문서번호 APCC-출장-17-0956(2017-08-22)

TRAVEL REPORT FORM
출장보고서

결재	선임연구원	팀장	부서장	부서장
	08/22 이성규	08/22 이현록	08/22 유진호	대결 08/22 김형진
협조				

I. **Reporter** 보고자

Department 소속	Position 직위(직급)	Name 성명	Travel Date 날짜	Note 비고
기후예측본부 정보서비스팀	선임연구원	이성규	08/13 – 08/20	

II. **Major Activities** 주요업무 수행내용

1. Main Contents and Activities 주요내용 및 활동

■ FOSS4G (Free and Open Source Software for Geospatial) BOSTON 2017 학술대회 참석

- 학술대회 장소: 미국 보스턴 Seaport Hotel & World Trade Center
- 학술대회 기간: 2017년 08월 14일 - 18일 (5일)

가. 08.14. (월) - 15. (화): 워크숍 참석

1) 08. 14. (월) 오전: Web 3D Geosptial Made Easy: An Introduction to Cesium 워크숍 참석
강사: Rachael Hwang (AGI)

- Cesium은 AGI (Analytical Graphics, Inc)에서 개발한 WebGL 기반의 3차원 GIS 엔진으로 Apache License 2.0 라이선스로 제공됨.
- Ceisum의 기본기능 (3차원 지도 표출 등), 고급기능 (3D Tiles 표출 등) 등을 실습하였음
- 교재: 붙임 1 참조

2) 08. 14. (월) 오후: Use GDAL and PKTOOLS for raster operations 워크숍 참석
강사: Giuseppe Amatulli (Yale University), Vaclav Petras (North Carolina State University)

- GDAL과 PKTOOLS은 벡터와 래스터자료를 처리하기 위한 소프트웨어 패키지로 GDAL은 X/MIT 라이선스, PKTOOLS은 GNU GPL (GNU General Public License) v3 라이선스로 제공됨.
- Ubuntu 리눅스 환경에서 래스터 자료 처리, 분석 명령어 등을 실습하였음
- 교재: http://spatial-ecology.net/

3) 08. 15. (화) 오전: Introduction to GeoTools 워크숍 참석
강사: Jody Garnett (Community Lead Boundless), Ian Turton (Open Source Envangalist, Astun Technology)

- GeoTools는 공간데이터 (Vector, Raster 등)을 다루기 위한 JAVA 기반의 라이브러리로 GNU LGPL (Lesser General Public License) 라이선스로 제공함.
- Eclipse 개발환경에서 GeoTools 라이브러리를 이용하기 위한 개발환경 구축, Vector 이용하여 Java 프로그래밍 언어에서 Feature, Geometry CRS, Query, Image, Map Style 등을 다루는 프로그래밍을 실습하였음.
- 교재: http://docs.geotools.org/latest/userguide/tutorial/index.html

4) 08. 15. (화) 오후: GeoTools DataStore Workshop 워크숍 참석

강사: Jody Garnett (Community Lead Boundless), Ian Turton (Open Source Envangalist, Astun Technology)

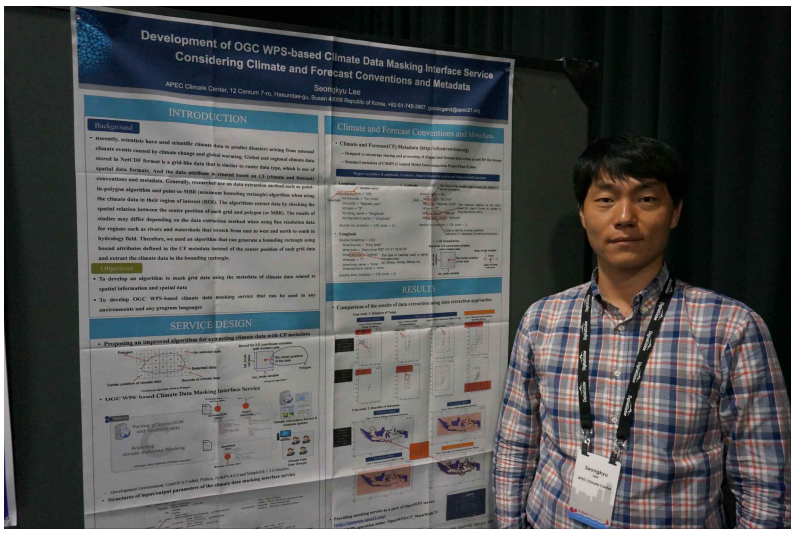
- GeoTools는 공간데이터 (Vector, Raster 등)을 다루기 위한 JAVA 기반의 라이브러리로 GNU LGPL (Lesser General Public License) 라이선스로 제공함.
- GIS 서버인 GeoTools에서 GeoTools 라이브러리를 이용하여 사용자 정의 DataStore를 개발하고 구축하는 과정에 대해 실습하였음.
- 교재: http://docs.geotools.org/latest/userguide/tutorial/datastore/index.html

나. 08.16. (수) - 18. (금): 학술대회 참석 및 발표

1) Academic & Regular Session 참석

2) Poster Session 참석 및 발표

- 발표제목: Development of OGC WPS-based Climate Data Masking Interface Service Considering Climate and Forecast Conventions and Metadata
- 포스터 발표 일정: 붙임 2 참조
- 발표 포스터: 붙임 3 참조



2. Relevance to APEC Climate Center's Activities 결론 및 소감

- 미국 보스턴에서 개최된 금번 국제학술회의는 약 1,140 명의 연구자와 소프트웨어 개발자가 참가하여 다양한 주제의 연구 및 활용사례가 발표되었으며, 한국은 전체 참가자 중에서 3위를 기록하여 국내에서의 FOSS4G에 대한 관심이 높음을 알 수 있었음.
- 전 세계적으로 Free and Open Source Software for Geospatial (FOSS4G) 중에서도 웹 기반의 3차원 엔진/라이브러리와 함께 3차원 서비스에 대한 관심이 증가하고 있었음.
- 워크숍과 학술대회에서 습득한 기술과 정보는 센터 내 과업을 수행하는데 도움이 되었으며, 향후 획

득한 정보 및 인적 네트워크를 연구개발에 적극 활용할 예정임.

III. References 참고사항

(with attachment of any information or report in case of attendance of conferences, workshops and meetings) 학술대회, 워크숍, 회의 등 참석 시 관련 정보 및 문서 첨부

1. Presented and Collected Materials 주요 수집자료

가. FOSS4G BOSTON 2017 워크숍 자료

1) Web 3D Geosptial Made Easy: An Introduction to Cesium
   - 붙임 1 참조
2) Use GDAL and PKTOOLS for raster operations
   - http://spatial-ecology.net/
3) Introduction to GeoTools
   - http://docs.geotools.org/latest/userguide/tutorial/index.html
4) GeoTools DataStore Workshop
   - http://docs.geotools.org/latest/userguide/tutorial/datastore/index.html

나. FOSS4G BOSTON 2017 프로시딩

2. Suggestions and Remarks 건의사항

TUTORIAL (HTTP://CESIUMJS.ORG/TUTORIALS.HTML)

Cesium Workshop

Contents

Overview

Welcome to the Cesium community! We're happy to have you. In order to get you developing your own web map applications as soon as possible, this tutorial will walk you through the development of a simple Cesium application from beginning to end. This tutorial will touch on many of the most important aspects of the Cesium API, but is no means comprehensive (Cesium has a lot of features!). Our goal is to introduce the fundamentals and the tools you'll need to explore the rest of Cesium.

We'll create a simple application for visualizing sample geocache locations in New York City. We'll load and style multiple types of 2D and 3D data and create several camera and display options that a user can set interactively. Finally, as high-tech geocachers, we'll load a 3D drone model to scout the geocache locations for us to take full advantage of our 3D visualization.

By the end of this tutorial, you will have a working overview of Cesium's features and know how to configure a Cesium viewer, load datasets, create and style geometry, work with 3D Tiles, control the camera, and add mouse interactivity to your application.

Our application for interactively visualizing sample geocache locations.

Setup

Just a few setup steps before we can get to development.

1. Make sure your system is Cesium compatible by visiting Hello World (http://cesiumjs.org/Cesium/Apps/HelloWorld.html). No globe? See Troubleshooting (http://cesiumjs.org/troubleshooting.html).
2. Install Node.js (https://nodejs.org/en/).
3. Get the workshop code (https://github.com/AnalyticalGraphicsInc/cesium-workshop). Either clone or download the zip and extract the contents.
4. In your console, navigate to the root `cesium-workshop` directory.
5. Run `npm install`.

6. Run `npm start`.

The console should tell you “Cesium development server running locally. Connect to http://localhost:8080/”. Don’t close the console! We’ll need to keep this process running.

Next, navigate to localhost:8080 (localhost:8080) in your browser. You should see our workshop application up and running. Stuck? The Getting Started Tutorial (http://cesiumjs.org/tutorials/cesium-up-and-running/) goes more in depth about Cesium setup.

The Application Directory

Now for a tour of our application directory! Note that this application directory was designed to be as simple as possible, and totally ignores the many varied modern JS frameworks in use today. But once you have a handle on things, feel free to experiment!

- **Source** : Our application code and data.
- **ThirdParty** : External libraries, in this case just Cesium.
- **LICENSE.md** : Terms of use for this application.
- **index.html** : Our main html page.
- **server.js** : The server we’ll run our application from.

Now take a look at `index.html`. This creates a `div` for our Cesium widget and a few basic input elements. Observe that Cesium Widget is just an ordinary `div` that can be styled and interacted with like any other `div`.

There are a few crucial lines to set this up:

First we include `Cesium.js` in a script tag in the HTML head. This defines the `Cesium` object, which contains the entire Cesium library.

```
<script src="ThirdParty/Cesium/Cesium.js"></script>
```

Cesium ships with a collection of widgets that require this CSS.

```
<style>@import url(ThirdParty/Cesium/Widgets/widgets.css);</style>
```

In the HTML body, we create a `div` for the Cesium Viewer widget.

```
<div id="cesiumContainer"></div>
```

Finally, in another script tag we add the JavaScript for the app at the end of the HTML body.

```
<script src="Source/App.js"></script>
```

That’s about it! The rest of the HTML in this file is for collecting user input, which we’ll use later.

Development Resources

For this tutorial and throughout the rest of your Cesium development career, we encourage you to rely on the following resources:

- Reference Documentation (http://cesiumjs.org/refdoc.html) : A complete guide to the Cesium API containing many code snippets.
- Sandcastle (http://cesiumjs.org/Cesium/Apps/Sandcastle/index.html) : A live-coding environment with a large gallery of code examples.
- Tutorials (http://cesiumjs.org/tutorials.html) : Detailed introductions to areas of Cesium development.
- Cesium Forum (http://cesiumjs.org/forum.html) : A resource for asking and answering Cesium-related questions.

Any time you get stuck, odds are one of these resources will have the answers you're looking for.

The Workflow

To follow along with this tutorial:

1. Open `Source/App.js` in your favorite text editor and delete the contents.
2. Copy into `Source/App.js` the contents of `Source/AppSkeleton.js` , which contains the commented version of the code.
3. Make sure your server is still running in the `cesium-workshop` directory, as described in Setup (http://cesiumjs.org/tutorials/Cesium-Workshop/#setup).
4. Navigate to localhost:8080 (localhost:8080). You should see a mostly black page now.
5. As the tutorial directs you, uncomment code, save `Source/App.js` and refresh the page to see your new changes reflected.

