|  |
| --- |
|  |
| Cesium and STK Components Training |
|  |

|  |
| --- |
|  |

Contents

[Getting Started with Cesium 2](#_Toc431758227)

[Download Cesium 2](#_Toc431758228)

[Setup on IIS 2](#_Toc431758229)

[Setup on Node.js 3](#_Toc431758230)

[Cesium Sandcastle 4](#_Toc431758231)

[Live Editing of Code 4](#_Toc431758232)

[Reference Documentation 5](#_Toc431758233)

[Simple Cesium Test Application 6](#_Toc431758234)

[Setup project components 6](#_Toc431758235)

[Entity API 10](#_Toc431758236)

[What is the Entity API? 10](#_Toc431758237)

[Shapes and Volumes 10](#_Toc431758238)

[Materials and Outlines 13](#_Toc431758239)

[Points, Billboards, and Labels & 3D Models 15](#_Toc431758240)

[Data sources, Imagery & Terrain 17](#_Toc431758241)

[Datasource Providers 17](#_Toc431758242)

[Imagery Providers 18](#_Toc431758243)

[Terrain Providers 18](#_Toc431758244)

[STK Components REST Web Service 20](#_Toc431758245)

[Create Asp.NET Web Api Project 20](#_Toc431758246)

[STK Components Calculation Class 22](#_Toc431758247)

[Run and Test 27](#_Toc431758248)

[Bringing it Together – Components and Cesium 29](#_Toc431758249)

[Adding Cesium to Asp.Net 29](#_Toc431758250)

[Connection to REST Api 33](#_Toc431758251)

# Getting Started with Cesium

## Download Cesium

1. The latest version of Cesium can be found on the Cesium website:

<http://cesiumjs.org/downloads.html>

Or on the Cesium GitHub page:

<https://github.com/AnalyticalGraphicsInc/cesium>

## Setup on IIS

1. Follow Microsoft’s instructions on how to set up Internet Information Services on your windows Machine. For windows 7, see this post:

<https://msdn.microsoft.com/en-us/library/ms181052(v=vs.80).aspx>

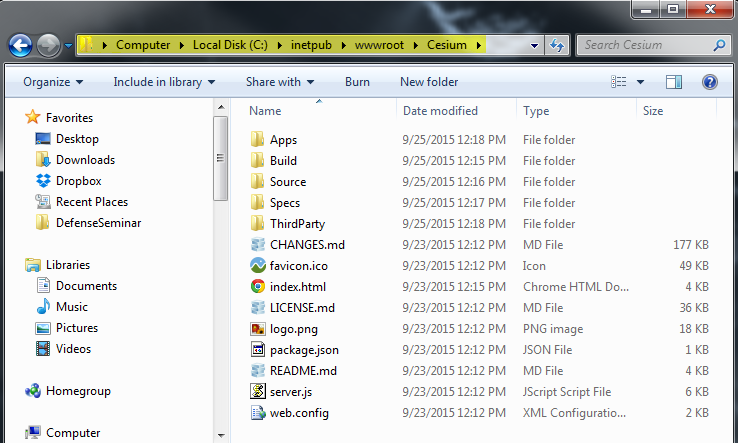
1. After IIS has been enabled and your computer restarted, you should be able to see the default IIS page at <http://localhost>



1. IIS will also have created a new directory that will serve as the physical path to your Default Web Site:

**C:\inetpub\wwwroot**

1. Unzip the Cesium download and move the entire folder into the directory above. You should have a directory structure like this:



1. You can now browse to the default Cesium project page here:

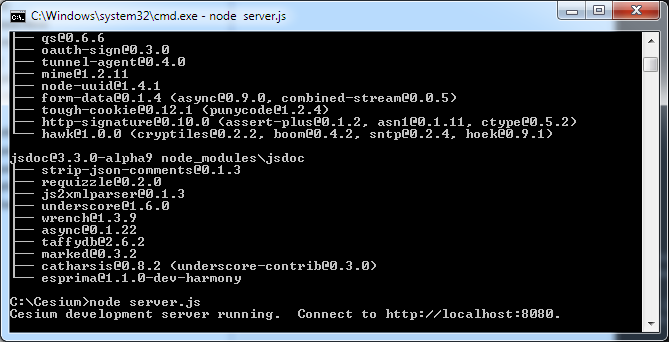
<http://localhost/Cesium/index.html>

## Setup on Node.js

1. Install Node.js from their website using the default install settings:

<http://nodejs.org>

1. Open a command shell in the Cesium root directory and download/install the required modules by executing **“npm install”**. This will create a ‘node\_modules’ directory in the root directory.
2. Finally, start your web server by executing node server.js in the root directory. You should see something like the below:



1. Now that we have a web server up and running, we can launch a browser and navigate to the hello world application here:

<http://localhost:8080/Apps/HelloWorld.html>

# Cesium Sandcastle

Cesium Sandcastle is an application that allows you to edit Cesium code in your web browser without installs or plugins. It comes pre-configured with a huge gallery of code examples, showing off the full range of Cesium capabilities. Applications built with Sandcastle can even be saved and downloaded. Check out the latest version of Sandcastle on the Cesium website:

<http://cesiumjs.org/Cesium/Apps/Sandcastle/index.html>

## Live Editing of Code

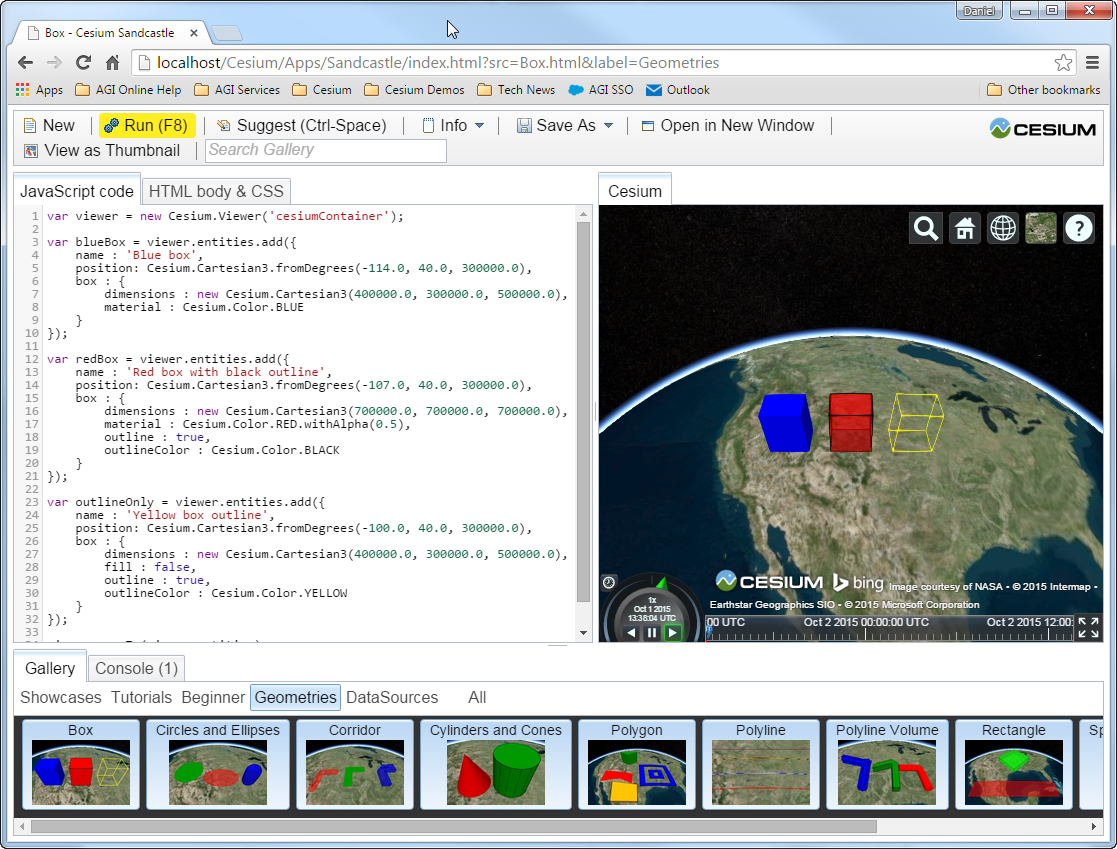
1. Open up the local version of Sandcastle by browsing to the following url:

<http://localhost/Cesium/Apps/Sandcastle/index.html>

1. The default example is the Cesium viewer. Notice the gallery of examples along the bottom of the application. Explore these with the instructor.
2. Try changing some code in one of the Geometries examples, like in the “Box” application, change the dimensions of the red box to:

dimensions : new Cesium.Cartesian3(400000.0, 300000.0, 500000.0)

1. Notice that the “Run” button immediately changes to yellow, indicating that a change has been made and the code can be rerun

