Animating a million letters using Three.js

- Writing efficient WebGL code
  - Drawing multiple objects using a single draw call
- Setting up the geometry and textures
- Vertex shader for animating the letters
- Remaining concerns
- Summary
- References

My goal in this article is to draw a million animated letters on the screen at a smooth frame rate. This task should be quite possible with modern GPUs. Each letter consists of two textured triangles, so we're only talking about two million triangles per frame.

If you're coming from a traditional JavaScript animation background, this all sounds like madness. Two million triangles updated every frame is definitely not something you would like to do with JavaScript today. But thankfully we have WebGL, which lets us tap into the awesome power of modern GPUs. And two million animated triangles is quite doable with a modern GPU and some shader magic.

Writing efficient WebGL code

Writing efficient WebGL code requires a certain mindset. The usual way to draw using WebGL is to set up your uniforms, buffers and shaders for each object, followed by a call to draw the object. This way of drawing works when drawing a small number of objects. To draw a large number of objects, you should minimize the amount of WebGL state changes. To start with, draw all objects using the same shader after each other, so that you don't have to change shaders between objects. For simple objects like particles, you could bundle several objects into a single buffer and edit it using JavaScript. That way you'd only have to reupload the vertex buffer instead of changing shader uniforms for every single particle.

But to go really fast, you need to push most of your computation to the shaders. That's what I'm trying to do here. Animate a million letters using shaders.

The article's code uses the Three.js library, which abstracts away all the tedious boilerplate from writing WebGL code. Instead of having to write hundreds of lines of WebGL state setup and error handling, with Three.js you only need to write a couple lines of code. It's also easy to tap into the WebGL shader system from Three.js.

Drawing multiple objects using a single draw call

Here's a small pseudo-code example of how you might draw multiple objects using a single draw call. The traditional way is to draw one object at a time like this:

```
for (var i=0; i<objects.length; i++) {
  // each added object requires a separate WebGL draw call
  scene.add(createNewObject(objects[i]));
}
renderer.render(scene, camera);
```

But the above method requires a separate draw call for each object. To draw multiple objects at once, you can bundle the objects into a single geometry and get away with a single draw call:

```
var geo = new THREE.Geometry();
for (var i=0; i<objects.length; i++) {
  // bundle the objects into a single geometry
  // so that they can be drawn with a single draw call
  addObjectToGeometry(geo, objects[i]);
}
// GOOD! Only one object to add to the scene!
scene.add(new THREE.Mesh(geo, material));
renderer.render(scene, camera);
```

Alright, now that you've got the basic idea, let's get back to writing the demo and start animating those million letters!

Setting up the geometry and textures

As the first step, I'm going to create a texture with the letter bitmaps on it. I'm using the 2D canvas for this. The resulting texture has all the letters I want to draw. The next step is to create a buffer with the texture coordinates to the letter sprite sheet. While this is an easy and straightforward method to set up the letters, it’s a bit wasteful as it uses two floats per vertex for the texture coordinates. A shorter way - left as an exercise to the reader - would be to pack the letter index and corner index into one number and convert that back to texture coordinates in the vertex shader.

Here's how I build the letter texture using Canvas 2D:

```
var fontSize = 16;

// The square letter texture will have 16 * 16 = 256 letters, enough for all 8-bit characters.
var lettersPerSide = 16;

var c = document.createElement('canvas');
c.width = c.height = fontSize*lettersPerSide;
var ctx = c.getContext('2d');
ctx.font = fontSize+'px Monospace';

// This is a magic number for aligning the letters on rows. YMMV.
var yOffset = -0.25;

// Draw all the letters to the canvas.
for (var i=0,y=0; y<lettersPerSide; y++) {
  for (var x=0; x<lettersPerSide; x++,i++) {
    var ch = String.fromCharCode(i);
    ctx.fillText(ch, x*fontSize, yOffset*fontSize+(y+1)*fontSize);
  }
}

// Create a texture from the letter canvas.
var tex = new THREE.Texture(c);
// Tell Three.js not to flip the texture.
tex.flipY = false;
```

```
// And tell Three.js that it needs to update the texture.
tex.needsUpdate = true;
```

I also upload the triangle array to the GPU. These vertices are used by the vertex shader to put the letters on the screen. The vertices are set to the letter positions in the text so that if you render the triangle array as-is, you get a basic layout rendering of the text.