Really stuck? You can follow along in sandcastle with a simplified version of the app (no UI):

- The complete code (http://cesiumjs.org/Cesium/Apps/Sandcastle/?src=Hello%20World.html&label=Showcases&gist=f1f911cf38dd76e1b27a392d6aa389d9)
- The commented code (http://cesiumjs.org/Cesium/Apps/Sandcastle/?src=Hello%20World.html&label=Showcases&gist=d34bc7d5c184e8db6f39d02ea4ba1970)

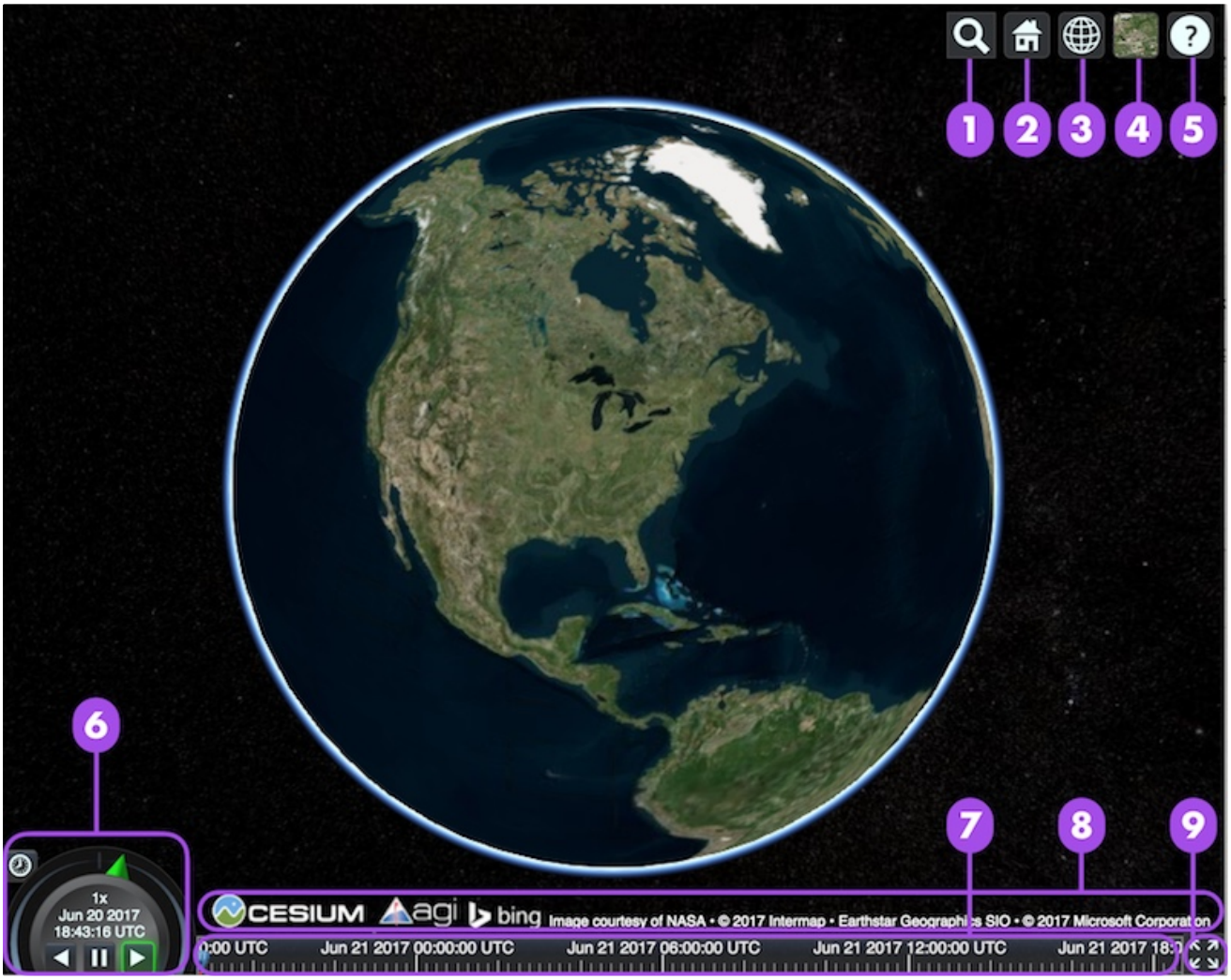
Now let's get started!

Creating the Viewer

The basis of any Cesium application is the Viewer (http://cesiumjs.org/Cesium/Build/Documentation/Viewer.html), an interactive 3D globe with lots of functionality right out of the box. Add the viewer to the 'cesiumContainer' `div` by uncommenting the first line.

```
var viewer = new Cesium.Viewer('cesiumContainer');
```

There's a lot included in that one line! You should see a basic globe like this:



By default, the scene handles mouse and touch input. Try exploring the globe using the default camera controls:

- Left click and drag - Pans the camera over the surface of the globe.
- Right click and drag - Zooms the camera in and out.
- Middle wheel scrolling - Also zooms the camera in and out.
- Middle click and drag - Rotates the camera around the point on the surface of the globe.

In addition to the globe itself, the Viewer comes with some helpful widgets by default, labelled in the above image.

1. Geocoder (http://cesiumjs.org/Cesium/Build/Documentation/Geocoder.html) : A location search tool that flies the camera to queried location. Uses Bing Maps data by default.
2. Home Button (http://cesiumjs.org/Cesium/Build/Documentation/HomeButton.html) : Flies the viewer back to a default view.
3. Scene Mode Picker (http://cesiumjs.org/Cesium/Build/Documentation/SceneModePicker.html) : Switches between 3D, 2D and Columbus View (CV) modes.
4. Base Layer Picker (http://cesiumjs.org/Cesium/Build/Documentation/BaseLayerPicker.html) : Chooses the imagery and terrain to display on the globe.
5. Navigation Help Button (http://cesiumjs.org/Cesium/Build/Documentation/NavigationHelpButton.html) : Displays the default camera controls.
6. Animation (http://cesiumjs.org/Cesium/Build/Documentation/Animation.html) : Controls the play speed for view animation.
7. Timeline (http://cesiumjs.org/Cesium/Build/Documentation/Timeline.html) : Indicates current time and allows users to jump to a specific time using the scrubber.

8. Credits Display (http://cesiumjs.org/Cesium/Build/Documentation/CreditDisplay.html) : Displays data attributions. Almost always required!
9. Fullscreen Button (http://cesiumjs.org/Cesium/Build/Documentation/FullscreenButton.html) : Makes the Viewer fullscreen.

We can configure our viewer to include or exclude these features and more by passing in a options object as a parameter when we create it. For this application, delete that first line and configure a new viewer by uncommenting the next few lines:

```
var viewer = new Cesium.Viewer('cesiumContainer', {
    scene3DOnly: true,
    selectionIndicator: false,
    baseLayerPicker: false
});
```

This will create a viewer without selection indicators, base layer picker or scene mode picker widgets, since these will be unnecessary for our app. For the full set of Viewer options, see the Viewer documentation (http://cesiumjs.org/Cesium/Build/Documentation/Viewer.html).

Adding Imagery

The next key element of our Cesium application is imagery. This is the set of images that tile over our virtual globe at various resolutions. To provide optimal performance, Cesium only requests and renders imagery tiles that are visible in the current view and that are at a resolution (also known as zoom level) appropriate to the camera's distance from the globe's surface and the globe's maximumScreenSpaceError (https://cesiumjs.org/Cesium/Build/Documentation/Globe.html#maximumScreenSpaceError).

For a more visual understanding of how imagery works, check out the Cesium Inspector (http://cesiumjs.org/Cesium/Apps/Sandcastle/index.html?src=Cesium%20Inspector.html&label=All).

Cesium provides lots of tools for working with imagery layers, such as color adjustment and layer blending. Some code examples: * Adding basic imagery (http://cesiumjs.org/Cesium/Apps/Sandcastle/index.html?src=Imagery%20Layers.html) * Adjusting imagery colors (http://cesiumjs.org/Cesium/Apps/Sandcastle/index.html?src=Imagery%20Adjustment.html) * Manipulating and ordering imagery layers (http://cesiumjs.org/Cesium/Apps/Sandcastle/index.html?src=Imagery%20Layers%20Manipulation.html) * Splitting imagery layers (http://cesiumjs.org/Cesium/Apps/Sandcastle/index.html?src=Imagery%20Layers%20Split.html)

Cesium provides support for imagery from many different providers (http://cesiumjs.org/Cesium/Build/Documentation/index.html?classFilter=ImageryProvider) out of the box.

Supported Imagery Formats: * WMS * TMS * WMTS (with time dynamic imagery) * ArcGIS * Bing Maps * Google Earth * Mapbox * Open Street Map servers * Single tile

By default, Cesium uses Bing Maps for imagery. Be careful, different data providers have different attribution requirements – make sure you have permission to use data from a particular provider, crediting them in the credits container if applicable. The imagery packaged with the Viewer is mostly for demo purposes. In order to

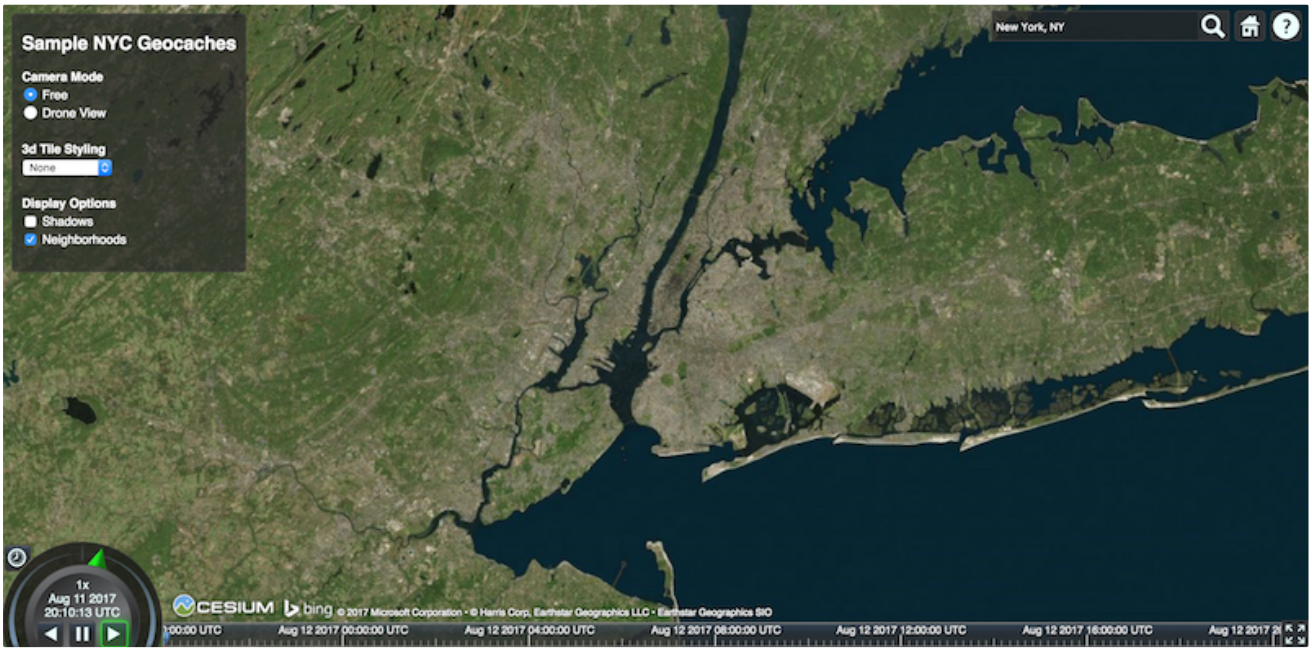
use the Bing imagery set in our application, we need to acquire a Bing key (https://msdn.microsoft.com/en-us/library/ff428642.aspx) of our own. Set the Bing key with a line like this (at the top of our application, before the viewer is created):

```
Cesium.BingMapsApi.defaultKey = 'AsarFiDvISunWhi137V7l5Bu80baB73npU98oTyjqKOb7NbrkiuBPZfDxgXTrGtQ'; //
```