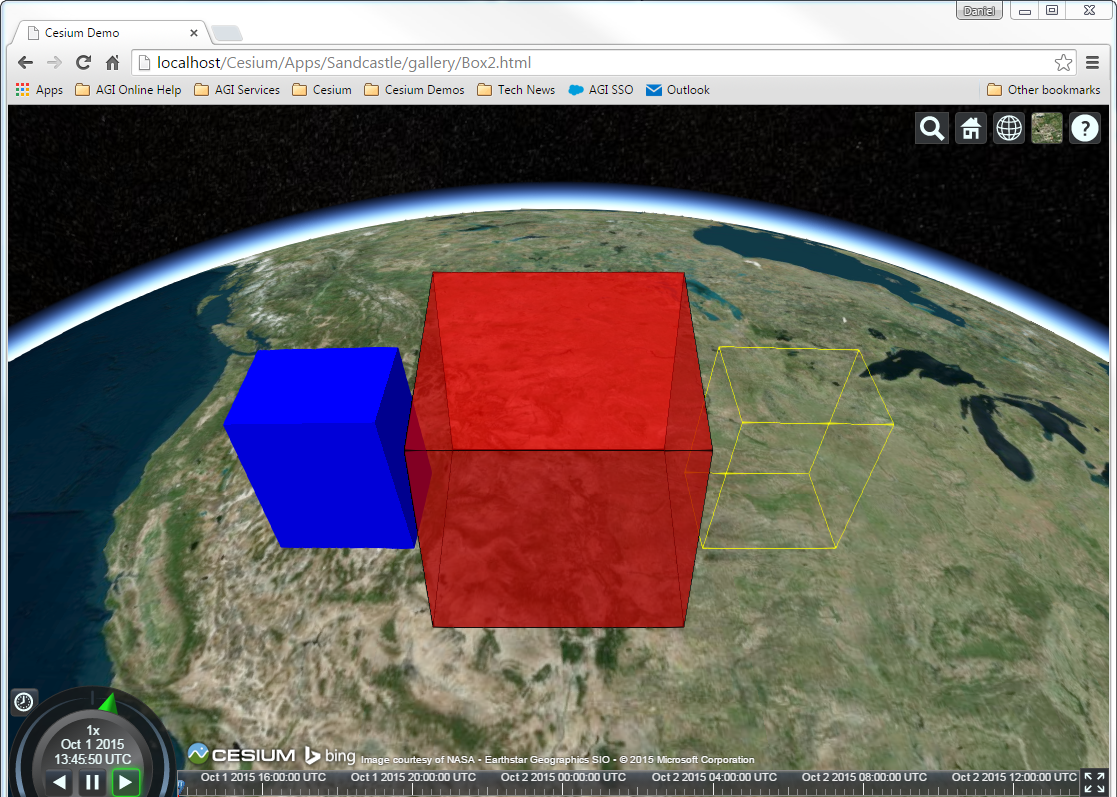


1. Run the code to see the new result.
2. Sandbox also allows you to save the application you have built within the sandbox for use later. Try this by clicking “Save As”. For this html to work, it must live in the Sandbox gallery directory. If you don’t want to overwrite the original Sandbox example, change the name of the html page before moving. Using the Box example above, change the name to Box2.html and move to

**C:\inetpub\wwwroot\Cesium\Apps\Sandcastle\gallery\Box2.html**

1. You can now browse directly to this example like this:

<http://localhost/Cesium/Apps/Sandcastle/gallery/Box2.html>



## Reference Documentation

The Cesium reference documentation can be found on the local installation at

<http://localhost/Cesium/Build/Documentation/index.html>

Within the documentation for each object there are links to the source code for each property and method.

# Simple Cesium Test Application

In this section we will build a simple Cesium client that will allow us to test different aspects of the Entity API as well as DataSources including imagery and terrain. The application will be built using only HTML, JavaScript, and JQuery.

## Setup project components

We will be building this project in WebMatrix3, but any development environment (including simple text editors) would be suitable. We will start by creating the directory structure for the application.

1. Create a new folder called “CesiumClient” in

**C:\CesiumTraining\exercise3**

1. Inside CesiumClient, create folders for the JavaScript and CSS files, “js” and “css”.
2. Copy the Cesium build folder into the “js” folder:

**C:\CesiumTraining\installs\Cesium-1.13\Build\Cesium**

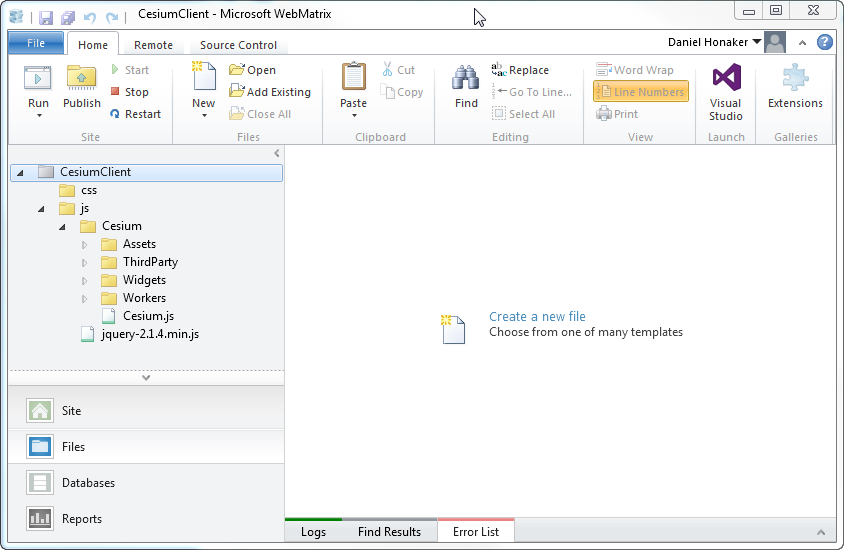
to

**C:\CesiumTraining\exercise3\CesiumClient\js**

1. Copy JQuery to the “js” folder:

**C:\CesiumTraining\materials\jquery-2.1.4.min.js**

1. Start Microsoft WebMatrix3
2. Select “Open” – “Folder” and browse to the new CesiumClient folder. When prompted to create a site on Windows Azure, select “Skip”
3. You should have a project that looks like this:



1. Right click on the CesiumClient folder and select “New File”. Select “HTML” and give it the name “index.html”
2. Add a meaningful title between the <title> tags, like “Cesium Client”.
3. Add jQuery, Cesium’s script, and Cesium’s stylesheet to the <head> below the <title>:

<head>

<meta charset="utf-8" />

<title>Cesium Client</title>

<link rel="stylesheet" href="js/Cesium/Widgets/widgets.css" />

<script src="js/jquery-2.1.4.min.js"></script>

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

</head>

1. Create a <div> in the <body>, this will be the DOM element for the Cesium Widget or Cesium Viewer

<body>

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

</body>

1. Finally, for a quick test, add this script block below the <div> in the <body>

<script>

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

</script>

1. Run the project in Debug mode.
2. Notice that the Cesium viewer does not take up the full screen. We will add a stylesheet that will ensure the “cesiumContainer” is maximized within the browser.
3. Right click on the “css” folder and select “New File” – “CSS” and give it a name like “site.css”
4. Edit the stylesheet like below

#cesiumContainer {

position: absolute;

top: 0;

left: 0;

height: 100%;

width: 100%;

margin: 0;

overflow: hidden;

padding: 0;

font-family: sans-serif;

}

.uiButton {

position: relative;

z-index: 2;

height: 30px;

width: 150px;

background-color: #4cff00;

font-weight: bold;

}

1. Now when we run the application the Cesium viewer will fill the browser.
2. Let’s get this application organized into modules that better conform to AMD patterns. We’ll start with a module called “app” that will contain Cesium initialization code, as well as event handlers for button clicks.
3. Right click on the “js” folder and select “New File” – “JavaScript” and name it “app.js”
4. Build a global variable that has a function for initializing the Cesium Viewer, “initCesium()”:

var app = (function () {

var viewer = {};

var \_initCesium = function () {

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

};

return {

initCesium: \_initCesium

};

})();

1. We will execute the function initCesium() once the DOM is ready. We will use jQuery’s ready() function in a separate script for this purpose.
2. Right click on the “js” folder and select “New File” – “JavaScript” and name it “\_run.js”.
3. Edit the contents of “\_run.js” like below

$(function () {

app.initCesium();

})

1. Load these scripts in index.html and remove the test <script> from the <body>

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="utf-8" />

<title>Cesium Client</title>

<link rel="stylesheet" href="js/Cesium/Widgets/widgets.css" />

<link rel="stylesheet" href="css/site.css" />

<script src="js/jquery-2.1.4.min.js"></script>

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

<script src="js/app.js"></script>

<script src="js/\_run.js"></script>

</head>

<body>

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

</body>

</html>

# Entity API

## What is the Entity API?

The Entity API is a high-level API for data-driven visualization. The goal of the Entity API is to expose a set of consistently designed high-level objects that aggregate related visualization and information into a unified data structure, which we call an Entity. It lets us concentrate on the presentation of our data rather than worrying about the underlying mechanism of visualization. It also provides constructs for easily building complex, time-dynamic visualization in a way that fits naturally alongside static data. The Entity API is able to provide flexible, high-performance visualization while exposing a consistent, easy to learn, and easy to use interface.

