIC_DRAW)
// etc.
```

Coming from Javascript background, initially I found WebGL's state machine way of doing things by constantly binding and unbinding really odd. One must exercise good discipline and always make sure to unbind stuff after using it, even in trivial programs like ours! Otherwise you risk things not working and hard to track bugs.

Drawing lots of quads

We have successfully rendered a single quad, but in order to make things more interesting and visually appealing, we need to draw more.

As we saw already, we can easily create new geometries with different position using our `makeQuad()` utility helper. We can pass them different positions and radiuses and compile each one of them into a separate GLSL program to be executed on the GPU. This will work, **however**:

As we saw in our update loop method `updateFrame`, to render our quad on each frame we must:

1. Use the correct GLSL program by calling `gl.useProgram()`
2. Bind the correct VAO describing our geometry by calling `gl.bindVertexArray()`
3. Issue a draw call with correct primitive type by calling `gl.drawArrays()`

So 3 WebGL commands in total.

What if we want to render 500 quads? Suddenly we jump to 500×3 or 1500 individual WebGL calls on each frame of our animation. If we want 1000quads we jump up to 3000 individual calls, without even counting all of the preparation WebGL bindings we have to do before our updateFrame loop starts.

Geometry Instancing is a way to reduce these calls. It works by letting you tell WebGL how many times you want the same thing drawn (the number of instances) with minor variations, such as rotation, scale, position etc. Examples include trees, grass, crowd of people, boxes in a warehouse, etc.

Just like `VAO` s, instancing is a 1st class citizen in WebGL2 and does not require extensions, unlike WebGL1. Let's augment our code to support geometry instancing and render 1000 quads with random positions.

First of all, we need to decide on how many quads we want rendered and prepare the offset positions for each one as a new array of 32bit floats. Let's do **1000 quads** and positions them randomly in our viewport:

```
/* rest of code */

/* How many quads we want rendered */
const QUADS_COUNT = 1000
/*
  Array to store our quads positions
  We need to layout our array as a continuous set
  of numbers, where each pair represents the X and Y
  or a single 2D position.

  Hence for 1000 quads we need an array of 2000 items
  or 1000 pairs of X and Y
*/
```

```
const quadsPositions = new Float32Array(QUADS_COUNT * 2)
for (let i = 0; i < QUADS_COUNT; i++) {
  /*
    Generate a random X and Y position
  */
  const randX = Math.random() * innerWidth
  const randY = Math.random() * innerHeight
  /*
    Set the correct X and Y for each pair in our array
  */
  quadsPositions[i * 2 + 0] = randX
  quadsPositions[i * 2 + 1] = randY
}

/*
  We also need to augment our makeQuad() method
  It no longer expects a single position, rather an array of positions
*/
const {
  quadProgram,
  quadVertexArrayObject,
} = makeQuad(quadsPositions)

/* rest of code */
```

Instead of a single position, we will now pass an array of positions into our `makeQuad()` method. Let's augment this method to receive our offsets array as a new variable input `a_offset` to our shaders which will contain the correct XY offset for a particular instance. To do this, we need to prepare our offsets as a new `WebGLBuffer` and instruct WebGL how to upack them, just like we did for `a_position` and `a_uv`

```
function makeQuad (quadsPositions, width = 70, height = 70, drawType =
  /* rest of code */
```

```
  /*
    Add offset positions for our individual instances
    They are declared and used in exactly the same way as
    "a_position" and "a_uv" above
  */
  const offsetsBuffer = gl.createBuffer()
  const a_offsetLocationOnGPU = gl.getAttribLocation(quadProgram, 'a_o
  gl.bindBuffer(gl.ARRAY_BUFFER, offsetsBuffer)
  gl.bufferData(gl.ARRAY_BUFFER, quadsPositions, drawType)
  gl.enableVertexAttribArray(a_offsetLocationOnGPU)
  gl.vertexAttribPointer(a_offsetLocationOnGPU, 2, gl.FLOAT, false, 0,
  /*
    HOWEVER, we must add an additional WebGL call to set this attribut
    change per instance, instead of per vertex like a_position and a_u
  */
  const instancesDivisor = 1
  gl.vertexAttribDivisor(a_offsetLocationOnGPU, instancesDivisor)