Again, different imagery providers will have different requirements for usage. Now that we have permission to use this imagery set, we can actually add the imagery layer. First, we create an ImageryProvider (http://cesiumjs.org/Cesium/Build/Documentation/ImageryProvider.html), passing in a data url and a few configuration options, then we add the ImageryProvider to viewer.imageryLayers (http://cesiumjs.org/Cesium/Build/Documentation/Viewer.html#imageryLayers).

```
// Add Bing imagery
viewer.imageryLayers.addImageryProvider(new Cesium.BingMapsImageryProvider({
    url : 'https://dev.virtualearth.net',
    mapStyle: Cesium.BingMapsStyle.AERIAL // Can also use Cesium.BingMapsStyle.ROADS
}));
```

With the above code additions, our application should look like this when you zoom in:



This is actually the same as the default imagery styling, but feel free to play with the `BingMapsStyle` to see the differences.

For more information on Imagery, see our Imagery Layers Tutorial (http://cesiumjs.org/tutorials/Imagery-Layers-Tutorial/).

Adding Terrain

Cesium supports streaming and visualizing global high-resolution terrain and water effects for oceans, lakes, and rivers. Mountain peaks, valleys, and other terrain features really show the benefit of a 3D globe compared to a 2D map. Like imagery, the Cesium engine will stream terrain data from a server, only requesting and rendering tiles as needed based on the current camera position.

Here are some demos of terrain datasets and configuration options: * ArcticDEM (http://cesiumjs.org/Cesium/Apps/Sandcastle/index.html?src=ArcticDEM.html&label=Showcases) : High resolution arctic terrain * PATerrain (http://cesiumjs.org/Cesium/Apps/Sandcastle/index.html?src=PATerrain.html&label=Showcases) : High resolution Pennsylvania terrain * Terrain display options (http://cesiumjs.org/Cesium/Apps/Sandcastle/index.html?src=Terrain.html&label=Showcases) : Some terrain configuration options and formats * Terrain exaggeration (http://cesiumjs.org/Cesium/Apps/Sandcastle/index.html?src=Terrain%20Exaggeration.html&label=Showcases) : Making terrain elevation differences more dramatic

Supported Terrain Formats: - Our own quantized-mesh (https://github.com/AnalyticalGraphicsInc/quantized-mesh) format - Heightmap - Google Earth Enterprise

In order to add terrain data, we create a CesiumTerrainProvider (http://cesiumjs.org/Cesium/Build/Documentation/CesiumTerrainProvider.html), specifying a data url and a few configuration options, then adding the provider as viewer.terrainProvider (http://cesiumjs.org/Cesium/Build/Documentation/Viewer.html?classFilter=viewer#terrainProvider).

```
// Load STK World Terrain
viewer.terrainProvider = new Cesium.CesiumTerrainProvider({
    url : 'https://assets.agi.com/stk-terrain/world',
    requestWaterMask : true, // required for water effects
    requestVertexNormals : true // required for terrain lighting
});
```

`requestWaterMask` and `requestVertexNormals` are configuration options which tell Cesium to request extra data for water and lighting effects. By default these are set to false.

Finally, now that we have have terrain, we need just one more line to make sure objects behind the terrain are correctly occluded. Only the front-most objects will be visible.

```
// Enable depth testing so things behind the terrain disappear.
viewer.scene.globe.depthTestAgainstTerrain = true;
```

We now have terrain and animated water. New York is pretty flat, so feel free to explore in order to see the new terrain in action. For a particularly obvious example, you can navigate to a more rugged area like the Grand Canyon or San Francisco.

For more information on terrain, see the Terrain Tutorial (https://cesiumjs.org/tutorials/Terrain-Tutorial/).

Configuring the Scene

Now just a little more setup to start our viewer in the right location and time. This involves interacting with viewer.scene (http://cesiumjs.org/Cesium/Build/Documentation/Scene.html), a container for all the graphical elements in our viewer.

To start, we can configure our scene to optionally enable lighting based on the sun's position with this line.

```
// Enable lighting based on sun/moon positions
viewer.scene.globe.enableLighting = true;
```

This will make the lighting in our scene change with the time of day, such that you can see part of the globe go dark from space when the sun is out of view.

Next, before we get started with setting up our initial view, let's go over a few basic Cesium types:

- Cartesian3 (http://cesiumjs.org/Cesium/Build/Documentation/Cartesian3.html) : a 3D Cartesian point – when used as a position it is relative to the center of the globe in meters using the Earth fixed-frame (ECR)
- Cartographic (http://cesiumjs.org/Cesium/Build/Documentation/Cartographic.html) : a position defined by latitude/longitude (in radians) and height off the globe's surface
- Heading Pitch Roll (http://cesiumjs.org/Cesium/Build/Documentation/HeadingPitchRoll.html) : A rotation (in radians) about the local axes in the East-North-Up frame. Heading is the rotation about the negative z axis. Pitch is the rotation about the negative y axis. Roll is the rotation about the positive x axis.
- Quaternion (http://cesiumjs.org/Cesium/Build/Documentation/Quaternion.html) : A 3D rotation represented as 4D coordinates.

These are the basic types necessary to position and orient Cesium objects within a scene and have a number of helpful conversion methods.

Now let's position our scene in NYC, where our data is located.

Camera Control

The Camera (http://cesiumjs.org/Cesium/Build/Documentation/Camera.html) is a property of viewer.scene (http://cesiumjs.org/Cesium/Build/Documentation/Scene.html) and controls what is currently visible. We can control the camera by setting its position and orientation directly, or by using the Cesium camera API, which is designed to flexibly specify camera position and orientation over time.

Some of the most commonly used methods are:

- Camera.setView(options) (http://cesiumjs.org/Cesium/Build/Documentation/Camera.html#setView) : set camera at specific position and orientation immediately
- Camera.zoomIn(amount) (http://cesiumjs.org/Cesium/Build/Documentation/Camera.html#zoomIn) : move camera forward along the view vector
- Camera.zoomOut(amount) (http://cesiumjs.org/Cesium/Build/Documentation/Camera.html#zoomOut) : move camera backwards along the view vector
- Camera.flyTo(options) (http://cesiumjs.org/Cesium/Build/Documentation/Camera.html#flyTo) : animates movement from current position to a new position
- Camera.lookAt(target, offset) (http://cesiumjs.org/Cesium/Build/Documentation/Camera.html#lookAt) : orient and position the camera to look at target point with given offset
- Camera.move(direction, amount) (http://cesiumjs.org/Cesium/Build/Documentation/Camera.html#move) : move the camera in any direction
- Camera.rotate(axis, angle) (http://cesiumjs.org/Cesium/Build/Documentation/Camera.html#rotate) : rotate the camera about any axis

To get an idea of what the API can do, check out these camera demos: * Camera API Demo (http://cesiumjs.org/Cesium/Apps/Sandcastle/index.html?src=Camera.html&label=Tutorials) * Custom Camera Controls Demo (http://cesiumjs.org/Cesium/Apps/Sandcastle/index.html?src=Camera%20Tutorial.html&label=Tutorials)

Now let's try one of these methods by moving the camera to New York. Set the initial view with camera.setView() (http://cesiumjs.org/Cesium/Build/Documentation/Camera.html), using a Cartesian3 and a HeadingPitchRoll for position and orientation :

```
// Create an initial camera view
var initialPosition = new Cesium.Cartesian3.fromDegrees(-73.998114468289017509, 40.674512895646692812,
var initialOrientation = new Cesium.HeadingPitchRoll.fromDegrees(7.1077496389876024807, -31.9872230915
var homeCameraView = {
    destination : initialPosition,
    orientation : {
        heading : initialOrientation.heading,
        pitch : initialOrientation.pitch,
        roll : initialOrientation.roll
    }
};
// Set the initial view
viewer.scene.camera.setView(homeCameraView);
```

The camera is now positioned and oriented to look down at Manhattan, and our view parameters are saved in a object that we can pass to other camera methods.

In fact, we can use this same view to update the effect of pressing the home button. Rather than having it return us to the default view of the globe from a distance, we can override the button to bring us to that initial view of Manhattan. We can adjust the animation by adding a few more options, then add an event listener that cancels the default flight, and calls `flyTo()` our new home view:

```
// Add some camera flight animation options
homeCameraView.duration = 2.0;
homeCameraView.maximumHeight = 2000;
homeCameraView.pitchAdjustHeight = 2000;
homeCameraView.endTransform = Cesium.Matrix4.IDENTITY;
// Override the default home button
viewer.homeButton.viewModel.command.beforeExecute.addEventListener(function (e) {
    e.cancel = true;
    viewer.scene.camera.flyTo(homeCameraView);
});
```

For more on basic camera controls, check out our Camera Tutorial (http://cesiumjs.org/tutorials/Camera-Tutorial/).

Clock Control

Next, we can configure the viewer Clock (http://cesiumjs.org/Cesium/Build/Documentation/Clock.html?classFilter=clock) and Timeline (http://cesiumjs.org/Cesium/Build/Documentation/Timeline.html?classFilter=timeline) to control the passage of time within our scene.

Here's the clock API in action (http://cesiumjs.org/Cesium/Apps/Sandcastle/index.html?src=Clock.html&label=Showcases).

When working with specific times, Cesium uses the JulianDate (http://cesiumjs.org/Cesium/Build/Documentation/JulianDate.html) type, which stores the number of days since noon on January 1, -4712 (4713 BC). For increased precision, this class stores the whole number part of the date and the seconds part of the date in separate components, and in order to be safe for arithmetic and represent leap seconds, the date is always stored in the International Atomic Time standard.

Here's an example of how we can set up our scene time options:

```
// Set up clock and timeline.
viewer.clock.shouldAnimate = true; // default
viewer.clock.startTime = Cesium.JulianDate.fromIso8601("2017-07-11T16:00:00Z");
viewer.clock.stopTime = Cesium.JulianDate.fromIso8601("2017-07-11T16:20:00Z");
viewer.clock.currentTime = Cesium.JulianDate.fromIso8601("2017-07-11T16:00:00Z");
viewer.clock.multiplier = 2; // sets a speedup
viewer.clock.clockStep = Cesium.ClockStep.SYSTEM_CLOCK_MULTIPLIER; // tick computation mode
viewer.clock.clockRange = Cesium.ClockRange.LOOP_STOP; // loop at the end
viewer.timeline.zoomTo(viewer.clock.startTime, viewer.clock.stopTime); // set visible range
```

This will set the rate of the scene animation, the start and end times and tell the clock to loop when it hits the end time. It also sets the timeline to the appropriate time range. Check out this clock example code (http://cesiumjs.org/Cesium/Apps/Sandcastle/index.html?src=Clock.html&label=All) to experiment with clock settings.