## Shapes and Volumes

The Entity API provides the following basic shape and volume types:

|  |
| --- |
| Boxes | http://cesiumjs.org/images/2013/11-04/boxGeometry.png | **entity.box** [Code example](http://cesiumjs.org/Cesium/Apps/Sandcastle/index.html?src=Box.html) [Reference documentation](http://cesiumjs.org/Cesium/Build/Documentation/BoxGraphics.html) |
| Circles and Ellipses | http://cesiumjs.org/images/2013/11-04/ellipseGeometry.png | **entity.ellipse** [Code example](http://cesiumjs.org/Cesium/Apps/Sandcastle/index.html?src=Circles%20and%20Ellipses.html) [Reference documentation](http://cesiumjs.org/Cesium/Build/Documentation/EllipseGraphics.html) |
| Corridor | http://cesiumjs.org/images/2013/11-04/corridorGeometry.png | **entity.corridor** [Code example](http://cesiumjs.org/Cesium/Apps/Sandcastle/index.html?src=Corridor.html) [Reference documentation](http://cesiumjs.org/Cesium/Build/Documentation/CorridorGraphics.html) |
| Cylinder and Cones | http://cesiumjs.org/images/2013/11-04/cylinderGeometry.png | **entity.cylinder** [Code example](http://cesiumjs.org/Cesium/Apps/Sandcastle/index.html?src=Cylinders%20and%20Cones.html) [Reference documentation](http://cesiumjs.org/Cesium/Build/Documentation/CyilnderGraphics.html) |
| Polygons | http://cesiumjs.org/images/2013/11-04/polygonGeometry.png | **entity.polygon** [Code example](http://cesiumjs.org/Cesium/Apps/Sandcastle/index.html?src=Polygon.html) [Reference documentation](http://cesiumjs.org/Cesium/Build/Documentation/PolygonGraphics.html) |
| Polylines | http://cesiumjs.org/images/2013/11-04/polylineGeometry.png | **entity.polyline** [Code example](http://cesiumjs.org/Cesium/Apps/Sandcastle/index.html?src=Polyline.html) [Reference documentation](http://cesiumjs.org/Cesium/Build/Documentation/PolylineGraphics.html) |
| Polyline Volumes | http://cesiumjs.org/images/2013/11-04/polylineVolumeGeometry.png | **entity.polylineVolume** [Code example](http://cesiumjs.org/Cesium/Apps/Sandcastle/index.html?src=Polyline%20Volume.html) [Reference documentation](http://cesiumjs.org/Cesium/Build/Documentation/PolylineVolume.html) |
| Rectangles | http://cesiumjs.org/images/2013/11-04/extentGeometry.png | **entity.rectangle** [Code example](http://cesiumjs.org/Cesium/Apps/Sandcastle/index.html?src=Rectangle.html) [Reference documentation](http://cesiumjs.org/Cesium/Build/Documentation/Rectangle.html) |
| Spheres and Ellipsoids | http://cesiumjs.org/images/2013/11-04/ellipsoidGeometry.png | **entity.ellipsoid** [Code example](http://cesiumjs.org/Cesium/Apps/Sandcastle/index.html?src=Spheres%20and%20Ellipsoids.html) [Reference documentation](http://cesiumjs.org/Cesium/Build/Documentation/EllipsoidGraphics.html) |
| Walls | http://cesiumjs.org/images/2013/11-04/wallGeometry.png | **entity.wall** [Code example](http://cesiumjs.org/Cesium/Apps/Sandcastle/index.html?src=Wall.html) [Reference documentation](http://cesiumjs.org/Cesium/Build/Documentation/WallGraphics.html) |

1. Let’s experiment with a few of these entity types in our application. We’ll start by creating a button that when clicked will execute code to build our entities.
2. Add an <input> of type=”button” and class=“uiButton” below the viewer <div> in the HTML <body>. Give it a value of “Shapes & Volumes”, and an onclick() event that references a function in app.js.
3. Add some code to test a few of the different shape types, for example:

var \_shapesVolumes = function () {

var positions = Cesium.Cartesian3.fromDegreesArray([

-115.0, 37.0,

-115.0, 33.0,

-107.0, 33.0,

-107.0, 37.0]);

var holes = Cesium.Cartesian3.fromDegreesArray([

-113.0, 36.5,

-113.0, 35.5,

-109.0, 35.5,

-109.0, 36.5]);

var ellipsoid = new Cesium.Entity({

id: 'ellipsoid1',

name: 'ellipsoid 1',

position: Cesium.Cartesian3.fromDegrees(-100.0, 30.0, 300000.0),

ellipsoid: {

radii: new Cesium.Cartesian3(200000.0, 200000.0, 300000.0),

outline: true

}

});

var wall = new Cesium.Entity({

id: 'wall1',

name: 'wall 1',

wall: {

positions: positions,

maximumHeights: [70000.0, 190000.0, 40000.0, 150000.0],

minimumHeights: [0, 20000.0, 30000.0, 50000.0],

outline: true

}

});

var polygon = new Cesium.Entity({

id: 'polygon1',

name: 'polygon 1',

polygon: {

hierarchy: {

positions: positions,

holes: [{ positions: holes}]

