GPU Powered Wind Visualization With Cesium

RAYMAN NG | April 29, 2019 | Share:

This is a guest post by Rayman Ng about his open source wind map built on top of CesiumJS.

Wind is an important element in studying the weather and climate, and it affects our daily lives in various ways. Analyzing wind is critical in many fields such as climate analysis and wind farm management. Visualizing it is crucial in being able to quickly understand the numerical wind data collected by measurement devices.

There are already some wind visualization applications, like Earth Nullschool and Windy, but unfortunately it seems that none of them can display the terrain, which is important for estimating the effect of wind on a specific location.

windy_terrain.mp4

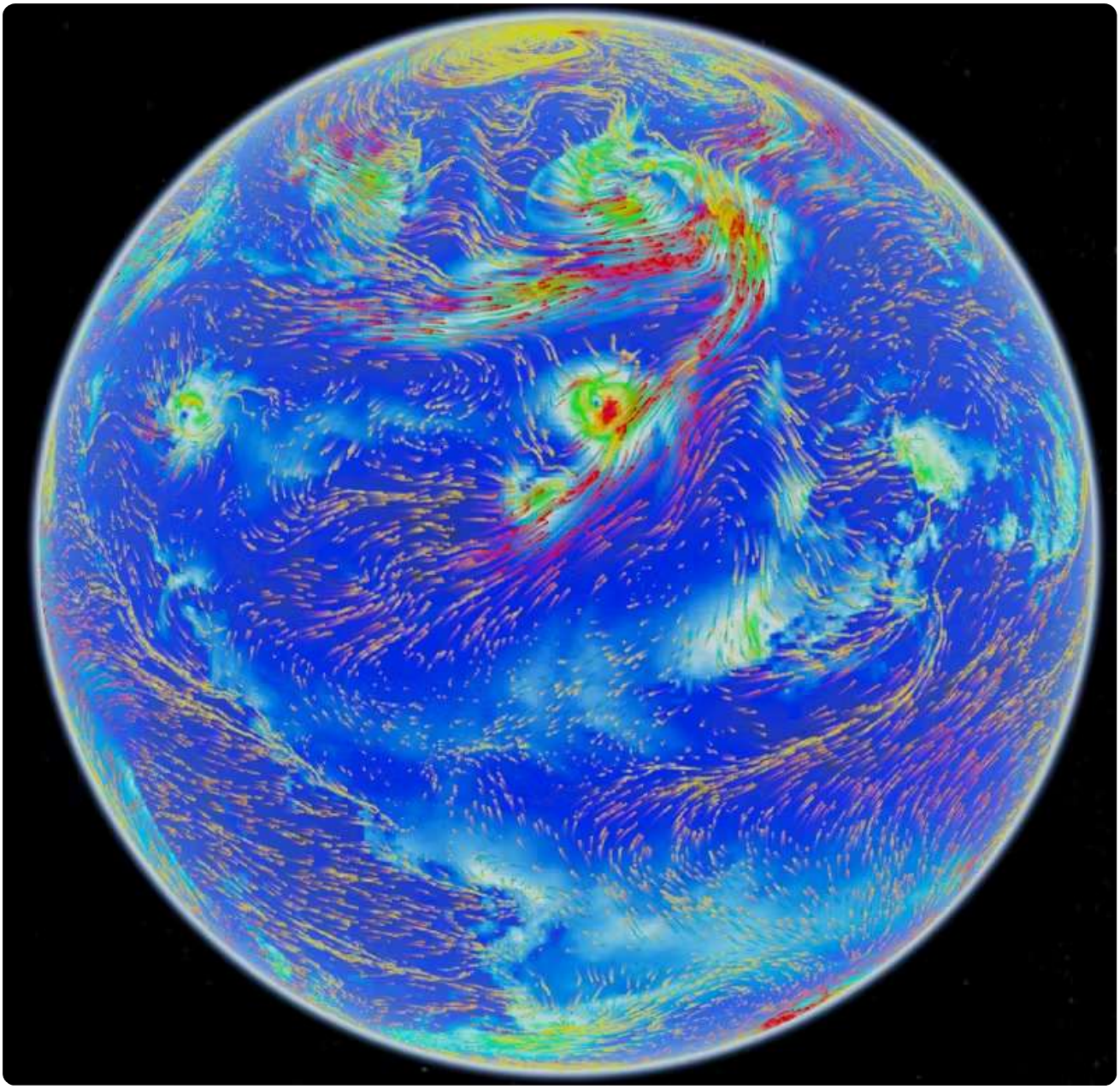
Cesium

00:03

Cesium makes it easy to visualize dynamic wind data with Cesium World Terrain.

