jayconrod.com

About Me

I'm a software engineer in New York City. I created the the Gypsum programming language, and I have worked on optimizing the V8 JavaScript engine for mobile phones. Most of my writing focuses on compilers and programming languages. I'm also interested in 3D graphics, high performance computing, cryptography, and security.

Subscribe

@jayconrod

E-mail me

Recent Posts

Writing Bazel rules: moving logic to execution

Writing Bazel rules: data and runfiles

Writing Bazel rules: library rule, depsets, providers

Writing Bazel rules: simple binary rule

An update on Gypsum and CodeSwitch

All Posts

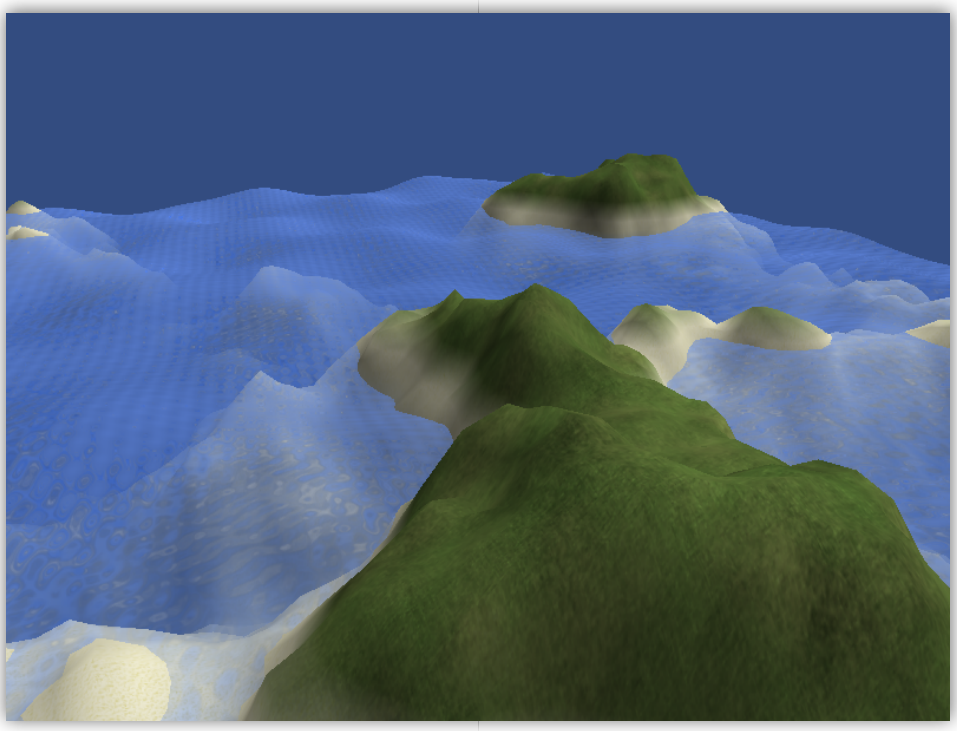
Tags

compilers (23) gypsum (22) codeswitch (9) python (9) virtual-machines (8) fenris (7) bazel (6) go (5) javascript (5) linux (5) v8 (5) emacs (4) imp (4) optimization (4) scala (4) web (4) 3d (3) parsers (3) android (2) bash (2) debugging (2) garbage-collection (2) opengl (2) benchmarks (1)

Water simulation demo in WebGL

Published on 2012-05-27
Edited on 2012-06-17
Tagged: 3d webgl

It's been a while since I played with anything graphics related. Since I'm doing a lot more JavaScript stuff these days at work (I optimize the V8 JavaScript engine used in the Android browser), I figured I'd throw together a more elaborate WebGL demo **than last time**. I did another **water simulation** a couple years ago, but this one is fancier and completely interactive!