},

outline: true

}

});

var box = new Cesium.Entity();

box.id = 'box1';

box.name = 'box 1';

box.position = Cesium.Cartesian3.fromDegrees(-90, 40.0, 200000.0);

var boxgraphics = new Cesium.BoxGraphics();

boxgraphics.dimensions = new Cesium.Cartesian3(400000.0, 600000.0, 300000.0);

boxgraphics.outline = true;

box.box = boxgraphics;

viewer.entities.add(ellipsoid);

viewer.entities.add(wall);

viewer.entities.add(polygon);

viewer.entities.add(box);

};

## Materials and Outlines

Regardless of their geometric definition, all shapes and volumes have a common set of properties that control their appearance. The fill property is a boolean that specifies if the interior of the surface is filled in, while the outline property controls whether the edges of the shape are outlined.

When fill is set to true, the material property determines what that filling looks like. There are two built in material properties for Entities, “Color” and “Image”. These materials can be assigned directly like this:

polygon.material = 'images/colorado.jpg';

box.material = Cesium.Color.RED;

For more complex materials, we need to create a MaterialProperty() instance ourselves. Currently, Entity shapes and volumes support colors, images, checkerboard, stripe, and grid materials. Here are a list of the interfaces:

* ColorMaterialProperty
* CompositeMaterialProperty
* GridMaterialProperty
* ImageMaterialProperty
* PolylineGlowMaterialProperty
* PolylineOutlineMaterialProperty
* StripeMaterialProperty

1. Try applying some materials to the entities you built in the last step. See the code below for an example of how to apply the Color, Image, and Grid materials.

var \_shapesVolumes = function () {

var positions = Cesium.Cartesian3.fromDegreesArray([

-115.0, 37.0,

-115.0, 33.0,

-107.0, 33.0,

-107.0, 37.0]);

var holes = Cesium.Cartesian3.fromDegreesArray([

-113.0, 36.5,

-113.0, 35.5,

-109.0, 35.5,

-109.0, 36.5]);

var ellipsoid = new Cesium.Entity({

id: 'ellipsoid1',

name: 'ellipsoid 1',

position: Cesium.Cartesian3.fromDegrees(-100.0, 30.0, 300000.0),

ellipsoid: {

radii: new Cesium.Cartesian3(200000.0, 200000.0, 300000.0),

outline: true

}

});

ellipsoid.ellipsoid.material = Cesium.Color.RED;

ellipsoid.ellipsoid.outlineColor = Cesium.Color.fromBytes(0, 0, 255, 255);

var wall = new Cesium.Entity({

id: 'wall1',

name: 'wall 1',

wall: {

positions: positions,

maximumHeights: [70000.0, 190000.0, 40000.0, 150000.0],

minimumHeights: [0, 20000.0, 30000.0, 50000.0],

outline: true

}

});

wall.wall.material = new Cesium.ImageMaterialProperty({

image: 'images/colorado.jpg',

repeat: new Cesium.Cartesian2(4, 1)

});

var polygon = new Cesium.Entity({

id: 'polygon1',

name: 'polygon 1',

polygon: {

hierarchy: {

positions: positions,

holes: [{ positions: holes}]

},

outline: true

}

});

polygon.polygon.material = 'images/colorado.jpg';

var box = new Cesium.Entity();

box.id = 'box1';

box.name = 'box 1';

box.position = Cesium.Cartesian3.fromDegrees(-90, 40.0, 200000.0);

var boxgraphics = new Cesium.BoxGraphics();

boxgraphics.dimensions = new Cesium.Cartesian3(400000.0, 600000.0, 300000.0);

boxgraphics.outline = true;

boxgraphics.material = new Cesium.GridMaterialProperty({

color: Cesium.Color.LIME,

cellAlpha: 0,

lineCount: new Cesium.Cartesian2(12, 12)

});

box.box = boxgraphics;

viewer.entities.add(ellipsoid);

viewer.entities.add(wall);

viewer.entities.add(polygon);

viewer.entities.add(box);

};

If you are working with primitives outside of the Entity API, you can build up a fabric from scratch using materials and uniforms. There are a large number of built in material types that you can use.

1. Explore the types listed here with the instructor:

<http://cesiumjs.org/Cesium/Build/Documentation/Material.html>

Fabric is a JSON schema for describing materials in Cesium. Materials represent the appearance of an object such as polygons, polylines, ellipsoids, and sensors.

Materials can be as simple as draping an image over an object, or applying a pattern such as stripes or a checkerboard. Using Fabric and GLSL, new materials can be scripted from scratch or created by combining existing materials in a hierarchy.

