How to Generate a Normal Map from an Image.

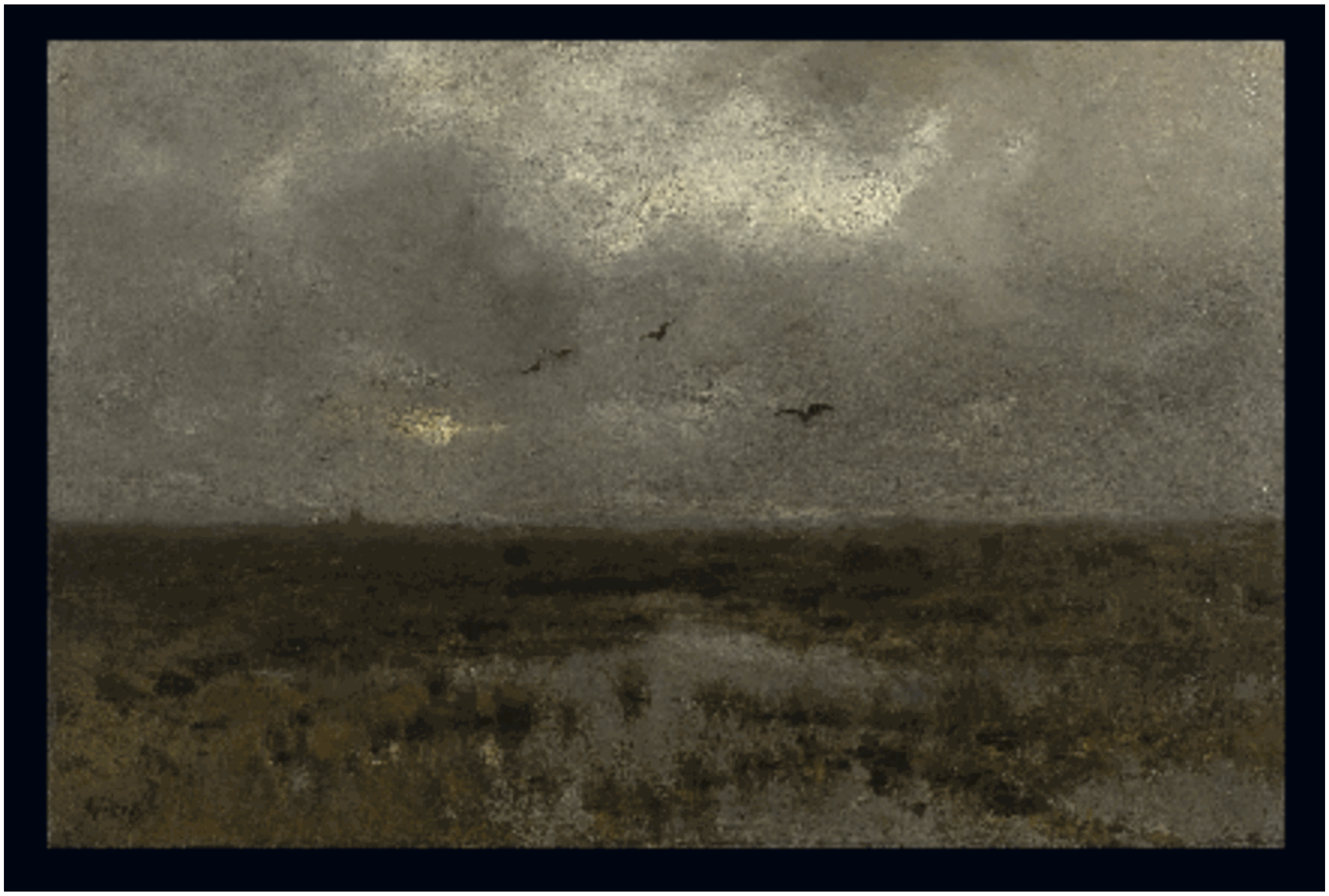
A J Kruschwitz · Follow
6 min read · Oct 9, 2021

I recently built a website that allows you to view paintings from the Rijksmuseum in 3D to demonstrate a different approach to a gallery walkthrough than the standard "google street-view from inside the museum" approach that can be seen at the National Gallery of Art. Our hope was that by rendering each image in it's own space, users would be able to inspect the brushstrokes of the painting up close, a study that would be dangerous and impolite in a real gallery.