Creating the geometry for the book:

```
var geo = new THREE.Geometry();

var i=0, x=0, line=0;
for (i=0; i<BOOK.length; i++) {
  var code = BOOK.charCodeAt(i); // This is the character code for the current letter.
  if (code > lettersPerSide * lettersPerSide) {
    code = 0; // Clamp character codes to letter map size.
  }
  var cx = code % lettersPerSide; // Cx is the x-index of the letter in the map.
  var cy = Math.floor(code / lettersPerSide); // Cy is the y-index of the letter in the map.

  // Add letter vertices to the geometry.
  var v,t;
  geo.vertices.push(
    new THREE.Vector3( x*1.1+0.05, line*1.1+0.05, 0 ),
    new THREE.Vector3( x*1.1+1.05, line*1.1+0.05, 0 ),
    new THREE.Vector3( x*1.1+1.05, line*1.1+1.05, 0 ),
    new THREE.Vector3( x*1.1+0.05, line*1.1+1.05, 0 )
  );
  // Create faces for the letter.
  var face = new THREE.Face3(i*4+0, i*4+1, i*4+2);
  geo.faces.push(face);
  face = new THREE.Face3(i*4+0, i*4+2, i*4+3);
  geo.faces.push(face);

  // Compute texture coordinates for the letters.
  var tx = cx/lettersPerSide,
      ty = cy/lettersPerSide,
      off = 1/lettersPerSide;
  var sz = lettersPerSide*fontSize;
  geo.faceVertexUvs[0].push([
```

```
      new THREE.Vector2( tx, ty+off ),
      new THREE.Vector2( tx+off, ty+off ),
      new THREE.Vector2( tx+off, ty )
    ]);
    geo.faceVertexUvs[0].push([
      new THREE.Vector2( tx, ty+off ),
      new THREE.Vector2( tx+off, ty ),
      new THREE.Vector2( tx, ty )
    ]);

    // On newline, move to the line below and move the cursor to the start of the line.
    // Otherwise move the cursor to the right.
    if (code == 10) {
      line--;
      x=0;
    } else {
      x++;
    }
  }
```

Vertex shader for animating the letters

With a simple vertex shader, I get a flat view of the text. Nothing fancy. Runs well, but if I want to animate it, I need to do the animation in JavaScript. And JavaScript is kinda slow for animating the six million vertices involved, especially if you want to do it on every frame. Maybe there is there a faster way.

Why yes, we can do procedural animation. What that means is that we do all our position and rotation math in the vertex shader. Now I don't need to run any JavaScript to update the positions of the vertices. The vertex shader runs very fast and I get a smooth frame rate even with a million triangles being individually animated every frame. To address the individual triangles, I round down the vertex coordinates so that all four points of a letter quad map to a single unique coordinate. Now I can use this coordinate to set the animation parameters for the letter in question.

To be able to successfully round down coordinates, coordinates from two different letters can't overlap. The easiest way to do this is by using square letter quads with a small offset separating the letter from the one on its right side and the line above it. For example, you could use a width and height of 0.5 for the letters and align the letters on integer coordinates. Now when you round

down the coordinate of any letter vertex, you get the bottom-left coordinate of the letter.

Rounding down vertex coordinates to find the top-left corner of a letter.

Rounding down vertex coordinates to find the top-left corner of a letter.