1. Explore some of the possible materials with the instructor in the Sandcastle materials showcase.

<http://localhost/Cesium/Apps/Sandcastle/index.html?src=Materials.html&label=Showcases>

## Points, Billboards, and Labels & 3D Models

Leaving shapes and volumes behind, let’s look at how to visualize specific points of interest in Cesium.

Creating a graphical point or label is simple, we just need to specify a position for our entity and point and label objects for the visualization. The four entity graphics types we will explore here are

* PointGraphics
* LabelGraphics
* BillboardGraphics
* ModelGraphics

PointGraphics and LabelGraphics are self-explanatory. Billboards are a marker that is always facing the user. ModelGraphics load a 3D model in glTF format.

1. Try adding some of these types in our example.
2. Create a new <input> button with the name “Billboards & Models”, and an onclick event like:

<body>

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

<input type="button" class="uiButton" value="Shapes & Volumes" onclick="app.shapesVolumes()"/>

<input type="button" class="uiButton" value="Billboards & Models" onclick="app.billboardsModels()"/>

</body>

1. Now add a function to app.js like the code below:

var \_billboardsModels = function () {

var point = new Cesium.Entity({

id: 'point1',

name: 'point 1',

position: Cesium.Cartesian3.fromDegrees(-95, 35, 10.0),

point: {

pixelSize: 12,

color: Cesium.Color.AQUA

},

label: {

text: 'Aqua Point',

font: '16pt sans-serif',

verticalOrigin: Cesium.VerticalOrigin.BOTTOM,

horizontalOrigin: Cesium.HorizontalOrigin.LEFT

}

});

var billboard = new Cesium.Entity({

id: 'billboard1',

name: 'billboard 1',

position: Cesium.Cartesian3.fromDegrees(-105, 40),

billboard: {

image: 'images/Facility.png',

scale: 1.5

}

});

var model = new Cesium.Entity({

id: 'model1',

name: 'model 1',

position: Cesium.Cartesian3.fromDegrees(-100, 45),

model: {

uri: 'images/Cesium\_Man.gltf'

}

});

viewer.entities.add(point);

viewer.entities.add(billboard);

viewer.entities.add(model);

};

# DataSources, Imagery & Terrain

## DataSource Providers

Cesium provides the ability to load data from GeoJSON, KML, and CZML documents using the DataSource interface. To build a data source for these objects requires the following interfaces

* GeoJsonDataSource
* KmlDataSource
* CzmlDataSource

Once created, these data sources can be added to the Cesium viewer like this:

var kmldata = Cesium.KmlDataSource.load('mydata.kmz')

viewer.dataSources.add(kmldata);

This creates a collection of entities in the Cesium Viewer that are grouped according to that specific data source object. You can interact with the individual entities in this collection through the DataSource.entities() member, which will return an EntityCollection object.

You can also group together any number of Cesium.Entity() objects into a custom data source using the CustomDataSource() interface. This is useful for grouping entities together by layer or type. In our previous example where we created shapes and volumes, we could combine these into a single custom data source like this:

var myCustomData = new Cesium.CustomDataSource('myData');

myCustomData.entities.add(ellipsoid);

myCustomData.entities.add(wall);

myCustomData.entities.add(polygon);

myCustomData.entities.add(box);

viewer.dataSources.add(myCustomData);

1. Experiment with adding data sources to our application. There is an example geojson file in the materials directory, “us\_counties.json”, you can add this to the project in a new folder called “data”.
2. Create an <input> button that calls a function like this:

var \_dataSources = function () {

var geojson = new Cesium.GeoJsonDataSource('USCounties');

geojson.load('data/us\_counties.json');

viewer.dataSources.add(data);

};

## Imagery Providers

Cesium provides the ability to load imagery data from a number of different sources using the ImageryProvider() interface. The available interfaces for imagery are

* ArcGisMapServerImageryProvider
* SingleTileImageryProvider
* BingMapsImageryProvider
* GoogleEarthImageryProvider
* MapboxImageryProvider
* OpenStreetMapImageryProvider
* WebMapTileServiceImageryProvider
* WebMapServiceImageryProvider

1. Experiment with adding imagery providers to your application. Create an <input> button that calls a function like this:

var \_imageryProviders = function () {

var provider = new Cesium.WebMapServiceImageryProvider({

url: 'http://mesonet.agron.iastate.edu/cgi-bin/wms/goes/conus\_ir.cgi',

layers: 'goes\_conus\_ir',

parameters: {

transparent: 'true',

format: 'image/png'

}

});

var layer = viewer.imageryLayers.addImageryProvider(provider);

layer.alpha = 0.5;

};

## Terrain Providers

Cesium allows the loading of streaming terrain data as well as single terrain tiles. Streaming terrain data is supported through the TerrainProvider interface, which includes these types