For this project, we needed to generate a normal map for each painting on runtime based only off of a high resolution JPG file provided to us by the museum's API. In researching this project, I found a handful of questions on stack overflow and stack exchange detailing working algorithms for normal map generation, but few explanations as to how those algorithms work. So, in this article I will attempt to articulate one quick and dirty algorithm for normal map generation and the math behind it. If you'd rather just look through the code, I've uploaded a repository with all the code shown on my github here.

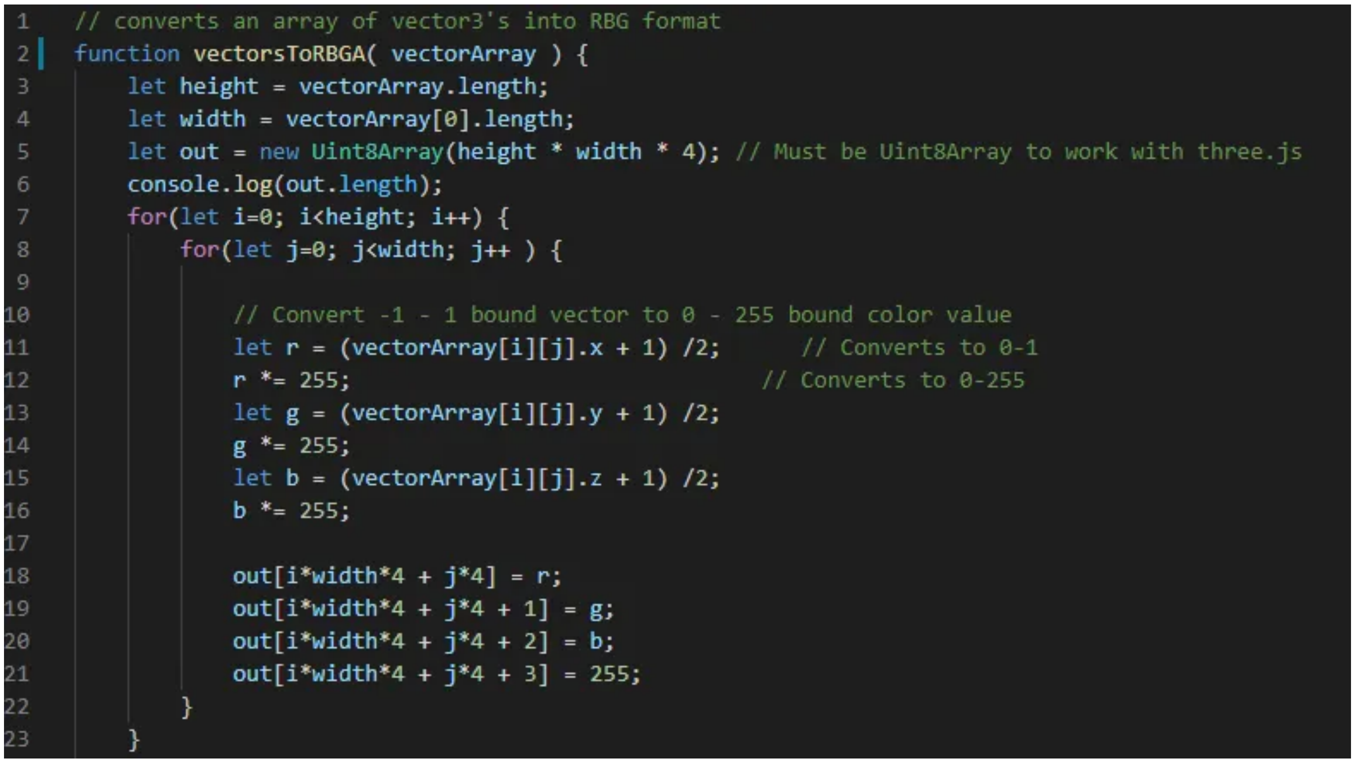
First to get everyone up to speed, what is a normal map? A normal map is a type of texture applied to 3d objects which describes the direction perpendicular to the surface at each point. The shader included in your 3d rendering engine will take that information and calculate which direction light should reflect off of the object at each point. If you'd like a more in depth explanation, check out this page from LearnOpenGL. Here's an example of a painting with and without a normal map applied. Look for shadows from the paint strokes in the second image.