Given that I could not find an existing visualization application that met all my requirements, I decided to make my own. I found Cesium, which contains almost everything I need: 3D globe and terrain, Web Map Service layer display, and a powerful rendering engine. Basically I just needed to implement the wind visualization part.

You can run the live demo in your browser or access the source code on GitHub .



Dynamic wind overlaid on WMS layer of wind speed.

How it works

A common technique to visualize wind is to use a particle system, which places lots of particles in the wind field and updates their positions regularly with the wind force. The trails of moving particles

can be used to reveal the flow pattern of the wind.

wind_on_atmosphere.mp4

Cesium

00:03

Computing this many particles is possible using the GPU, since this is otherwise too slow to render in real time.

At first I tried the Entity API to draw the particle trails, but the performance was not satisfying when I placed more than 10,000 particles. After some investigation, I realized that the Entity API performs computation on the CPU, and the computation for more than 10,000 particles is just too much for my CPU. For better performance, I needed to move the computation to the GPU, but I still had to render the trails, which meant working with the low level Cesium Renderer module.

In Cesium, the key object of the rendering procedure is `Cesium.DrawCommand` : it is created in `Cesium.Primitive` , dispatched by `Cesium.Scene` , and executed in the render engine. The `Cesium.DrawCommand` contains everything needed in rendering, for example, `Cesium.Framebuffer` , `Cesium.Texture` , and `Cesium.ShaderProgram` .

To perform custom rendering, a custom `DrawCommand` is required. To build a `DrawCommand` , first initialize its components, which are `ShaderProgram` , `Texture` , `Uniforms` , and `Framebuffer` , piece by piece, and then combine them together to create a `DrawCommand` object. To inject the `DrawCommand` into the render engine, a custom primitive object is needed. It does not need to implement all the methods of `Cesium.Primitive` , only the update, `isDestroyed` , and the destroy method are necessary. The update method will be called before the start of each rendering.

As for the computation on GPU (also known as GPGPU), by using the technique of rendering to texture, it is similar to doing custom rendering; just use a fullscreen quad as vertex shader and write the calculation code in the fragment shader. Fortunately, Cesium already provides the rendering to texture function—all I needed to do was simply pass my fragment shader code to `Cesium.ComputeCommand` and use `Cesium.ComputeEngine` to perform the GPGPU.

The final step is to add the `CustomPrimitive` that contains the custom `DrawCommand` to the `PrimitiveCollection` of `Scene`. Below is a fully commented example of how to create a primitive and add it to the scene with a custom draw command.

```
class CustomPrimitive {
    constructor() {

        // most of the APIs in the renderer module are private,
        // so you may want to read the source code of Cesium to figure out how to initialize th
        // or you can take my wind visualization code as a example (https://github.com/RaymanNg/
        var vertexArray = new Cesium.VertexArray(parameters);
        var primitiveType = Cesium.PrimitiveType.TRIANGLES // you can set it to other values
        var uniformMap = {
            uniformName: function() {
                // return the value corresponding to the name in the function
                // value can be a number, Cesium Cartesian vector, or Cesium.Texture
            }
        }
        var modelMatrix = new Cesium.Matrix4(parameters);
        var shaderProgram = new Cesium.ShaderProgram(parameters);
        var framebuffer = new Cesium.Framebuffer(parameters);
        var renderState = new Cesium.RenderState(parameters);
        var pass = Cesium.Pass.OPAQUE // if you want the command to be executed in other pass,


        this.commandToExecute = new Cesium.DrawCommand({
            owner: this,
            vertexArray: vertexArray,
            primitiveType: primitiveType,
            uniformMap: uniformMap,
            modelMatrix: modelMatrix,
            shaderProgram: shaderProgram,
            framebuffer: framebuffer,
            renderState: renderState,
            pass: pass
        });
    }

    update(frameState) {
        // if (!this.show) return;
        // if you do not want to show the CustomPrimitive, use return statement to bypass the u

        frameState.commandList.push(this.commandToExecute);
    }

    isDestroyed() {
        // return true or false to indicate whether the CustomPrimitive is destroyed
    }

    destroy() {
        // this method will be called when the CustomPrimitive is no longer used
    }
```