  /*
    Stop recording and unbind the Vertex Array Object descriptor for t
  */
  gl.bindVertexArray(null)

  /* rest of code */
}
```

We need to augment our original `vertexArray` responsible for passing data into our `a_position` GLSL variable. We **no longer** need to offset it to the desired position like in the first example, now the `a_offset` variable will take care of this in the vertex shader:

```
const vertexArray = new Float32Array([
  /*
    First set of 3 points are for our first triangle
  */
 -width / 2,  height / 2, // Vertex 1 (X, Y)
```

```
    width / 2,  height / 2, // Vertex 2 (X, Y)
    width / 2, -height / 2, // Vertex 3 (X, Y)
    /*
      Second set of 3 points are for our second triangle
    */
   -width / 2,  height / 2, // Vertex 4 (X, Y)
    width / 2, -height / 2, // Vertex 5 (X, Y)
   -width / 2, -height / 2  // Vertex 6 (X, Y)
  ])
```

We also need to augment our vertex shader to consume and use the new `a_offset` input variable we pass from Javascript:

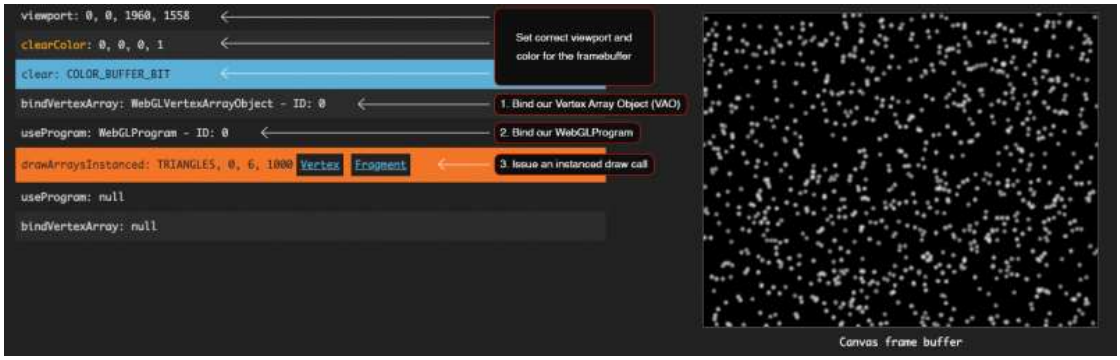
```
const vertexShaderSource = `#version 300 es
  /* rest of GLSL code */
  /*
    This input vector will change once per instance
  */
  in vec4 a_offset;

  void main () {
     /* Account a_offset in the final geometry posiiton */
     vec4 newPosition = a_position + a_offset;
     gl_Position = u_projectionMatrix * newPosition;
  }
  /* rest of GLSL code */
`
```

And as a final step we need to change our `drawArrays` call in our `updateFrame` to `drawArraysInstanced` to account for instancing. This new method expects the exact same arguments and adds **`instanceCount`** as last one:

```
function updateFrame (timestampMs) {
   /* rest of code */
   {
     const drawPrimitive = gl.TRIANGLES
     const vertexArrayOffset = 0
     const numberOfVertices = 6 // 6 vertices = 2 triangles = 1 quad
     gl.drawArraysInstanced(drawPrimitive, vertexArrayOffset, numberOf
   }
   /* rest of code */
}
```

And with all these changes, here is our updated example:

Even though we increased the amount of rendered objects by 1000x, we are still making **3 WebGL calls** on each frame. That's a pretty great performance win!



All WebGL calls needed to draw our 1000 quads in a single `updateFrame()` call. Note the amount of needed calls did not increase from the previous example thanks to instancing.

Post Processing with a fullscreen quad

Now that we have our 1000 quads successfully rendering to the device screen on each frame, we can turn them into metaballs. As we established, we need to scan the pixels of the picture we generated in the previous steps and determine the alpha value of each pixel. If it is below a certain threshold, we discard it, otherwise we color it.

To do this, instead of rendering our scene directly to the screen as we do right now, we need to render it to a texture. We will do our post processing on this texture and render the result to the device screen.

> Post-Processing is a technique used in graphics that allows you to take a current input texture, and manipulate its pixels to produce a transformed image. This can be used to apply shiny effects like volumetric lighting, or any other filter type effect you've seen in applications like Photoshop or Instagram.
>
> *Nicolas Garcia Belmonte*

The basic technique for creating these effects is pretty straightforward:

1. A WebGLTexture is created with the same size as the canvas and attached as a color attachment to a WebGLFramebuffer. At the beginning of our `updateFrame()` method, the framebuffer is set as the render target, and the entire scene is rendered normally to it.
2. Next, a full-screen quad is rendered to the device screen using the texture generated in step 1 as an input. The shader used during the rendering of the quad is what contains the post-process effect.

Creating a texture and framebuffer to render to

A framebuffer is just a collection of attachments. Attachments are either textures or renderbuffers. Let's create a WebGLTexture and attach it to a framebuffer as the first color attachment:

```
/* rest of code */

const renderTexture = makeTexture()
const framebuffer = makeFramebuffer(renderTexture)

function makeTexture (textureWidth = canvas.width, textureHeight = can
  /*
    Create the texture that we will use to render to
  */
  const targetTexture = gl.createTexture()
  /*
    Just like everything else in WebGL up until now, we need to bind i
    so we can configure it. We will unbind it once we are done with it
  */
  gl.bindTexture(gl.TEXTURE_2D, targetTexture)

  /*
    Define texture settings
  */
  const level = 0
  const internalFormat = gl.RGBA
```

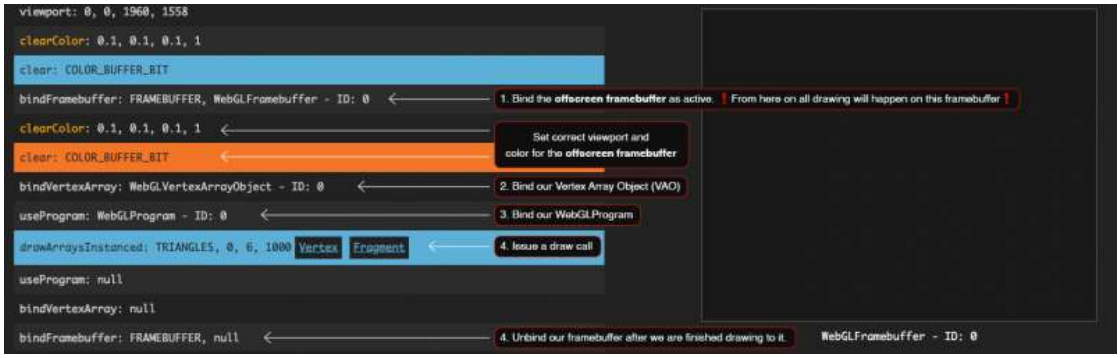
```
  const border = 0
  const format = gl.RGBA
  const type = gl.UNSIGNED_BYTE
  /*
    Notice how data is null. That's because we don't have data for thi
    We just need WebGL to allocate the texture
  */
  const data = null
  gl.texImage2D(gl.TEXTURE_2D, level, internalFormat, textureWidth, te

  /*
    Set the filtering so we don't need mips
  */
  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR)
  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.CLAMP_TO_EDGE)
  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.CLAMP_TO_EDGE)

  return renderTexture
}

function makeFramebuffer (texture) {
  /*
    Create and bind the framebuffer
  */
  const fb = gl.createFramebuffer()
  gl.bindFramebuffer(gl.FRAMEBUFFER, fb)

  /*
    Attach the texture as the first color attachment
  */
  const attachmentPoint = gl.COLOR_ATTACHMENT0
  gl.framebufferTexture2D(gl.FRAMEBUFFER, attachmentPoint, gl.TEXTURE_
}
```

We have successfully created a texture and attached it as color attachment to a framebuffer. Now we can render our scene to it. Let's augment our `updateFrame()` method:

```
function updateFrame () {
  gl.viewport(0, 0, canvas.width, canvas.height)
  gl.clearColor(0, 0, 0, 0)
  gl.clear(gl.COLOR_BUFFER_BIT)