That's it for our initial scene configuration! Now when you run your application, you should see the following:

Loading and Styling Entities

Now that we've set the stage for our application with viewer configuration, imagery and terrain, we can go ahead and add the main focus of our application, sample geocache data.

For easy visualization, Cesium supports popular vector formats GeoJson and KML, as well as our own vector format, CZML.

Regardless of the initial format, all spatial data in Cesium are represented using the Entity API, a geometry library that provides flexible visualization in a format that is efficient for Cesium to render. A Cesium Entity (http://cesiumjs.org/Cesium/Build/Documentation/Entity.html) is a data object that can be paired with a styled graphical representation and positioned in space and time. The sandcastle gallery provides many examples of simple entities (http://cesiumjs.org/Cesium/Apps/Sandcastle/index.html?src=Box.html&label=Geometries). To get up to speed on the basics of the Entity API, take a break from this application and read Visualizing Spatial Data (http://cesiumjs.org/tutorials/Visualizing-Spatial-Data/) first.

Here are examples of different entity types: * Boxes (http://cesiumjs.org/Cesium/Apps/Sandcastle/index.html?src=Box.html&label=Geometries) * Polygons (http://cesiumjs.org/Cesium/Apps/Sandcastle/index.html?src=Polygon.html&label=Geometries) * Polylines (http://cesiumjs.org/Cesium/Apps/Sandcastle/index.html?src=Polyline.html&label=Geometries) * Billboards (http://cesiumjs.org/Cesium/Apps/Sandcastle/index.html?src=Billboards.html&label=Beginner)

Once you've got a handle on what an Entity looks like, loading datasets with Cesium will be easy to understand. To read in a data file, create a DataSource (http://cesiumjs.org/Cesium/Build/Documentation/DataSource.html) appropriate to your data's format, which will parse the data file hosted at a specified url and create an EntityCollection (http://cesiumjs.org/Cesium/Build/Documentation/EntityCollection.html) containing an Entity for each geospatial object in the dataset. DataSource just defines an interface – the exact kind of data source you'll need will depend on the data format. For example, a KML uses a KmlDataSource (http://cesiumjs.org/Cesium/Build/Documentation/KmlDataSource.html). Here's what it looks like:

```
var kmlOptions = {
    camera : viewer.scene.camera,
    canvas : viewer.scene.canvas,
    clampToGround : true
};
// Load geocache points of interest from a KML file
// Data from : http://catalog.opendata.city/dataset/pediacities-nyc-neighborhoods/resource/91778048-3c
var geocachePromise = Cesium.KmlDataSource.load('./Source/SampleData/sampleGeocacheLocations.kml', kml
```

This code reads our sample geocache points from a KML file by calling KmlDataSource.load (http://cesiumjs.org/Cesium/Build/Documentation/KmlDataSource.html?classFilter=data#.load) with a few options. For a KmlDataSource, the camera and canvas options are required (necessary for working with network links). The clampToGround option enables ground clamping (http://cesiumjs.org/Cesium/Apps/Sandcastle/index.html?src=Ground%20Clamping.html&label=Showcases), a popular display option that moves entities to conform to terrain rather than just the ellipsoid surface.

Since this load is handled asynchronously, this returns a Promise (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise) to a KmlDataSource (http://cesiumjs.org/Cesium/Build/Documentation/KmlDataSource.html) which will hold all our newly created entities.

If you're not familiar with the `Promise` API for working with asynchronous loads, the "asynchronous" here basically means you should do what you need to do with the data in a callback provided to `.then`. In order to actually add this collection of entities to the scene, we must wait until the promise resolves, then add the KmlDataSource to viewer.datasources (http://cesiumjs.org/Cesium/Build/Documentation/Viewer.html?classFilter=viewer#dataSources). Uncomment the following lines:

```
// Add geocache billboard entities to scene and style them
geocachePromise.then(function(dataSource) {
    // Add the new data as entities to the viewer
    viewer.dataSources.add(dataSource);
});
```

These newly-created Entities come with useful functionality by default. Try clicking (display infobox) and double-clicking (zoom in) on a point or neighborhood. Next we'll work on adding custom-styling to improve the look of our app.

For KML and CZML files, declarative styling can be built into the file. However, for this application, let's practice manually styling our entities. To do this, we'll take a similar approach to this styling example (http://cesiumjs.org/Cesium/Apps/Sandcastle/index.html?src=GeoJSON%20and%20TopoJSON.html&label=Showcases) by waiting for our datasources to load, then iterating though all the entities in a datasource collection and modifying and adding attributes. Our geocache point markers are created as Billboards (http://cesiumjs.org/Cesium/Build/Documentation/BillboardGraphics.html?classFilter=billboard) by default, so to modify the appearance of any of those entities, we do this:

```
// Add geocache billboard entities to scene and style them
geocachePromise.then(function(dataSource) {
    // Add the new data as entities to the viewer
    viewer.dataSources.add(dataSource);

    // Get the array of entities
    var geocacheEntities = dataSource.entities.values;

    for (var i = 0; i < geocacheEntities.length; i++) {
        var entity = geocacheEntities[i];
        if (Cesium.defined(entity.billboard)) {
            // Entity styling code here
        }
    }
});
```

We can improve the appearance of our markers by adjusting their anchor points, removing the labels to reduce clutter and setting the displayDistanceCondition (http://cesiumjs.org/Cesium/Build/Documentation/DistanceDisplayCondition.html) so that only points within a set distance from the camera are visible.

```
// Add geocache billboard entities to scene and style them

        if (Cesium.defined(entity.billboard)) {
            // Adjust the vertical origin so pins sit on terrain
            entity.billboard.verticalOrigin = Cesium.VerticalOrigin.BOTTOM;
            // Disable the labels to reduce clutter
            entity.label = undefined;
            // Add distance display condition
            entity.billboard.distanceDisplayCondition = new Cesium.DistanceDisplayCondition(10.0, 2000
        }
```

For more help with distanceDisplayCondition, see the sandcastle example (http://cesiumjs.org/Cesium/Apps/Sandcastle/index.html?src=Distance%20Display%20Conditions.html&label=Showcases).

Next, let's improve the infobox (http://cesiumjs.org/Cesium/Build/Documentation/InfoBox.html) for each of our geocache entities. The title of the info box is the entity name, and the contents are the entity description, displayed as HTML.

You'll notice that the default descriptions aren't very helpful. Since we're displaying geocache locations, let's update them to display the latitude and longitude of our points.

First, we'll convert the entity's position into a Cartographic, then read the latitude and longitude from the Cartographic and add it to the description in an HTML table.

On click, our geocache point entities will now display a nicely-formatted infobox with just the data we need.

```
// Add geocache billboard entities to scene and style them
        if (Cesium.defined(entity.billboard)) {
            // Adjust the vertical origin so pins sit on terrain
            entity.billboard.verticalOrigin = Cesium.VerticalOrigin.BOTTOM;
            // Disable the labels to reduce clutter
            entity.label = undefined;
            // Add distance display condition
            entity.billboard.distanceDisplayCondition = new Cesium.DistanceDisplayCondition(10.0, 2000
            // Compute latitude and longitude in degrees
            var cartographicPosition = Cesium.Cartographic.fromCartesian(entity.position.getValue(Cesi
            var latitude = Cesium.Math.toDegrees(cartographicPosition.latitude);
            var longitude = Cesium.Math.toDegrees(cartographicPosition.longitude);
            // Modify description
            var description = '<table class="cesium-infoBox-defaultTable cesium-infoBox-defaultTable-l
            description += '<tr><th>' + "Latitude" + '</th><td>' + latitude + '</td></tr>';
            description += '<tr><th>' + "Longitude" + '</th><td>' + longitude + '</td></tr>';
            description += '</tbody></table>';
            entity.description = description;
        }
```

Our geocache markers now should look like this:

For our geocaching application, it might also be helpful to visualize what neighborhood a particular point will fall into. Let's try loading a GeoJson file containing polygons for each of the NYC neighborhoods. Loading a GeoJson file is ultimately very similar to the load process we just used for a KML. But in this case, we use a GeoJsonDataSource (http://cesiumjs.org/Cesium/Build/Documentation/GeoJsonDataSource.html) instead, and exclude the camera and canvas options. And like with the previous datasource, we need to add it to viewer.datasources to actually add data to the scene.

```
var geojsonOptions = {
    clampToGround : true
};
// Load neighborhood boundaries from KML file
var neighborhoodsPromise = Cesium.GeoJsonDataSource.load('./Source/SampleData/neighborhoods.geojson',

// Save an new entity collection of neighborhood data
var neighborhoods;
neighborhoodsPromise.then(function(dataSource) {
    // Add the new data as entities to the viewer
    viewer.dataSources.add(dataSource);
});
```

Since our geocache points are already looking clean, let's style the neighborhood polygons we loaded. Just like with the billboard styling we just did, we begin by iterating through the neighborhood dataSource entities once the dataSource has loaded, this time checking that each entity's polygon is defined:

```
// Save an new entity collection of neighborhood data
var neighborhoods;
neighborhoodsPromise.then(function(dataSource) {
    // Add the new data as entities to the viewer
    viewer.dataSources.add(dataSource);
    neighborhoods = dataSource.entities;

    // Get the array of entities
    var neighborhoodEntities = dataSource.entities.values;
    for (var i = 0; i < neighborhoodEntities.length; i++) {
        var entity = neighborhoodEntities[i];

        if (Cesium.defined(entity.polygon)) {
            // entity styling code here
        }
    }
});
```

Since we're displaying neighborhoods, let's rename each entity to use the neighborhood as its name. The original GeoJson file that we read in has neighborhood as a property. Cesium stores the GeoJson properties under entity.properties (http://cesiumjs.org/Cesium/Build/Documentation/Entity.html?classFilter=entity#properties), so we can set the neighborhood names like this:

```
// entity styling code here

            // Use geojson neighborhood value as entity name
            entity.name = entity.properties.neighborhood;
```

Rather than leaving all our neighborhoods the same color, we can assign each polygon a new ColorMaterialProperty (http://cesiumjs.org/Cesium/Build/Documentation/ColorMaterialProperty.html?classFilter=material) by setting the material to a random Color (http://cesiumjs.org/Cesium/Build/Documentation/Color.html?classFilter=color).