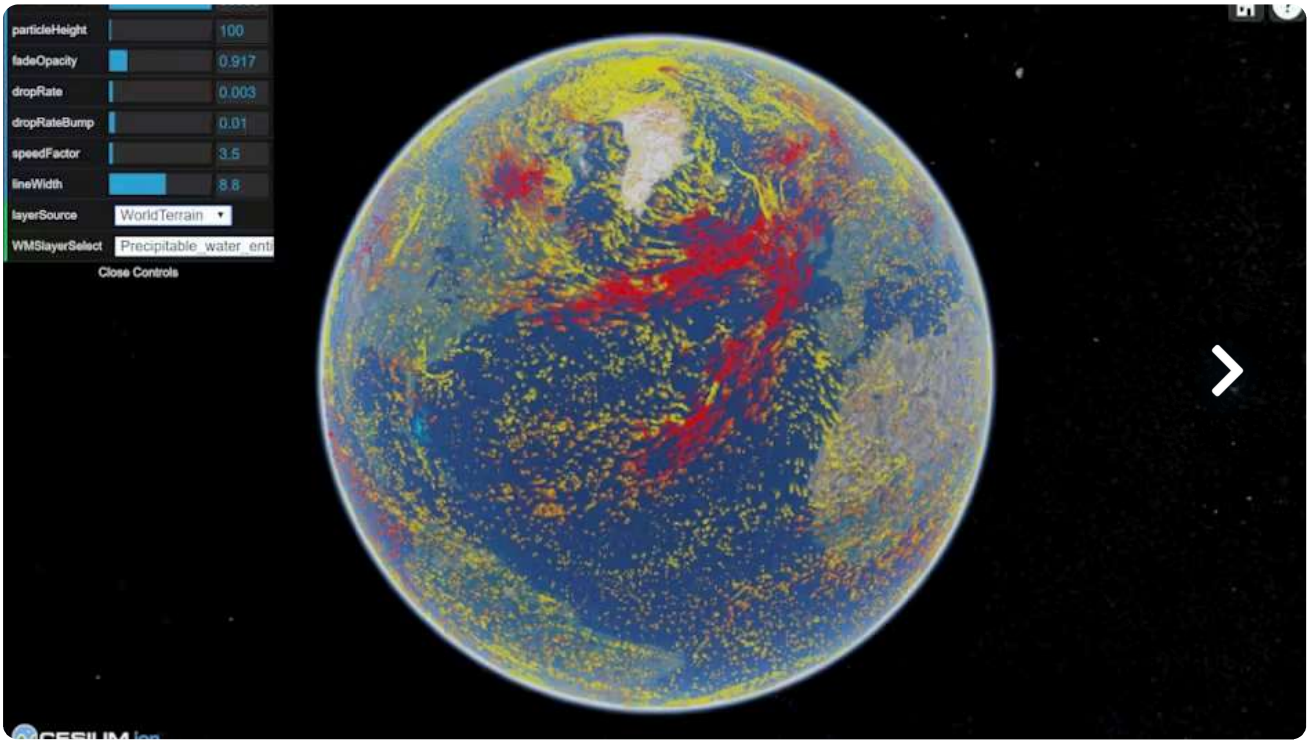
```
}

// To begin the custom rendering, add the CustomPrimitive to the Scene
var viewer = new Cesium.Viewer('cesiumContainer');
var customPrimitive = new CustomPrimitive();
viewer.scene.primitives.add(customPrimitive);
```

Conclusion

Compared with the Entity API, the Primitive API with custom draw commands provides lower level functions, making it possible to achieve better performance but requiring more coding work. By making use of the powerful render engine, wind visualization is possible with satisfying performance, and I believe that there are many other ways to use this powerful yet general render engine.

For example, it was very useful to be able to switch between different imagery layers, so you can see dynamic wind data in its global context and easily switch to comparing it with historic wind data. This is shown below using the slider.