  /*
    Bind the framebuffer we created
    From now on until we unbind it, each WebGL draw command will rende
  */
  gl.bindFramebuffer(gl.FRAMEBUFFER, framebuffer)

  /* Set the offscreen framebuffer background color to be transparent
  gl.clearColor(0.2, 0.2, 0.2, 1.0)
  /* Clear the offscreen framebuffer pixels */
  gl.clear(gl.COLOR_BUFFER_BIT)

  /*
    Code for rendering our instanced quads here
  */

  /*
    We have successfully rendered to the framebuffer at this point
    In order to render to the screen next, we need to unbind it
  */
  gl.bindFramebuffer(gl.FRAMEBUFFER, null)

  /* Issue next frame paint */
  requestAnimationFrame(updateFrame)
}
```

Let’s take a look at our result:

As you can see, we get an empty screen. There are **no errors** and the program is running just fine – keep in mind however that we are rendering to a separate framebuffer, not the default device screen framebuffer!



Our program produces black screen, since we are rendering to the offscreen framebuffer

In order to display our offscreen framebuffer back on the screen, we need to render a fullscreen quad and use the framebuffer's texture as an input.

Creating a fullscreen quad and displaying our texture on it

Let's create a new quad. We can reuse our `makeQuad()` method from the above snippets, but we need to augment it to support instancing optionally and be able to put vertex and fragment shader sources as outside argument

variables. This time we need only one quad and the shaders we need for it are different.

Take a look at the updated `makeQuad()` signature:

```
/* rename our instanced quads program & VAO */
const {
  quadProgram: instancedQuadsProgram,
  quadVertexArrayObject: instancedQuadsVAO,
} = makeQuad({
  instancedOffsets: quadsPositions,
  /*
    We need different set of vertex and fragment shaders
    for the different quads we need to render, so pass them from outsi
  */
  vertexShaderSource: instancedQuadVertexShader,
  fragmentShaderSource: instancedQuadFragmentShader,
  /*
    support optional instancing
  */
  isInstanced: true,
})
```

Let's use the same method to create a new fullscreen quad and render it. First our vertex and fragment shader:

```
const fullscreenQuadVertexShader = `#version 300 es
   in vec4 a_position;
   in vec2 a_uv;

   uniform mat4 u_projectionMatrix;

   out vec2 v_uv;
```

```
    void main () {
     gl_Position = u_projectionMatrix * a_position;
     v_uv = a_uv;
    }
`

const fullscreenQuadFragmentShader = `#version 300 es
  precision highp float;

  /*
    Pass our texture we render to as an uniform
  */
  uniform sampler2D u_texture;

  in vec2 v_uv;

  out vec4 outputColor;

  void main () {
    /*
      Use our interpolated UVs we assigned in Javascript to lookup
      texture color value at each pixel
    */
    vec4 inputColor = texture(u_texture, v_uv);

    /*
      0.5 is our alpha threshold we use to decide if
      pixel should be discarded or painted
    */
    float cutoffThreshold = 0.5;
    /*
      "cutoff" will be 0 if pixel is below 0.5 or 1 if above

      step() docs - https://thebookofshaders.com/glossary/?search=step
    */
    float cutoff = step(cutoffThreshold, inputColor.a);

    /*
      Let's use mix() GLSL method instead of if statement
      if cutoff is 0, we will discard the pixel by using empty color w
      otherwise, let's use black with alpha of 1
```

```
      mix() docs - https://thebookofshaders.com/glossary/?search=mix
    */
    vec4 emptyColor = vec4(0.0);
    /* Render base metaballs shapes */
    vec4 borderColor = vec4(1.0, 0.0, 0.0, 1.0);
    outputColor = mix(
      emptyColor,
      borderColor,
      cutoff
    );

    /*
      Increase the treshold and calculate new cutoff, so we can render
    */
    cutoffThreshold += 0.05;
    cutoff = step(cutoffThreshold, inputColor.a);
    vec4 fillColor = vec4(1.0, 1.0, 0.0, 1.0);
    /*
      Add new smaller metaballs color on top of the old one
    */
    outputColor = mix(
      outputColor,
      fillColor,
      cutoff
    );
  }
`
```

Let’s use them to create and link a valid GLSL program, just like when we rendered our instances:

```
const {
  quadProgram: fullscreenQuadProgram,
  quadVertexArrayObject: fullscreenQuadVAO,
} = makeQuad({
  vertexShaderSource: fullscreenQuadVertexShader,
```

```
  fragmentShaderSource: fullscreenQuadFragmentShader,
  isInstanced: false,
  width: innerWidth,
  height: innerHeight
})
/*
  Unlike our instances GLSL program, here we need to pass an extra uni
  Tell the shader to use texture unit 0 for u_texture
*/
gl.useProgram(fullscreenQuadProgram)
const u_textureLocation = gl.getUniformLocation(fullscreenQuadProgram,
gl.uniform1i(u_textureLocation, 0)
gl.useProgram(null)
```

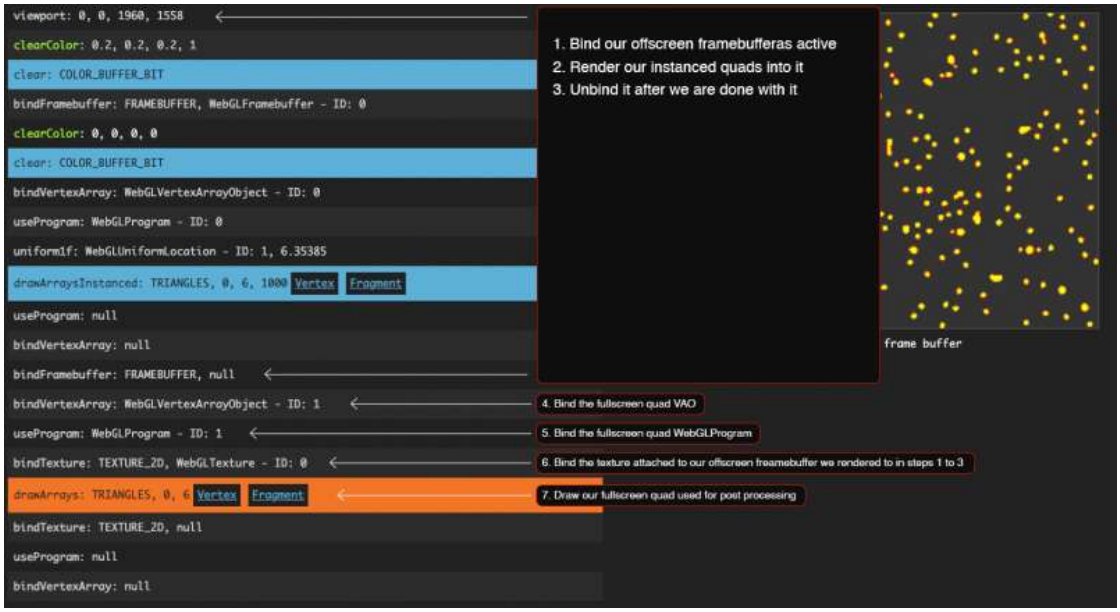
Finally we can render the fullscreen quad with the result texture as an uniform `u_texture`. Let's change our `updateFrame()` method:

```
function updateFrame () {
 gl.bindFramebuffer(gl.FRAMEBUFFER, framebuffer)
 /* render instanced quads here */
 gl.bindFramebuffer(gl.FRAMEBUFFER, null)

 /*
   Render our fullscreen quad
 */
 gl.bindVertexArray(fullscreenQuadVAO)
 gl.useProgram(fullscreenQuadProgram)
 /*
  Bind the texture we render to as active TEXTURE_2D
 */
 gl.bindTexture(gl.TEXTURE_2D, renderTexture)
 {
   const drawPrimitive = gl.TRIANGLES
   const vertexArrayOffset = 0
   const numberOfVertices = 6 // 6 vertices = 2 triangles = 1 quad
   gl.drawArrays(drawPrimitive, vertexArrayOffset, numberOfVertices)
```