Painting without normal map


Painting with normal map applied

Normal maps are used to create the illusion of depth. The R, G, and B values of the image represent the X, Y, and Z components of the normal vector at that pixel respectively. These are unit vectors, so a component vector will have a length of between -1 and 1, but this value must be represented in the image with the bounds of 0–255. The light blue color taking up the majority of the image represents a flat surface, or a normal vector pointing straight up, (0, 0, 1). The pixel color in the normal map is (127, 127, 255).

Stepping into JavaScript to start to implement this algorithm, we already know we will need a few functions. First, we will need some code to convert our vectors into RGB values. To make the code more readable, instead of using a 1d array of vectors, we use a 2d array, where each nested array contains one row of pixels.

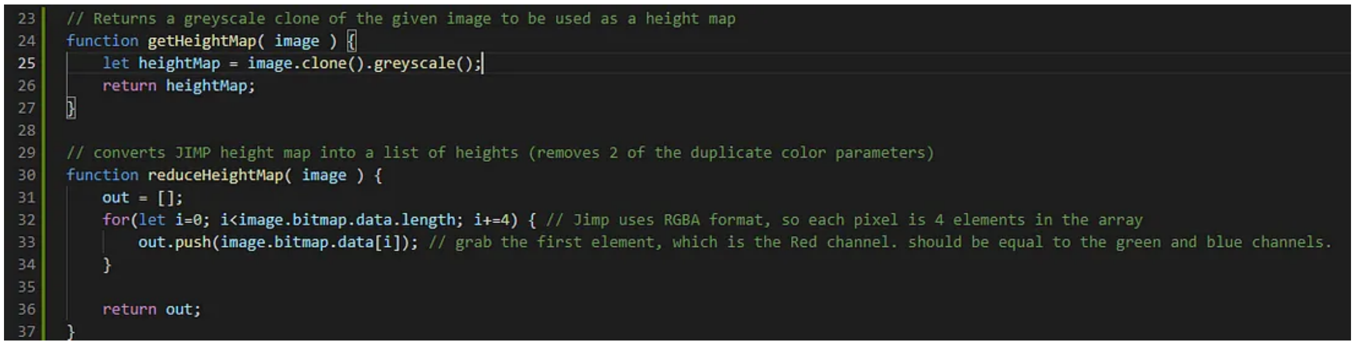


```
// converts an array of vector3's into RBG format
function vectorsToRBGA( vectorArray ) {
    let height = vectorArray.length;
    let width = vectorArray[0].length;
    let out = new Uint8Array(height * width * 4); // Must be Uint8Array to work with three.js
    console.log(out.length);
    for(let i=0; i<height; i++) {
        for(let j=0; j<width; j++ ) {

            // Convert -1 - 1 bound vector to 0 - 255 bound color value
            let r = (vectorArray[i][j].x + 1) /2;      // Converts to 0-1
            r *= 255;                                  // Converts to 0-255
            let g = (vectorArray[i][j].y + 1) /2;
            g *= 255;
            let b = (vectorArray[i][j].z + 1) /2;
            b *= 255;

            out[i*width*4 + j*4] = r;
            out[i*width*4 + j*4 + 1] = g;
            out[i*width*4 + j*4 + 2] = b;
            out[i*width*4 + j*4 + 3] = 255;
        }
    }
```

Code for converting a 2d array of vectors to a 1d RGBA format array

Next, we know we will need to normalize our vectors. Luckily, my 3D rendering engine three.js has built in vector tools, so we won't need to worry about implementing this ourselves.