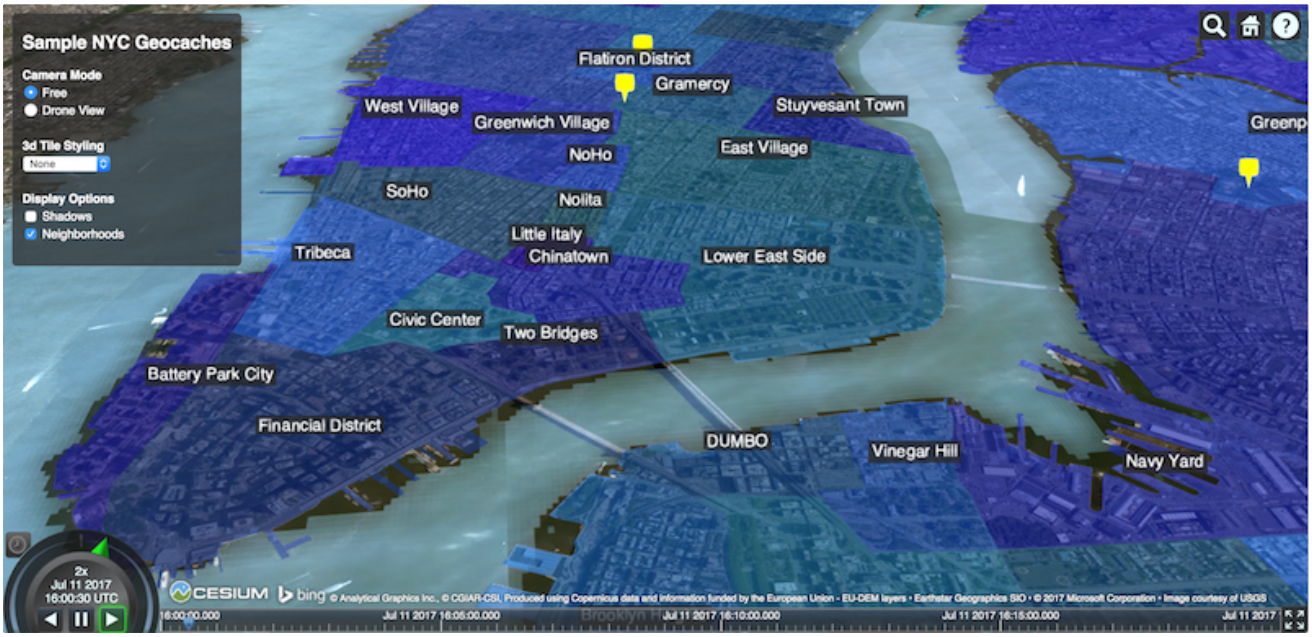
```
// entity styling code here

            // Set the polygon material to a random, translucent color.
            entity.polygon.material = Cesium.Color.fromRandom({
                red : 0.1,
                maximumGreen : 0.5,
                minimumBlue : 0.5,
                alpha : 0.6
            });
```

Finally, let's generate a Label (http://cesiumjs.org/Cesium/Build/Documentation/LabelGraphics.html?classFilter=label) for each entity with a few basic styling options. To keep things neat, we can use disableDepthTestDistance (http://cesiumjs.org/Cesium/Build/Documentation/LabelGraphics.html?classFilter=label#disableDepthTestDistance) to have Cesium always render the labels in front of whatever 3D object might occlude it.

However, note that a Label is always positioned at `Entity.position`. A `Polygon` is created with an undefined position since it has a list of constituent point positions. We can generate a polygon position by taking the center of the polygon positions:

```
// entity styling code here

            // Generate Polygon position
            var polyPositions = entity.polygon.hierarchy.getValue(Cesium.JulianDate.now()).positions;
            var polyCenter = Cesium.BoundingSphere.fromPoints(polyPositions).center;
            polyCenter = Cesium.Ellipsoid.WGS84.scaleToGeodeticSurface(polyCenter);
            entity.position = polyCenter;
            // Generate labels
            entity.label = {
                text : entity.name,
                showBackground : true,
                scale : 0.6,
                horizontalOrigin : Cesium.HorizontalOrigin.CENTER,
                verticalOrigin : Cesium.VerticalOrigin.BOTTOM,
                distanceDisplayCondition : new Cesium.DistanceDisplayCondition(10.0, 8000.0),
                disableDepthTestDistance : Number.POSITIVE_INFINITY
            };
```

This gives us labelled polygons that look like this:



Although our labelled polygons are now nicely styled, our entire scene has become a bit cluttered, so let's make create a way to hide the polygons. In general, we can hide entities by setting visibility with Entity.show (http://cesiumjs.org/Cesium/Build/Documentation/Entity.html?classFilter=entity#show). However, this only sets visibility for a single entity, and we'd like to hide or show all the neighborhood entities at once.

We can do this by adding all our neighborhood entities to a parent entity, as shown in this example (http://cesiumjs.org/Cesium/Apps/Sandcastle/index.html?src=Show%20or%20Hide%20Entities.html&label=Showcases) or by simply using the `show` property of Entity Collection (http://cesiumjs.org/Cesium/Build/Documentation/EntityCollection.html). We can then set visibility for all the child entities at once by changing the value of `neighborhoods.show`:

```
neighborhoods.show = false;
```

Finally, let's add a high-tech view of our nyc geocaches by adding a drone flight over the city.

Since a flight path is just a series of positions over time, we can add this data from a CZML file (https://github.com/AnalyticalGraphicsInc/czml-writer/wiki/CZML-Guide). CZML is a format for describing a time-dynamic graphical scene, primarily for display in a web browser running Cesium. It describes lines, points, billboards, models, and other graphical primitives, and specifies how they change with time. CZML is to Cesium what KML is to Google Earth, a standard format that allows for most Cesium features to be used via a declarative styling language (in this case a JSON schema)

Our CZML file defines an entity (visualized by default as a point) with its position defined as a series of positions at different time points. It's a nice example of the Property system that the Entity API uses to store dynamic values.

- Property Types Demo (http://cesiumjs.org/Cesium/Apps/Sandcastle/index.html?src=Hello%20World.html&label=Showcases&gist=c1b1ef84af7dd0b40b3102ffed164a3d)

```
// Load a drone flight path from a CZML file
var dronePromise = Cesium.CzmlDataSource.load('./Source/SampleData/SampleFlight.czml');

dronePromise.then(function(dataSource) {
    viewer.dataSources.add(dataSource);
});
```

In this case, the CZML file has Cesium display the drone flight using a PathGraphics (http://cesiumjs.org/Cesium/Build/Documentation/PathGraphics.html), a property of the entity which displays its position over time. A path joins discrete samples into a continuous line to be visualized using interpolation.

We now have all the data we need to finish our nyc geocache application, all loaded and styled as entities within our scene.

Finally, let's improve the look of our drone flight. First of all, rather then settling for a simple point, we can load a 3D model to represent our drone and attach it to the entity.

- 3D Model Demo (http://cesiumjs.org/Cesium/Apps/Sandcastle/index.html?src=3D%20Models.html&label=Showcases)
- 3D Model Coloring Demo (http://cesiumjs.org/Cesium/Apps/Sandcastle/index.html?src=3D%20Models%20Coloring.html&label=Showcases)

Cesium supports loading 3D models based on glTF (GL Transmission Format), a royalty-free specification for the efficient transmission and loading of 3D scenes and models by applications which minimizes file size and runtime processing. Don't have a glTF? We provide an online converter (http://cesiumjs.org/convertmodel.html) for converting COLLADA and OBJ files to glTF.

Let's load a drone Model (http://cesiumjs.org/Cesium/Build/Documentation/ModelGraphics.html) with nice physically-based shading and some animation:

```
var drone;
dronePromise.then(function(dataSource) {
    viewer.dataSources.add(dataSource);
    drone = dataSource.entities.values[0];
    // Attach a 3D model
    drone.model = {
        uri : './Source/SampleData/Models/CesiumDrone.gltf',
        minimumPixelSize : 128,
        maximumScale : 2000
    };
});
```

Now our model looks nice, but unlike the original point, the drone model has orientation, which looks strange when the drone doesn't turn as it moves. Fortunately, Cesium provides VelocityOrientationProperty (http://cesiumjs.org/Cesium/Build/Documentation/VelocityOrientationProperty.html) that will automatically compute an orientation based on an entity's positions sampled forward and backwards in time:

```
// Add computed orientation based on sampled positions
    drone.orientation = new Cesium.VelocityOrientationProperty(drone.position);
```

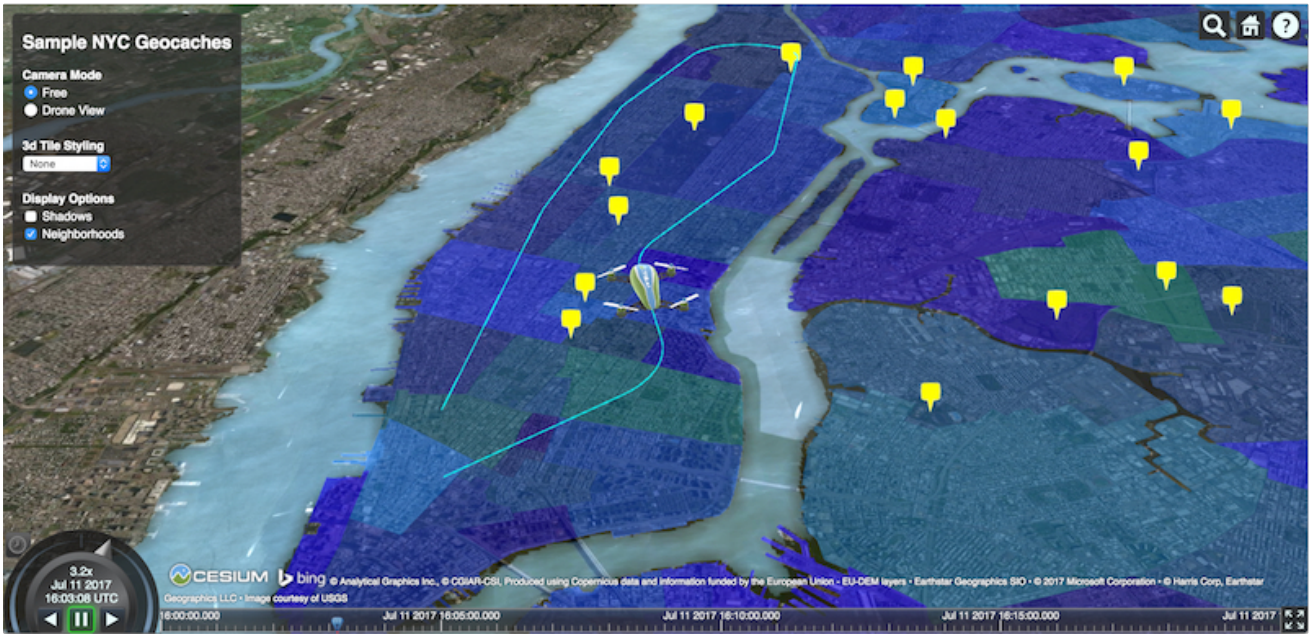
Now our drone model will turn as expected.

There's one more thing we can do to improve the look of our drone flight. It may not be obvious from a distance, but the drone's path is made of linear segments that look unnatural – this is because Cesium uses linear interpolation to construct a path from sampled points by default. However, the interpolation options can be configured.

- Interpolation Demo (http://cesiumjs.org/Cesium/Apps/Sandcastle/index.html?src=Interpolation.html&label=All)

To get a smoother looking flight path, we can change the interpolation options like this:

```
// Smooth path interpolation
    drone.position.setInterpolationOptions({
        interpolationDegree : 3,
        interpolationAlgorithm : Cesium.HermitePolynomialApproximation
    });
```



3D Tiles

Our team sometimes describes Cesium as being like a 3D game engine for real world data. However, working with real world data is much more difficult than working with typical video game assets since real data can be incredibly high-resolution, and require accurate visualization. Fortunately, Cesium in collaboration with the open source community has developed 3D Tiles (http://cesiumjs.org/2015/08/10/Introducing-3D-Tiles/), an open specification (https://github.com/AnalyticalGraphicsInc/3d-tiles) for streaming massive heterogeneous 3D geospatial datasets.