To better understand the animated vertex shader, I'm going to go through a simple run-of-the-mill vertex shader first. This is what normally happens when you draw a 3D model to the screen. The vertices of the model are transformed by a couple transformation matrices to project each 3D vertex onto the 2D screen. Whenever a triangle defined by three of these vertices lands inside the viewport, the pixels it covers are processed by the fragment shader to color them. Anyway, here's the simple vertex shader:

```
varying float vUv;

void main() {
  // modelViewMatrix, position and projectionMatrix are magical
  // attributes that Three.js defines for us.

  // Transform current vertex by the modelViewMatrix
  // (bundled model world position & camera world position matrix).
  vec4 mvPosition = modelViewMatrix * position;

  // Project camera-space vertex to screen coordinates
  // using the camera's projection matrix.
  vec4 p = projectionMatrix * mvPosition;

  // uv is another magical attribute from Three.js.
  // We're passing it to the fragment shader unchanged.
  vUv = uv;

  gl_Position = p;
}
```

And now, the animated vertex shader. Basically, it does the same thing as the simple vertex shader, but with a small twist. Instead of transforming each vertex by just the transformation matrices, it applies a time-dependent animated transformation as well. To make each letter animate a bit differently, the animated vertex shader also modifies the animation based on the coordinates of the letter. It's going to look a good deal more complicated than the simple vertex shader because, well, it *is* more complicated.

```
uniform float uTime;
uniform float uEffectAmount;
```

```
varying float vZ;
varying vec2 vUv;

// rotateAngleAxisMatrix returns the mat3 rotation matrix
// for given angle and axis.
mat3 rotateAngleAxisMatrix(float angle, vec3 axis) {
  float c = cos(angle);
  float s = sin(angle);
  float t = 1.0 - c;
  axis = normalize(axis);
  float x = axis.x, y = axis.y, z = axis.z;
  return mat3(
    t*x*x + c,    t*x*y + s*z,  t*x*z - s*y,
    t*x*y - s*z,  t*y*y + c,    t*y*z + s*x,
    t*x*z + s*y,  t*y*z - s*x,  t*z*z + c
  );
}

// rotateAngleAxis rotates a vec3 over the given axis by the given angle and
// returns the rotated vector.
vec3 rotateAngleAxis(float angle, vec3 axis, vec3 v) {
  return rotateAngleAxisMatrix(angle, axis) * v;
}

void main() {
  // Compute the index of the letter (assuming 80-character max line length).
  float idx = floor(position.y/1.1)*80.0 + floor(position.x/1.1);

  // Round down the vertex coords to find the bottom-left corner point of the letter.
  vec3 corner = vec3(floor(position.x/1.1)*1.1, floor(position.y/1.1)*1.1, 0.0);

  // Find the midpoint of the letter.
  vec3 mid = corner + vec3(0.5, 0.5, 0.0);

  // Rotate the letter around its midpoint by an angle and axis dependent on
  // the letter's index and the current time.
  vec3 rpos = rotateAngleAxis(idx+uTime,
    vec3(mod(idx,16.0), -8.0+mod(idx,15.0), 1.0), position - mid) + mid;
```

```
  // uEffectAmount controls the amount of animation applied to the letter.
  // uEffectAmount ranges from 0.0 to 1.0.
  float effectAmount = uEffectAmount;

  vec4 fpos = vec4( mix(position,rpos,effectAmount), 1.0 );
  fpos.x += -35.0;

  // Apply spinning motion to individual letters.
  fpos.z += ((sin(idx+uTime*2.0)))*4.2*effectAmount;
  fpos.y += ((cos(idx+uTime*2.0)))*4.2*effectAmount;

  vec4 mvPosition = modelViewMatrix * fpos;

  // Apply wavy motion to the entire text.
  mvPosition.y += 10.0*sin(uTime*0.5+mvPosition.x/25.0)*effectAmount;
  mvPosition.x -= 10.0*cos(uTime*0.5+mvPosition.y/25.0)*effectAmount;

  vec4 p = projectionMatrix * mvPosition;

  // Pass texture coordinates and the vertex z-coordinate to the fragment shader.
  vUv = uv;
  vZ = p.z;

  // Send the final vertex position to WebGL.
  gl_Position = p;
}
```