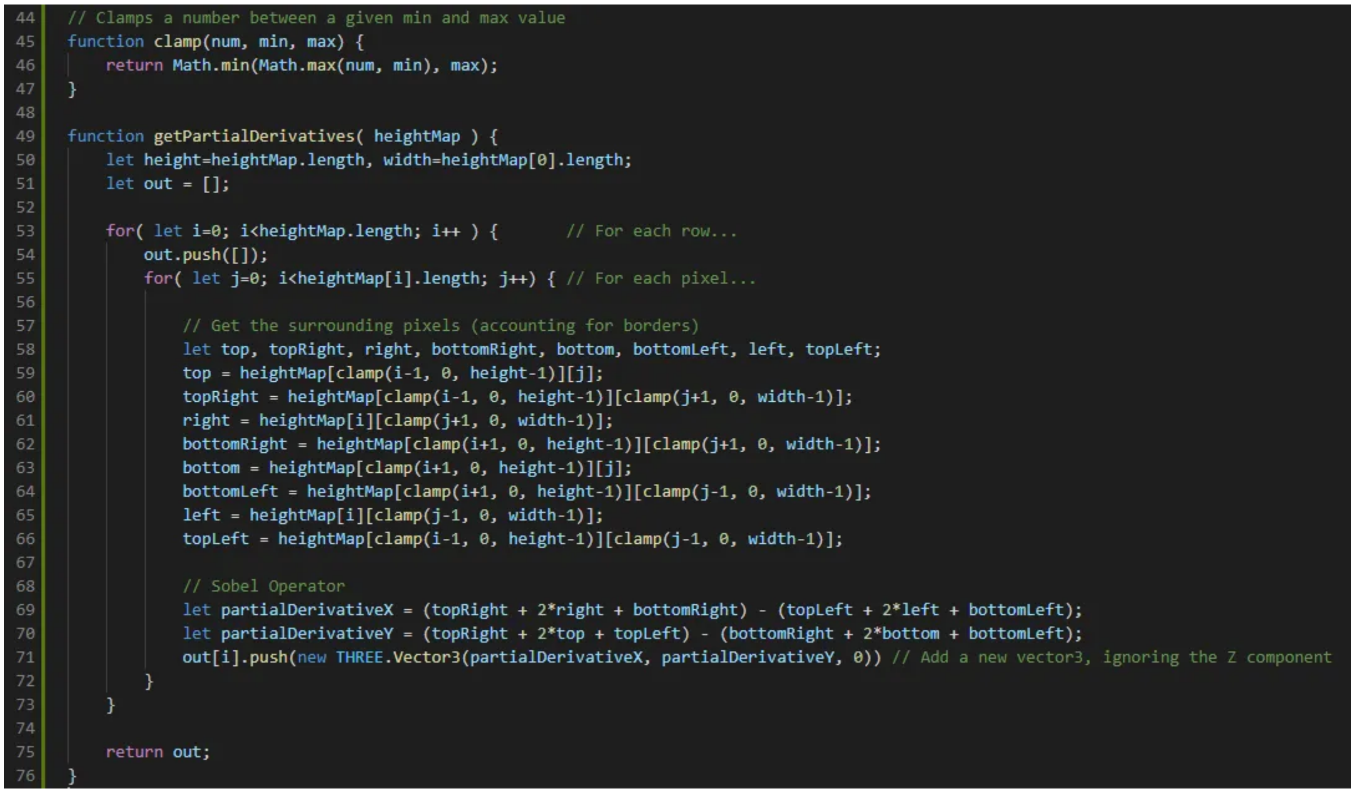
Finally, we're left with the problem of calculating our normal vectors at each point, and this is where we'll need to make some concessions. In order to calculate which direction the object is facing at a given point, we will need to know the height of the image at that point. Unfortunately, there is no way to calculate the actual height of an object at each point from a single image (In my research I came across this website by CPetry on GitHub, which can generate a normal map based on 4 images of an object in different lighting conditions for a more accurate result), but we can make some assumptions that will allow us to get a cheap and dirty height map. If we assume that the lighter parts of the image are higher, we can convert the image to greyscale and have all the data we need to generate our height map, which will be used to generate the normal map.



```
// Returns a greyscale clone of the given image to be used as a height map
function getHeightMap( image ) {
    let heightMap = image.clone().greyscale();
    return heightMap;
}

// converts JIMP height map into a list of heights (removes 2 of the duplicate color parameters)
function reduceHeightMap( image ) {
    out = [];
    for(let i=0; i<image.bitmap.data.length; i+=4) { // Jimp uses RGBA format, so each pixel is 4 elements in the array
        out.push(image.bitmap.data[i]); // grab the first element, which is the Red channel. should be equal to the green and blue channels.
    }

    return out;
}
```