Using a technique conceptually similar to Cesium's terrain and imagery streaming, 3D Tiles make it possible to view gigantic models, including buildings datasets, CAD (or BIM) models, point clouds, and photogrammetry models, which would otherwise be impossible to view interactively.

- The 3D Tiles Inspector (http://cesiumjs.org/Cesium/Apps/Sandcastle/index.html?src=3D%20Tiles%20Inspector.html&label=3D%20Tiles) is a debugging tool that offers a glimpse under the hood.

Here are some 3D Tiles demos showcasing different formats: * Photogrammetry Model (http://cesiumjs.org/Cesium/Apps/Sandcastle/index.html?src=3D%20Tiles%20Photogrammetry.html&label=3D%20Tiles) * Building Information Model (http://cesiumjs.org/Cesium/Apps/Sandcastle/index.html?src=3D%20Tiles%20BIM.html&label=3D%20Tiles) * Point Cloud Model (http://cesiumjs.org/Cesium/Apps/Sandcastle/index.html?src=3D%20Tiles%20Point%20Cloud.html&label=3D%20Tiles) * All Formats (http://cesiumjs.org/Cesium/Apps/Sandcastle/index.html?src=3D%20Tiles%20Formats.html&label=3D%20Tiles)

In our application, we'll use a Cesium3DTileset (http://cesiumjs.org/Cesium/Build/Documentation/Cesium3DTileset.html) to add realism to our visualization by displaying full 3D models of all the buildings in New York! Adding a tileset is easy:

```
// Load the NYC buildings tileset
var city = viewer.scene.primitives.add(new Cesium.Cesium3DTileset({
    url: 'https://beta.cesium.com/api/assets/1461?access_token=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.ey
    maximumScreenSpaceError: 16 // default value
}));
```

Like adding an ordinary 3D model, we specify a url from which to retrive our data, then add the object to the scene. In this case, we add the tileset to scene.primitives, not scene.entities because 3D Tiles is not yet part of the Entity API. The maximumScreenSpaceError specifies how much how much detail Cesium will render for a given view, so the lower the number, the more detailed the visuals. Highly detailed visuals of course come with a performance cost, so wbe aware when changing this setting.

You may notice that the buildings are not correctly positioned at ground level. This is a common problem with 3D tilesets, and fortunately it's easy to fix. We can adjust the position of the tileset by modifying its modelMatrix (http://cesiumjs.org/Cesium/Build/Documentation/Cesium3DTileset.html?classFilter=3dtil#modelMatrix).

We can find the model's current offset from the ground by converting the tileset's bounding sphere into a Cartographic (http://cesiumjs.org/Cesium/Build/Documentation/Cartographic.html?classFilter=cartographic), then adding the desired offset and resetting the modelMatrix:

```
// Adjust the tileset height so its not floating above terrain
var heightOffset = -32;
city.readyPromise.then(function(tileset) {
    // Position tileset
    var boundingSphere = tileset.boundingSphere;
    var cartographic = Cesium.Cartographic.fromCartesian(boundingSphere.center);
    var surface = Cesium.Cartesian3.fromRadians(cartographic.longitude, cartographic.latitude, 0.0);
    var offset = Cesium.Cartesian3.fromRadians(cartographic.longitude, cartographic.latitude, heightOf
    var translation = Cesium.Cartesian3.subtract(offset, surface, new Cesium.Cartesian3());
    tileset.modelMatrix = Cesium.Matrix4.fromTranslation(translation);
});
```

We now have over 1.1 million building models streaming into our scene!

3D Tiles also allows us to style parts of our tileset using the 3D Tiles styling language (https://github.com/AnalyticalGraphicsInc/3d-tiles/tree/master/Styling). A 3D Tiles style defines expressions to evaluate color (RGB and translucency) and show properties for a Cesium3DTileFeature (http://cesiumjs.org/Cesium/Build/Documentation/Cesium3DTileFeature.html), a part of the tileset such as an individual building in a city. Styling is often based on the feature's properties stored in the tile's batch table. A feature property can be anything like height, name, coordinates, construction date, etc. but is built into the tileset asset. Styles are defined with JSON and expressions written in a small subset of JavaScript augmented for styling. Additionally the styling language provides a set of built-in functions to support common math operations.

A Cesium3DTileStyle (http://cesiumjs.org/Cesium/Build/Documentation/Cesium3DTileStyle.html?classFilter=Cesium) is defined like this:

```
var defaultStyle = new Cesium.Cesium3DTileStyle({
    color : "color('white')",
    show : true
});
```

This style simply will make all the buildings in our NYC tileset white and always visible. In order to actually set the tileset to use this style, we set `city.style` :

```
city.style = defaultStyle;
```

We can define as many styles as we'd like. Here's another, making the building transparent:

```
var transparentStyle = new Cesium.Cesium3DTileStyle({
    color : "color('white', 0.3)",
    show : true
});
```

Applying the same style to every feature in our tileset is only scratching the surface. We can also use properties specific to each feature to determine styling. Here's an example that colors buildings based on their height:

```
var heightStyle = new Cesium.Cesium3DTileStyle({
    color : {
        conditions : [
            ["${height} >= 300", "rgba(45, 0, 75, 0.5)"],
            ["${height} >= 200", "rgb(102, 71, 151)"],
            ["${height} >= 100", "rgb(170, 162, 204)"],
            ["${height} >= 50", "rgb(224, 226, 238)"],
            ["${height} >= 25", "rgb(252, 230, 200)"],
            ["${height} >= 10", "rgb(248, 176, 87)"],
            ["${height} >= 5", "rgb(198, 106, 11)"],
            ["true", "rgb(127, 59, 8)"]
        ]
    }
});
```

In order to swap between styles, we can add just a little more code to listen for HTML input:

```
var tileStyle = document.getElementById('tileStyle');
function set3DTileStyle() {
    var selectedStyle = tileStyle.options[tileStyle.selectedIndex].value;
    if (selectedStyle === 'none') {
        city.style = defaultStyle;
    } else if (selectedStyle === 'height') {
        city.style = heightStyle;
    } else if (selectedStyle === 'transparent') {
        city.style = transparentStyle;
    }
}

tileStyle.addEventListener('change', set3DTileStyle);
```

For more examples of 3D Tiles and how to use and style them, check out the 3D Tiles sandcastle demos (http://cesiumjs.org/Cesium/Apps/Sandcastle/index.html?src=Hello%20World.html&label=3D%20Tiles).

3D Tiles demos: * Formats (http://cesiumjs.org/Cesium/Apps/Sandcastle/index.html?src=3D%20Tiles%20Formats.html&label=3D%20Tiles) * Photogrammetry Model (http://cesiumjs.org/Cesium/Apps/Sandcastle/index.html?src=3D%20Tiles%20Photogrammetry.html&label=3D%20Tiles) * Feature Styling (http://cesiumjs.org/Cesium/Apps/Sandcastle/index.html?src=3D%20Tiles%20Feature%20Styling.html&label=3D%20Tiles)

If you're curious about how 3D Tilesets are generated or have some data of your own to convert, you can read more here (https://groups.google.com/d/msg/cesium-dev/xLkYIuku9hA/t_AxUBepAgAJ).

Interactivity

Finally, let's add some mouse interactivity. To improve the visibility of our geocache markers, we can change their styling when a user hovers over a marker to highlight it.

To achieve this, we'll use picking, a Cesium feature that returns data from the 3D scene given a pixel position on the viewer canvas.

There are several different types of picking. * Scene.pick (http://cesiumjs.org/Cesium/Build/Documentation/Scene.html#pick) : returns an object containing the primitive at the given window position. * Scene.drillPick (http://cesiumjs.org/Cesium/Build/Documentation/Scene.html#drillPick) : returns a list of objects containing all the primitives at the given window position. * Globe.pick (http://cesiumjs.org/Cesium/Build/Documentation/Globe.html?classFilter=globe#pick) : returns the intersection point of a given ray with the terrain.

Here are some examples of picking in action: * Picking Demo (http://cesiumjs.org/Cesium/Apps/Sandcastle/index.html?src=Picking.html&label=Showcases) * 3D Tiles Feature Picking Demo (http://cesiumjs.org/Cesium/Apps/Sandcastle/index.html?src=3D%20Tiles%20Feature%20Picking.html&label=3D%20Tiles)

Since we want our highlight effect to trigger on hover, first we'll need to create a mouse action handler. For this we'll use a ScreenSpaceEventHandler (http://cesiumjs.org/Cesium/Build/Documentation/ScreenSpaceEventHandler.html), a set of handler that triggers specified functions on user input actions. ScreenSpaceEventHandler.setInputAction() (http://cesiumjs.org/Cesium/Build/Documentation/ScreenSpaceEventHandler.html#setInputAction) listens for a type of user action – a ScreenSpaceEventType (http://cesiumjs.org/Cesium/Build/Documentation/ScreenSpaceEventType.html), and runs a specific function, passing in the user action as a parameter. Here, we'll pass it a function that take movement as input:

```
var handler = new Cesium.ScreenSpaceEventHandler(viewer.scene.canvas);
handler.setInputAction(function(movement) {}, Cesium.ScreenSpaceEventType.MOUSE_MOVE);
```

Next let's actually write our highlighting function. The handler will pass in a mouse movement from which we can extract a window position to use with `pick()` . If the pick returns a billboard object, we know that we're hovering over a marker. Then, using what we learned about `Entity` styling, we can apply a highlight style.

```
// If the mouse is over a point of interest, change the entity billboard scale and color
handler.setInputAction(function(movement) {
    var pickedPrimitive = viewer.scene.pick(movement.endPosition);
    var pickedEntity = (Cesium.defined(pickedPrimitive)) ? pickedPrimitive.id : undefined;
    // Highlight the currently picked entity
    if (Cesium.defined(pickedEntity) && Cesium.defined(pickedEntity.billboard)) {
        pickedEntity.billboard.scale = 2.0;
        pickedEntity.billboard.color = Cesium.Color.ORANGERED;
    }
}, Cesium.ScreenSpaceEventType.MOUSE_MOVE);
```

This successfully triggers the highlight style change for markers. However, you'll notice that the markers stay highlighted when we move the cursor away. We can fix that by keeping track of the last marker that was highlighted, and restoring the original styling.

Here's the full function, with the marker highlighting and unhighlighting working:

```
// If the mouse is over a point of interest, change the entity billboard scale and color
var previousPickedEntity = undefined;
handler.setInputAction(function(movement) {
    var pickedPrimitive = viewer.scene.pick(movement.endPosition);
    var pickedEntity = (Cesium.defined(pickedPrimitive)) ? pickedPrimitive.id : undefined;
    // Unhighlight the previously picked entity
    if (Cesium.defined(previousPickedEntity)) {
        previousPickedEntity.billboard.scale = 1.0;
        previousPickedEntity.billboard.color = Cesium.Color.WHITE;
    }
    // Highlight the currently picked entity
    if (Cesium.defined(pickedEntity) && Cesium.defined(pickedEntity.billboard)) {
        pickedEntity.billboard.scale = 2.0;
        pickedEntity.billboard.color = Cesium.Color.ORANGERED;
        previousPickedEntity = pickedEntity;
    }
}, Cesium.ScreenSpaceEventType.MOUSE_MOVE);
```

That's it! We've now successfully added a mouse movement handler and on hover behavior for our marker entities.