```
    }
    /*
      Just like everything else, unbind our texture once we are done rend
    */
    gl.bindTexture(gl.TEXTURE_2D, null)
    gl.useProgram(null)
    gl.bindVertexArray(null)
    requestAnimationFrame(updateFrame)
  }
```

And here is our final result (I also added a simple animation to make the effect more apparent):

And here is the breakdown of one `updateFrame()` call:



You can clearly see how we render our 1000 instanced quads in separate framebuffer in steps 1 to 3. We then draw and manipulate the resulting texture to a fullscreen quad that we render in steps 4 to 7.

Aliasing issues

On my 2016 MacBook Pro with retina display I can clearly see aliasing issues with our current example. If we are to add bigger radiuses and blow our animation to fullscreen the problem will become only more noticeable.

The issue comes from the fact we are rendering to a 8bit `gl.UNSIGNED_BYTE` texture. If we want to increase the detail, we need to switch to floating point textures (32 bit float `gl.RGBA32F` or 16 bit float `gl.RGBA16F`). The catch is that these textures are not supported on all hardware and are not part of WebGL2 core. They are available through optional extensions, that we need to check if exist.

The extensions we are interested in to render to 32bit floating point textures are

- `EXT_color_buffer_float`

- `OES_texture_float_linear`

If these extensions are present on the user device, we can use `internalFormat = gl.RGBA32F` and `textureType = gl.FLOAT` when creating our render textures. If they are not present, we can optionally fallback and render to 16bit floating textures. The extensions we need in that case are:

- `EXT_color_buffer_half_float`
- `OES_texture_half_float_linear`

If these extensions are present, we can use `internalFormat = gl.RGBA16F` and `textureType = gl.HALF_FLOAT` for our render texture. If not, we will fallback to what we have used up until now – `internalFormat = gl.RGBA` and `textureType = gl.UNSIGNED_BYTE` .

Here is our updated `makeTexture()` method:

```
function makeTexture (textureWidth = canvas.width, textureHeight = can
  /*
   Initialize internal format & texture type to default values
  */
  let internalFormat = gl.RGBA
  let type = gl.UNSIGNED_BYTE

  /*
    Check if optional extensions are present on device
  */
  const rgba32fSupported = gl.getExtension('EXT_color_buffer_float') &

  if (rgba32fSupported) {
    internalFormat = gl.RGBA32F
    type = gl.FLOAT
  } else {
```

```
    /*
      Check if optional fallback extensions are present on device
    */
    const rgba16fSupported = gl.getExtension('EXT_color_buffer_half_fl
    if (rgba16fSupported) {
      internalFormat = gl.RGBA16F
      type = gl.HALF_FLOAT
    }
  }

  /* rest of code */

  /*
    Pass in correct internalFormat and textureType to texImage2D call
  */
  gl.texImage2D(gl.TEXTURE_2D, level, internalFormat, textureWidth, te

  /* rest of code */
}
```

And here is our updated result:

Conclusion

I hope I managed to showcase the core principles behind WebGL2 with this demo. As you can see, the API itself is low-level and requires quite a bit of typing, yet at the same time is really powerful and let's you draw complex scenes with fine-grained control over the rendering.

Writing production ready WebGL requires even more typing, checking for optional features / extensions and handling missing extensions and fallbacks, so I would advise you to use a framework. At the same time, I believe it is important to understand the key concepts behind the API so you can successfully use higher level libraries like threejs and dig into their internals if needed.

I am a big fan of twgl, which hides away much of the verbosity of the API, while still being really low level with a small footprint. This demo's code can easily be reduced by more then half by using it.

I encourage you to experiment around with the code after reading this article, plug in different values, change the order of things, add more draw commands and what not. I hope you walk away with a high level understanding of core WebGL2 API and how it all ties together, so you can learn more on your own.

Find this project on Github

Georgi Nikolov

I am a frontend developer living and working in Berlin. I specialise in developing rich user interfaces and graphics,

such as websites, web apps, animations and visualisations. During my spare time, I learn and strive to improve my programming skills and math for computer graphics / animation.

https://georgi-nikolov.com/

Twitter Codepen Github

Audio-based Image Distortion Effects with WebGL

Interactive WebGL Hover Effects

How to Unroll Images with Three.js