To use the vertex shader, I use a `THREE.ShaderMaterial`, a material type that lets you use custom shaders and specify uniforms for them. Here's how I'm using THREE.ShaderMaterial in the demo:

```
// First, set up uniforms for the shader.
var uniforms = {

  // map contains the letter map texture.
  map: { type: "t", value: 1, texture: tex },

  // uTime is the urrent time.
  uTime: { type: "f", value: 1.0 },

  // uEffectAmount controls the amount of animation applied to the letters.
```

```
  uEffectAmount: { type: "f", value: 0.0 }
};

// Next, set up the THREE.ShaderMaterial.
var shaderMaterial = new THREE.ShaderMaterial({
  uniforms: uniforms,

  // I have my shaders inside HTML elements like
  // <script id="vertex" type="text/x-glsl-vert">... shaderSource ... <script>

  // The below gets the contents of the vertex shader script element.
  vertexShader: document.querySelector('#vertex').textContent,

  // The fragment shader is a bit special as well, drawing a rotating
  // rainbow gradient.
  fragmentShader: document.querySelector('#fragment').textContent
});

// I set depthTest to false so that the letters don't occlude each other.
shaderMaterial.depthTest = false;
```

On every animation frame, I update the shader uniforms:

```
// I'm controlling the uniforms through a proxy control object.
// The reason I'm using a proxy control object is to
// have different value ranges for the controls and the uniforms.
var controller = {
  effectAmount: 0
};

// I'm using <a href="http://code.google.com/p/dat-gui/">DAT.GUI</a> to do a quick & easy GUI
for the demo.
var gui = new dat.GUI();
gui.add(controller, 'effectAmount', 0, 100);

var animate = function(t) {
  uniforms.uTime.value += 0.05;
  uniforms.uEffectAmount.value = controller.effectAmount/100;
  bookModel.position.y += 0.03;
```

```
  renderer.render(scene, camera);
  requestAnimationFrame(animate, renderer.domElement);
};
animate(Date.now());
```

And there you have it, shader-based animation. It looks pretty complex, but the only thing it really does is move the letters around in a way that depends on the current time and each letter's index. If performance wasn't a concern, you could have this logic running in JavaScript. However, at tens of thousands of animated objects, JavaScript stops being a viable solution.

Remaining concerns

One problem now is that JavaScript doesn’t know about the particle positions. If you really need to know where your particles are, you could duplicate the vertex shader logic in JavaScript and recalculate the vertex positions using a web worker every time you need the positions. That way your rendering thread doesn’t have to wait for the math and you can continue animating at a smooth frame rate.

For more controllable animation, you could use render-to-texture functionality to animate between two sets of positions provided by JavaScript. First, render the current positions to a texture, then animate towards positions defined in a separate texture provided by JavaScript. The nice thing about this is that you can update a small fraction of the JavaScript-provided positions per frame and still continue animating all the letters every frame with the vertex shader tweening the positions.

Another concern is that 256 characters is far too few to do non-ASCII texts. If push the texture map size to 4096x4096 while decreasing the font size to 8px, you can fit the entire UCS-2 character set into the texture map. However, 8px font size is not very readable. To do larger font sizes, you can use multiple textures for your font. See this sprite atlas demo for an example. Another thing that would help is to create only the letters used in your text.

Summary

In this article, I walked you through implementing a vertex shader -based animation demo using Three.js. The demo animates a million letters in real-time on a 2010 MacBook Air. The implementation bundled an entire book into a single geometry object for efficient drawing. The individual letters were animated by figuring out out which vertices belong to which letter and animating the vertices based on the index of the letter in the book text.

References

- Three.js
- Metamorphosis by Franz Kafka on Project Gutenberg
- Sprite Atlas demo

□□□ #3
□ □□□ □□□ 2026-10-31 22:28:35 UTC
□ cocobook-sync □□□ 2026-11-04 17:38:31 UTC