Camera Modes

To show off our drone flight, let's experiment with camera modes. We'll keep it simple with two basic camera modes that users can toggle between.

- Free Mode : default camera controls
- Drone Mode : have the camera follow the drone through its flight at a fixed distance away

No code is necessary for free mode, since it uses the default controls. As for the drone follow mode, we can position the camera looking at the drone with an offset using the viewer's built in entity tracking functionality. This sets the camera to be at a fixed offset from a specified entity, even as it moves. To track an entity, we simply set viewer.trackedEntity (http://cesiumjs.org/Cesium/Build/Documentation/Viewer.html?classFilter=viewer#trackedEntity).

To switch back to the free camera mode, we can just set `viewer.trackedEntity` back to undefined, then use `camera.flyTo()` to return to our home view.

Here's the camera mode function:

```
// Create a follow camera by tracking the drone entity
function setViewMode() {
    if (droneModeElement.checked) {
        viewer.trackedEntity = drone;
    } else {
        viewer.trackedEntity = undefined;
        viewer.scene.camera.flyTo(homeCameraView);
    }
}
```

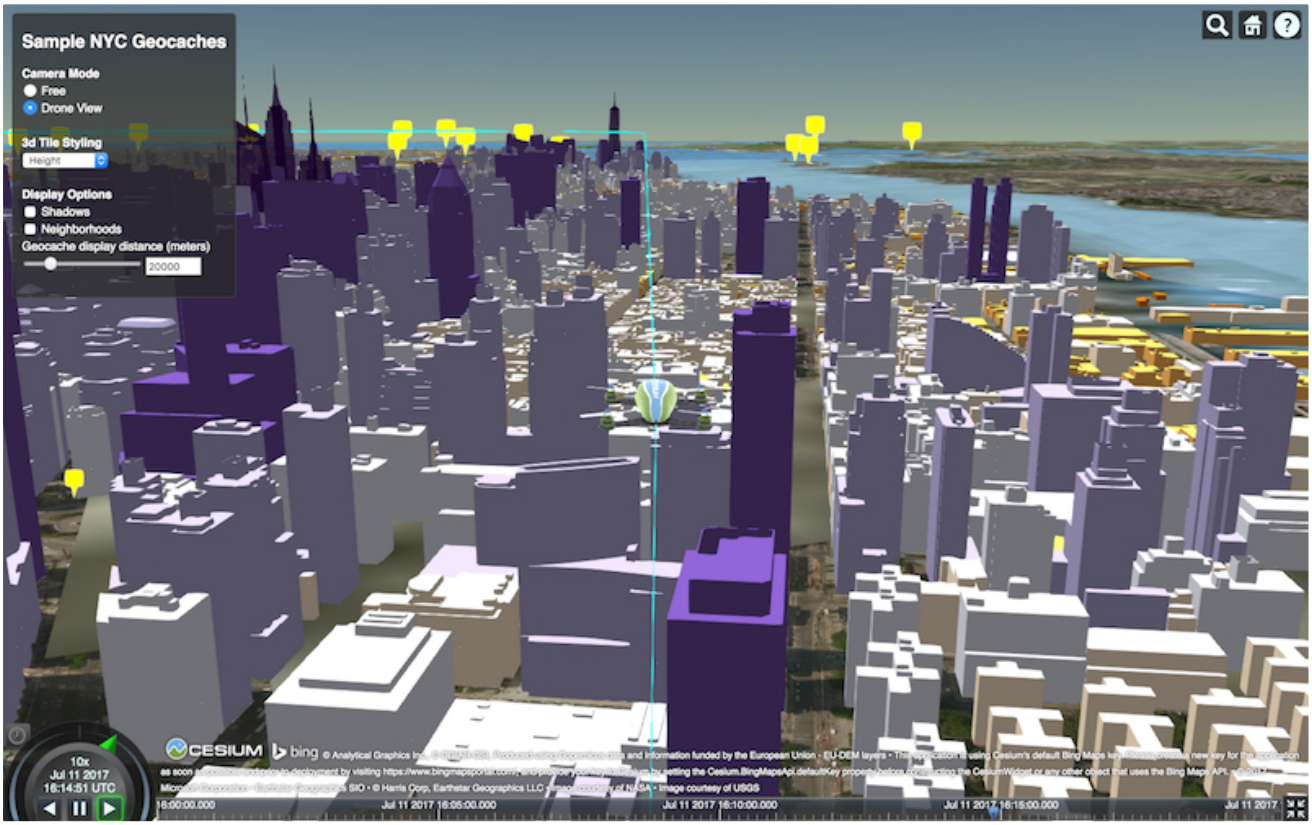
In order to attach this to the HTML input, we can attach this function to `change` events on the appropriate elements:

```
var freeModeElement = document.getElementById('freeMode');
var droneModeElement = document.getElementById('droneMode');

// Create a follow camera by tracking the drone entity
function setViewMode() {
    if (droneModeElement.checked) {
        viewer.trackedEntity = drone;
    } else {
        viewer.trackedEntity = undefined;
        viewer.scene.camera.flyTo(homeCameraView);
    }
}

freeModeElement.addEventListener('change', setCameraMode);
droneModeElement.addEventListener('change', setCameraMode);
```

Finally, entities are tracked automatically when a user double-clicks on them. We can add some handling to automatically update the UI if the user starts tracking the drone via clicking:

```
viewer.trackedEntityChanged.addEventListener(function() {
    if (viewer.trackedEntity === drone) {
        freeModeElement.checked = false;
        droneModeElement.checked = true;
    }
});
```

That's it for our two camera modes – we can now freely switch to a drone camera view that looks like this:



Extras

The rest of the code just adds a few extra visualization options. Similar to our previous interactions with the HTML elements, we can attach listener functions to toggle shadows, and the neighborhood polygon visibility.

```
var shadowsElement = document.getElementById('shadows');
var neighborhoodsElement =  document.getElementById('neighborhoods');

shadowsElement.addEventListener('change', function (e) {
    viewer.shadows = e.target.checked;
});

neighborhoodsElement.addEventListener('change', function (e) {
    neighborhoods.show = e.target.checked;
    tileStyle.value = 'transparent';
    city.style = transparentStyle;
});
```

And since the 3D Tiles may take not load instantaneously, we can also add a loading indicator that is removed only when the tileset has loaded (and hence the promise has resolved).

```
// Finally, wait for the initial city to be ready before removing the loading indicator.
var loadingIndicator = document.getElementById('loadingIndicator');
loadingIndicator.style.display = 'block';
city.readyPromise.then(function () {
    loadingIndicator.style.display = 'none';
});
```

Congratulations! That's it for our app! You've now walked through the development of a complete Cesium application from end-to-end. Feel free to explore and experiment with the code we've provided here to further your Cesium education.

What's next? Try looking at a few more tutorials (http://cesiumjs.org/tutorials.html) and sandcastle examples (http://cesiumjs.org/Cesium/Apps/Sandcastle/index.html) for more ideas about what you can do with Cesium. And remember, Cesium is more than an API – it's a community! Keep in touch about your Cesium projects on the forum (http://cesiumjs.org/forum.html).

Happy developing!

☐ **SCROLL TO TOP**

Overview

Home (http://cesiumjs.org/index.html)
Features (http://cesiumjs.org/features.html)
Downloads (http://cesiumjs.org/downloads.html)
Demos (http://cesiumjs.org/demos.html)

About

About (http://cesiumjs.org/about.html)
Blog (http://cesiumjs.org/blog.html)
Presentations (http://cesiumjs.org/publications.html)

Developers

Forum (http://cesiumjs.org/forum.html)
Tutorials (http://cesiumjs.org/tutorials.html)
Documentation (http://cesiumjs.org/refdoc.html)
Code Examples (http://cesiumjs.org/Cesium/Apps/Sandcastle/index.html)
From Google Earth (http://cesiumjs.org/for-google-earth-developers.html)

(https://github.com/AnalyticalGraphicsInc/cesium/) (https://twitter.com/cesiumjs)

Poster Session

12:30 PM → 01:30 PM Commonwealth Complex

Posters will be up from 12:30pm Wednesday to 12:30pm Friday

Poster Presentations:

- Optimum Object Analysis Of Islands Activities On South China Sea By DNB On VIIRS
  - *Ichio Asanuma, Takashi Yamaguchi, Jong-geol Park, and Keneth J. Mackin*
- Development of OGC WPS-based Climate Data Masking Interface Service Considering Climate and Forecast Conventions and Metadata
  - *Seongkyu Lee*
- An Attempt to Identify And Visualise The "Most-Used-Road-Segment" in a City
  - *Mohammed Zia*
- Reactive analyze, mapping and sharing Cloud Platform - Introduction to PINOGIO and gPocket.
  - *Hanjin Lee, Minpa Lee, Geonwoo Yu, Siwoon Jung*
- Massive computation to derive a global high-resolution stream network using GRASS-GIS
  - *Giuseppe Amatulli, Sami Domisch, Peter Raymond*
- Improving and extending a multi-modal, national US accessibility database
  - *Brendan Murphy, Andrew Owen, David Levinson*
- Estimating potential forest storage inside rural properties with kriging of the National Forest inventory and Satellite Radar Imagery
  - *Leandro Meneguelli Biondo, Renato Vinicius Oliveira Castro, Denilson Pereira Passo, Humberto Navarro de Mesquita Junior*
- Analysis of sensitivity of feature extraction and matching Algorithms using street view

Development of OGC WPS-based Climate Data Masking Interface Service Considering Climate and Forecast Conventions and Metadata

Seongkyu Lee
APEC Climate Center, 12 Centum 7-ro, Haeundae-gu, Busan 48058 Republic of Korea, +82-51-745-3967, geoslegend@apcc21.org

INTRODUCTION

Background

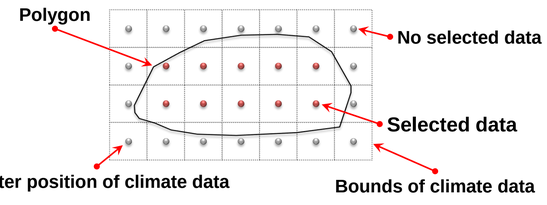
- **Recently, scientists have used scientific climate data to predict disasters arising from unusual climate events caused by climate change and global warming. Global and regional climate data stored in NetCDF format is a grid-like data that is similar to raster data type, which is one of spatial data formats. And the data attribute is created based on CF (climate and forecast) conventions and metadata. Generally, researcher use an data extraction method such as point-in-polygon algorithm and point-in-MBR (minimum bounding rectangle) algorithm when using the climate data in their region of interest (ROI). The algorithms extract data by checking the spatial relation between the center position of each grid and polygon (or MBR). The results of studies may differ depending on the data extraction method when using fine resolution data for regions such as rivers and watersheds that stretch from east to west and north to south in hydrology field. Therefore, we need an algorithm that can generate a bounding rectangle using bound attributes defined in the CF metadata instead of the center position of each grid data and extract the climate data in the bounding rectangle.**

Objectives

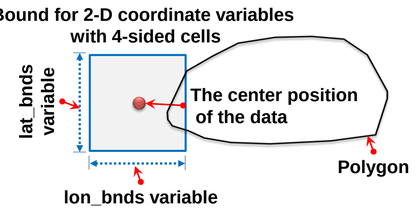
- **To develop an algorithm to mask grid data using the metadata of climate data related to spatial information and spatial data**
- **To develop OGC WPS-based climate data masking service that can be used in any environments and any program languages**