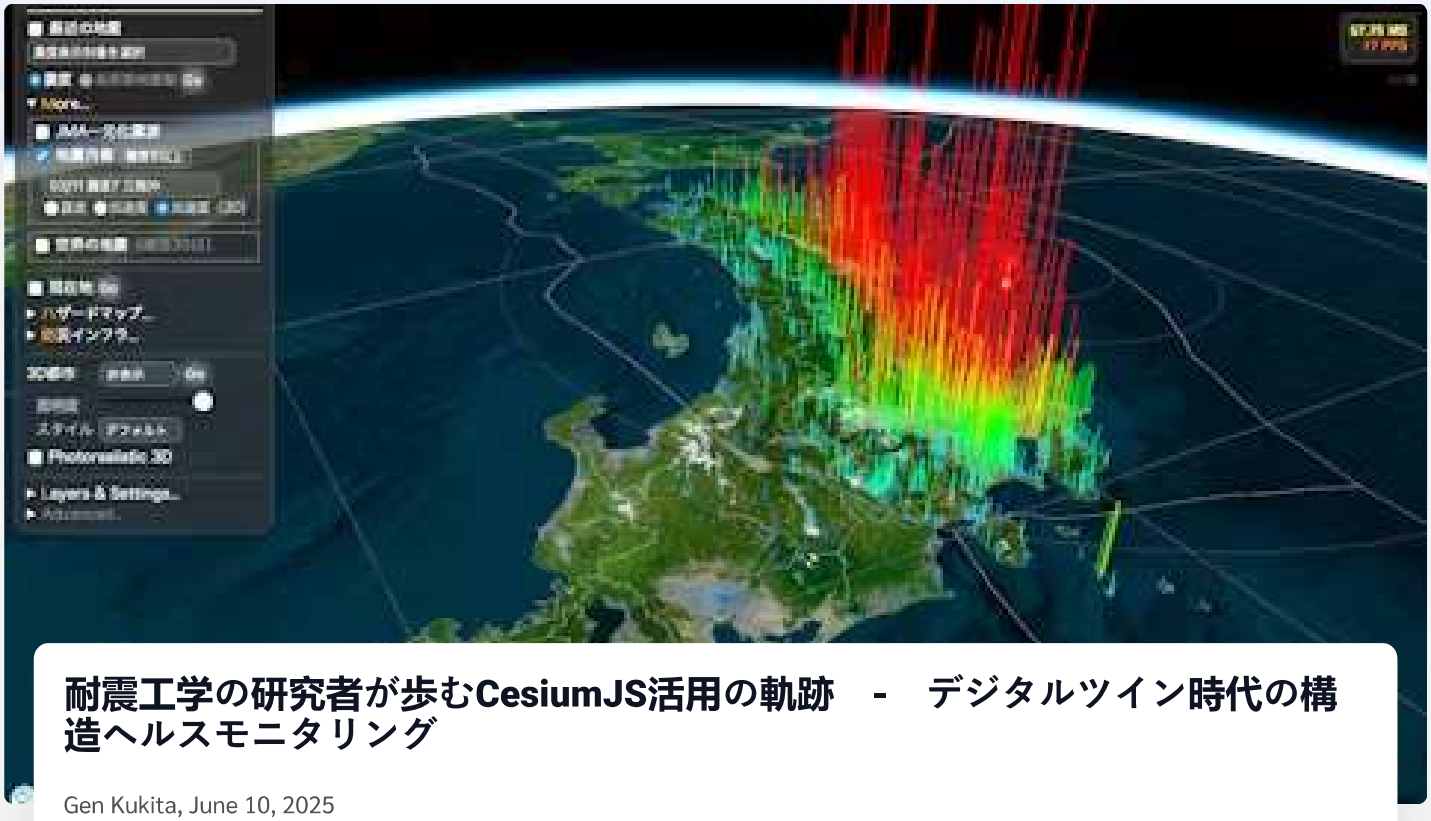
Acknowledgments

I spent several weeks on coding this demo. It would have definitely taken longer without the help of:

- Kind experts in the Cesium Forum, especially Omar Shehata, a member of Cesium team.
- The author of the Cesium custom primitive tutorial on GitHub, which provides code examples for custom rendering in Cesium.

Written by Rayman Ng

Read more



耐震工学の研究者が歩むCesiumJS活用の軌跡　-　デジタルツイン時代の構造ヘルスモニタリング

Gen Kukita, June 10, 2025

In **CERTIFIED DEVELOPER**, **USER STORIES**, **CLIMATE & ENVIRONMENT**, **CESIUMJS**

The Weather Company Brings Forecasts to Military Planning with Cesium

Monica Witzig-Wamsley, March 31, 2025

In **MODELING & SIMULATION**, **CESIUM FOR UNREAL**, **USER STORIES**, **CLIMATE & ENVIRONMENT**, **CESIUMJS**, **FEDERAL & DEFENSE**

Cookie Policy

Server Status