Code for converting an image to a height map

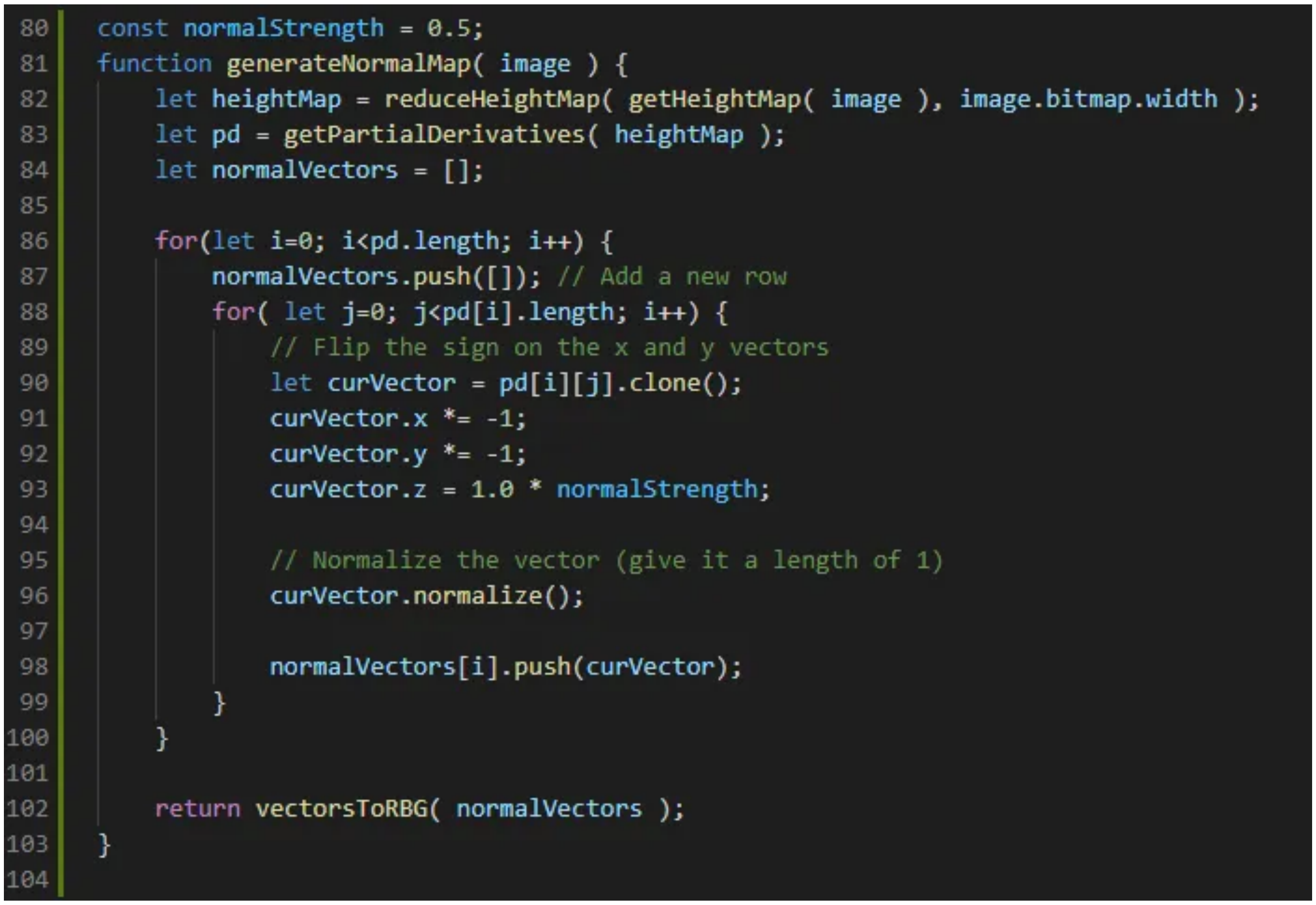
With our height map generated, we can begin calculating our normal map. We can come up with a vector tangent to the object at any given point by calculating the partial derivative, or the rate of change in height, at each point in the X and Y direction. We will be using a symmetric derivative to calculate this. Essentially, the symmetric derivative checks the height shortly before the given point, and subtracts it from the height shortly after a given point to come up with a linear rate of change between those two points. For now, we will leave out the Z component of this tangent vector because it will have to be estimated later. We can calculate these derivatives using the Sobel Operator.



```
// Clamps a number between a given min and max value
function clamp(num, min, max) {
    return Math.min(Math.max(num, min), max);
}

function getPartialDerivatives( heightMap ) {
    let height=heightMap.length, width=heightMap[0].length;
    let out = [];

    for( let i=0; i<heightMap.length; i++ ) {       // For each row...
        out.push([]);
        for( let j=0; i<heightMap[i].length; j++) { // For each pixel...

            // Get the surrounding pixels (accounting for borders)
            let top, topRight, right, bottomRight, bottom, bottomLeft, left, topLeft;
            top = heightMap[clamp(i-1, 0, height-1)][j];
            topRight = heightMap[clamp(i-1, 0, height-1)][clamp(j+1, 0, width-1)];
            right = heightMap[i][clamp(j+1, 0, width-1)];
            bottomRight = heightMap[clamp(i+1, 0, height-1)][clamp(j+1, 0, width-1)];
            bottom = heightMap[clamp(i+1, 0, height-1)][j];
            bottomLeft = heightMap[clamp(i+1, 0, height-1)][clamp(j-1, 0, width-1)];
            left = heightMap[i][clamp(j-1, 0, width-1)];
            topLeft = heightMap[clamp(i-1, 0, height-1)][clamp(j-1, 0, width-1)];

            // Sobel Operator
            let partialDerivativeX = (topRight + 2*right + bottomRight) - (topLeft + 2*left + bottomLeft);
            let partialDerivativeY = (topRight + 2*top + topLeft) - (bottomRight + 2*bottom + bottomLeft);
            out[i].push(new THREE.Vector3(partialDerivativeX, partialDerivativeY, 0)) // Add a new vector3, ignoring the Z component
        }
    }

    return out;
}
```