SERVICE DESIGN

- **Proposing an improved algorithm for extracting climate data with CF metadata**

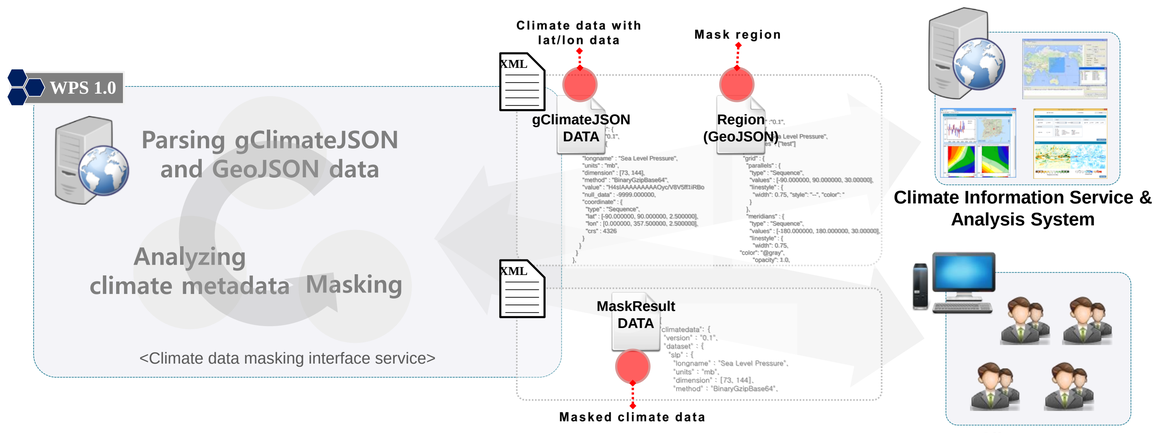


<Traditional algorithm: Point in Polygon>



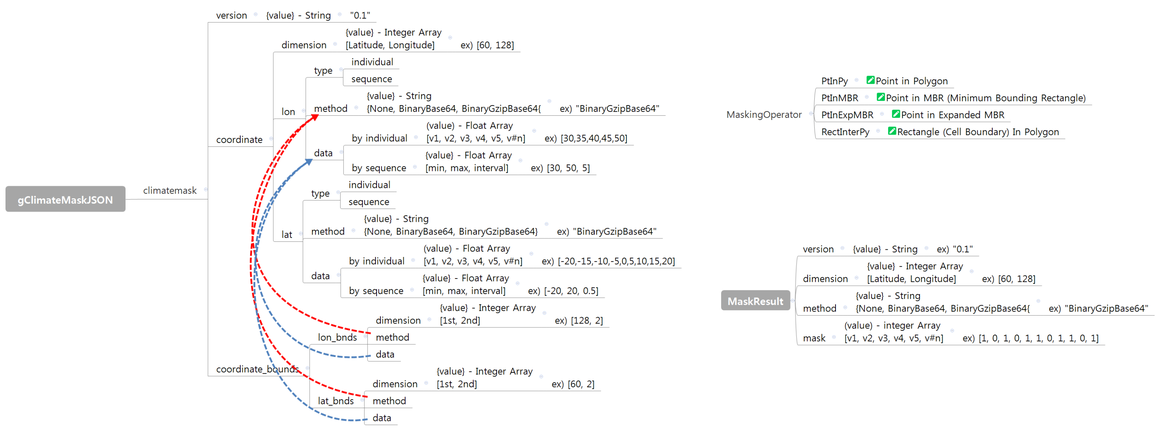
<Proposed algorithm>

- **OGC WPS-based Climate Data Masking Interface Service**



- Development Environment: CentOS 6.5 64bit, Python, PyWPS 4.0.0 and Matplotlib 1.5.0 libraries

- **Structures of input/output parameters of the climate data masking interface service**



Climate and Forecast Conventions and Metadata

- **Climate and Forecast(CF) Metadata (http://cfconventions.org)**
  - **Designed to encourage sharing and processing of climate and forecast data written in netCDF file format**
  - **Standard metadata of CMIP5 (Coupled Model Intercomparison Project Phase 5) data**

Major variables (Longitude, Latitude, Time) related to spatial and temporal information

- **Longitude**

```
double lon(lon = 128)
  lon:bounds = "lon_bnds"
  lon:units = "degrees_east"
  lon:axis = "X"
  lon:long_name = "longitude"
  lon:standard_name = "longitude"
double lon_bnds(lon = 128, bnds = 2)
```

Variable name; Data type

- **Latitude**

```
double lat(lat = 60)
  lat:bounds = "lat_bnds"
  lat:units = "degrees_north"
  lat:axis = "Y"
  lat:long_name = "latitude"
  lat:standard_name = "latitude"
double lon_bnds(lon = 128, bnds = 2)
```

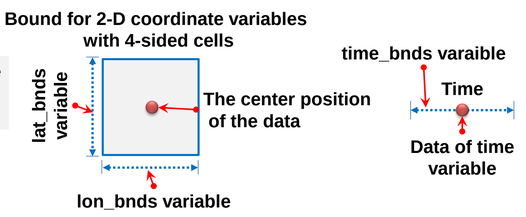
The name of the variable that contains the vertices of the cell boundaries.
Unit
The attribute defined by the NUG (NetCDF User's Guide) to contain a long descriptive name.
Used to identify physical quantities Defined in CF Metadata Conventions document

- **Longitude**

```
double time(time = 120)
  time:bounds = "time_bnds"
  time:units = "days since 0001-01-01 00:00:00"
  time:calendar = "noleap"
  time:axis = "T"
  time:long_name = "time"
  time:standard_name = "time"
double time_bnds(lon = 128, bnds = 2)
```

The type of calendar used to define the data (date)
ex) 360day, 365day, 366day, etc.

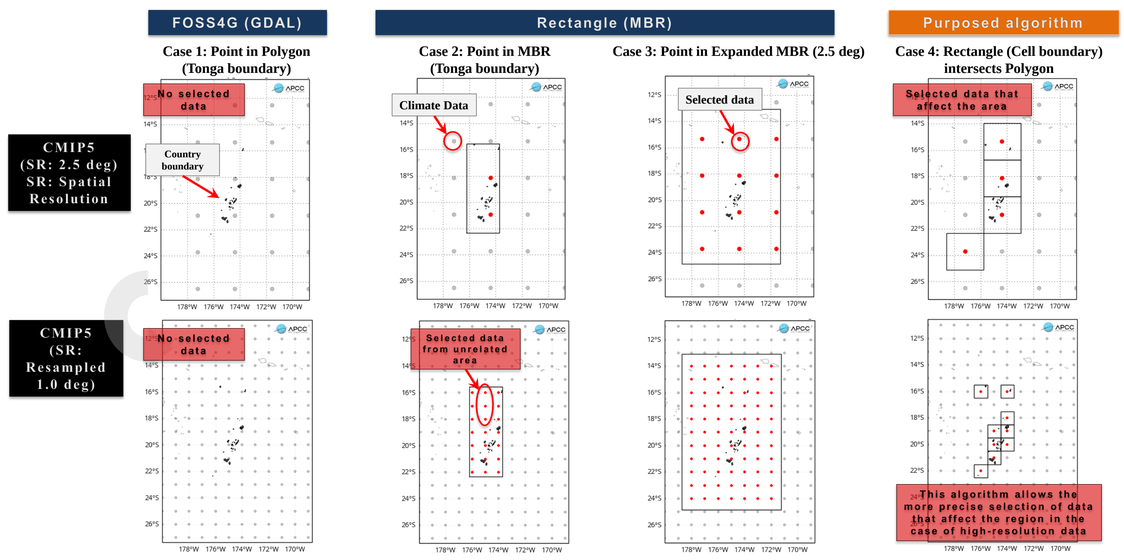
- **Cell boundaries**



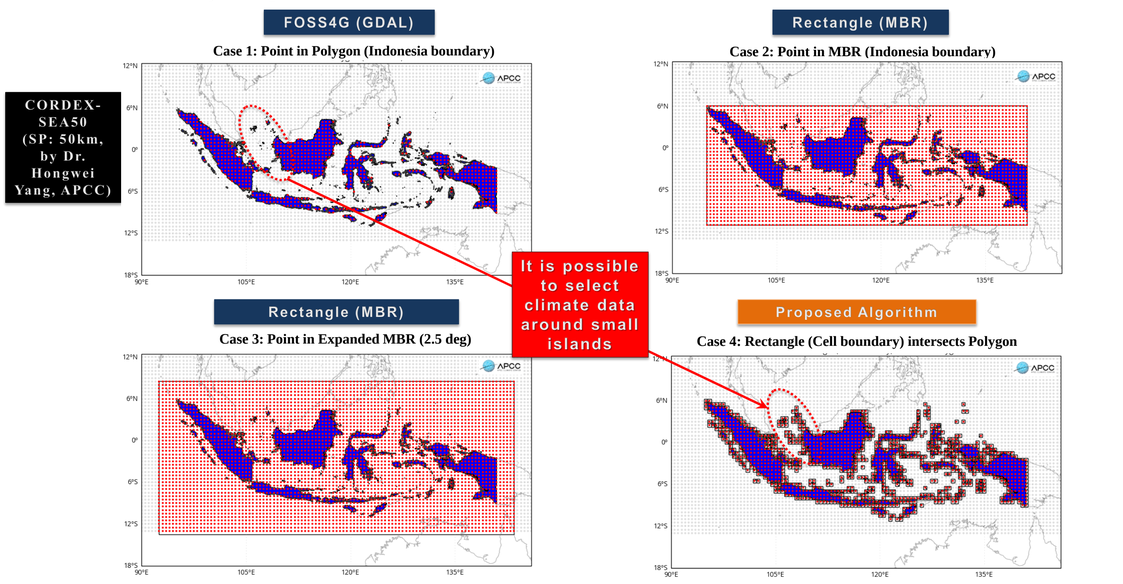
RESULTS

- **Comparison of the results of data extraction using data extraction approaches**

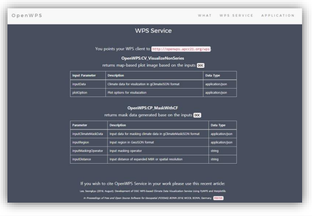
Case study 1: Kingdom of Tonga



Case study 2: Republic of Indonesia



- **Providing masking service as a part of OpenWPS service (http://openwps.apcc21.org)**
  OGC WPS operation name: OpenWPS:CP_MaskWithCF



CONCLUSION & REMARKS

- ❖ The improved algorithm was proposed for extracting climate data with CF metadata and OGC WPS-based climate data masking service was implemented using Apache HTTP Server and PyWPS 4.0.
- ❖ Compared with the traditional methods, the proposed algorithm and service can be used to accurately extract small islands.
- ❖ Users are able to take advantage of the data extraction, using scientific climate data, via the internet more easily and conveniently, making the service accessible to a larger range of users.

FOSS4G BOSTON 2017
August 14-19, 2017
Seaport Hotel & World Trade Center, Boston, MA, USA

APEC CLIMATE CENTER

* Acknowledge: This research was supported by APEC Climate Center