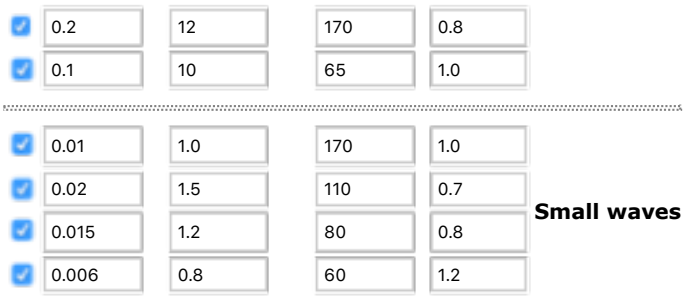
Instructions:

- Rotate the camera by dragging the mouse.
- Zoom in and out with the mouse wheel
- Adjust the waves using the form below

Click here for full page demo

	Amplitude	Wavelength	Direction	Speed	
☑	0.5	20	45	0.4	**Big waves**
☑	0.4	15	120	0.8	

c++ (1) concurrency (1) functional-programming (1) gdb (1) interpreter (1) java (1) machine-learning (1) object-oriented (1) parallelization (1) testing (1) webgl (1)



What you see above is done entirely in HTML and JavaScript. No Flash involved, no browser plug-ins required.

The water is a simple sum of eight 2D sine waves. Every parameter of every wave can be adjusted using the form above. The vertex shader uses the first four rows (big waves) to compute the height (Z coordinate) of the water at each vertex. The water mesh itself is a 256x256 grid, which is not high enough resolution for smaller waves. The fragment shader uses all eight rows to compute the normal and the reflection vector based on the derivative of the waves and the camera position. The reflection vector is used to access a regular 2D tiled sky texture. Try turning off all the waves to see it. You could also use a skybox for this, but this works well enough for me.

The reflectance of the water is determined by the steepness of the camera angle with respect to the surface of the water. This is done per pixel, and it takes the waves into account. I originally implemented **Fresnel equations** for this. Although it gave physically accurate results, they didn't really look the way I wanted. I ended up settling on a quick approximation using the sine of the angle.

You'll notice that as the water gets deeper, the terrain gets less visible. This is a fog effect computed when the terrain is drawn. You can still see it if you turn off the water rendering. The amount of fog is proportional to the underwater distance between the camera and the terrain. You could probably simplify this by just using the water depth, but I think this looks better.

The terrain itself is drawn using a simple 256x256 height map I made in GIMP and a pre-generated normal map. The terrain mesh is just a grid of 2D points; the Z coordinate is pulled out of the height map by the vertex shader. I also made the sand and grass textures in GIMP.

Unfortunately, WebGL is not a simple library to use directly. Most people use some kind of framework to interact with it. I wanted to understand it better though, so I ended up writing my own framework. Here are some links to the JavaScript files. They are BSD licensed, so feel free to use them in your own projects.

- **vector.js** - a basic 3-vector library
- **matrix.js** - a basic 4x4 matrix library with some useful transformations
- **camera.js** - manages a camera which rotates around a sphere
- **shader.js** - extracts and compiles GLSL vertex and fragment shaders. Also locates uniforms and attributes.
- **buffer.js** - manages vertex buffer objects
- **texture.js** - loads textures from images
- **mesh.js** - ties it all together

I encourage you to hit "View Source" in your browser and look at `initTerrainMesh` or `initWaterMesh` to see how these are used. Basically, you create a new `Mesh` or `IndexedMesh` object by giving a WebGL primitive (e.g., `TRIANGLES`), a shader, and dictionaries of uniforms, attributes, and textures. After that, you just call `mesh.prepare()` to set up everything and `mesh.draw()` to draw the mesh.

The interactive component of the demo is done with JQuery. This is the first time I've used JQuery, and it is really a wonderful framework. When I originally learned JavaScript

(in 1998 or so), there were no frameworks, and the DOM APIs were an incompatible mess. They are still an incompatible mess (and there are more browsers now), but with JQuery, you rarely have to use them directly. Things like this have made me appreciate JavaScript as a language and are winning me over to dynamic languages in general.

I hope you enjoyed this demo. I think technologies like WebGL are really exciting. There are many other emerging HTML5 standards like WebWorkers and WebSockets which are really increasing the capability of web apps. I don't think web apps will ever entirely replace traditional native apps, but I still think they have a bright future.