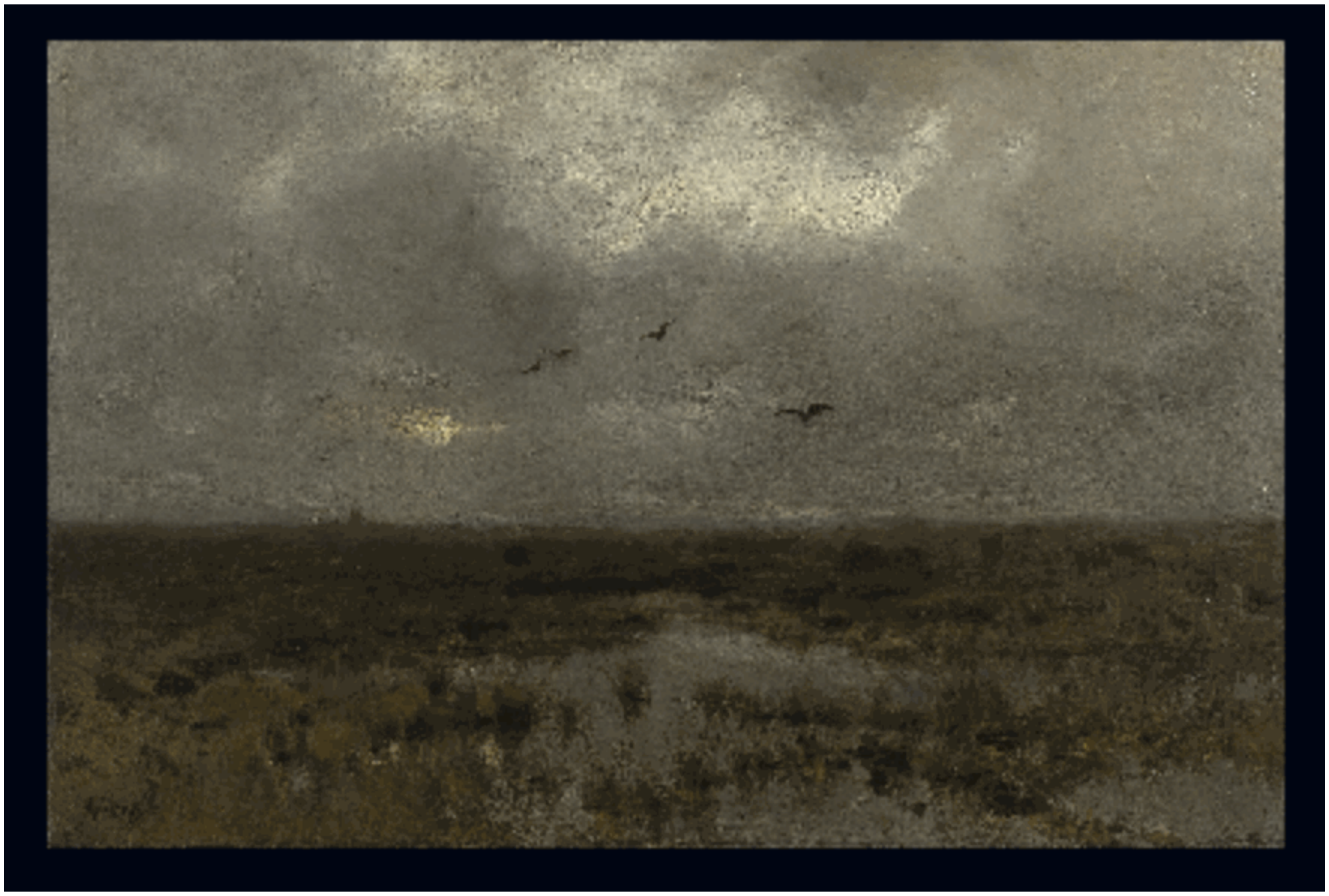
Code for generating the partial derivatives necessary for normal map generation

Now that we have these derivatives, we can get a rough approximation of the normal vector by simply multiplying each value by -1. This way, The x and y components are set exactly opposite the tangent vector, which will get us almost all the way to our normal vector! Now we need to calculate a Z value. The easiest way to do this, and the way which gives the most control to 3D artists if they were to be using your software, is to simply estimate the Z component as 1 divided by some strength value greater than 0. When the strength is very high, the Z component will be very low, and even minor changes in the X and Y vector components will show a large change in direction. When strength is low, the Z component will be higher, which will dilute the X and Y components of the vector when normalizing. This Z estimation is not the most accurate, but does provide quite a bit of control on your part. If you find that your normal maps don't appear as you'd like them, you can modify the strength value to compensate. So, when we put it all together, our algorithm will look like this…



```
const normalStrength = 0.5;
function generateNormalMap( image ) {
    let heightMap = reduceHeightMap( getHeightMap( image ), image.bitmap.width );
    let pd = getPartialDerivatives( heightMap );
    let normalVectors = [];

    for(let i=0; i<pd.length; i++) {
        normalVectors.push([]); // Add a new row
        for( let j=0; j<pd[i].length; i++) {
            // Flip the sign on the x and y vectors
            let curVector = pd[i][j].clone();
            curVector.x *= -1;
            curVector.y *= -1;
            curVector.z = 1.0 * normalStrength;

            // Normalize the vector (give it a length of 1)
            curVector.normalize();

            normalVectors[i].push(curVector);
        }
    }

    return vectorsToRBG( normalVectors );
}
```

And we will get results that, when applied to our object, look like this…



So there we are! a normal map generated and applied. Unfortunately, this method has a few substantial drawbacks. First, because we assumed that brighter things in the original image were higher for our height map, our algorithm will struggle with dark foreground objects on a dark background. The effect is subtle enough that this didn't pose many problems for our use case, but it may cause issues for other applications. To fix this, you could use a more sophisticated height map generation algorithm or if possible start with an accurate height map for your texture. The current code also generates these images on runtime, which for larger images can take quite a while (2–3 seconds generally). Using different tools, you could leverage GPU power to run these calculations a bit faster, and you could use a more lightweight data type than Three.js's Vector3 to store your vectors.

Thank you for reading, if you'd like to get in touch feel free to reach out to me at a.j.kruschwitz@gmail.com, and have a great day!

JavaScript 3d Threejs Shaders