* EllipsoidTerrainProvider
* ArcGisImageServerTerrainProvider
* VRTheWorldTerrainProvider
* CesiumTerrainProvider

Single terrain tiles are also supported using the TerrainData() interface which includes these types

* HeightmapTerrainData
* QuantizedMeshTerrainData

1. Experiment with adding terrain providers to your application. Create an <input> button that calls a function like this:

var \_terrainProviders = function () {

var provider = new Cesium.CesiumTerrainProvider({

url: '//assets.agi.com/stk-terrain/world',

requestVertexNormals: true

});

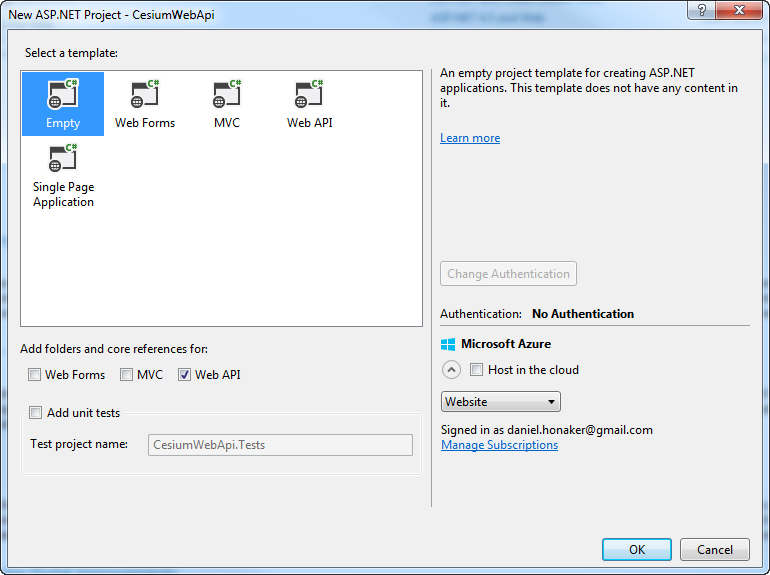
viewer.terrainProvider = provider;

};

# STK Components REST Web Service

## Create Asp.NET Web Api Project

1. Start Visual Studio and create a new ASP.NET Web Application named, “CesiumWebApi”
2. Under ASP.NET Templates, select empty. Then check the box for “Web Api”



1. Right click on the Controllers folder in the Solution Explorer and select “New – Controller”. Name the controller “SatelliteController”.
2. Create a GET REST service called Test()

using System.Net;

using System.Net.Http;

using System.Web.Http;

namespace CesiumWebApi.Controllers

{

public class SatelliteController : ApiController

{

[HttpGet]

public HttpResponseMessage Test()

{

return Request.CreateResponse(HttpStatusCode.OK, "Working");

}

}

}

1. Then edit the default Web Api Routes so that we can interact with the Controller’s actions.

using System.Web.Http;

namespace CesiumWebApi

{

public static class WebApiConfig

{

public static void Register(HttpConfiguration config)

{

// Web API configuration and services

// Web API routes

config.MapHttpAttributeRoutes();

config.Routes.MapHttpRoute(

name: "CesiumWebApi",

routeTemplate: "api/{controller}/{action}",

defaults: new { action = RouteParameter.Optional }

);

}

}

}

1. Test the service by running Visual Studio in debug mode. The address it opens in the browser will be your IIS express domain and port number, i.e. <http://localhost:54321/>
2. From there you can test your GET service like this:

<http://localhost:54321/api/Satellite/Test>

Which should return “Working”.

## STK Components Calculation Class

Now it is time to add the calculation classes that our Web Api will interact with on the Server. To do that we will create a separate project with two fundamental types, Calculations and Models. Our definition of “model” is derived from the MVC pattern – A model stores data that is retrieved according to commands from the Controller and displayed in the View (or in our case, additionally passed to the calculation classes). Our calculation classes will allow us to execute STK Component and STK Server analysis, and return that analysis as Model data to the Controllers.

1. Right click on the Solution and select Add – New Project.
2. Choose C# Class Library as your project type, and name it StkComponentsCalculations
3. Add a new folder called “Models”. To that folder add a new class called “SatelliteState”
4. Create the SatelliteState model like below:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace StkComponentsCalculations.Models

{

public class SatelliteState

{

public SatelliteState() { }

public DateTime StartTime { get; set; }

public DateTime StopTime { get; set; }

public string SatelliteName { get; set; }

public double SemiMajorAxis { get; set; }

public double Eccentricity { get; set; }

public double Inclination { get; set; }

public double ArgOfPeriapsis { get; set; }

public double RAAN { get; set; }

public double TrueAnomaly { get; set; }

public void Verify()

{

if (StopTime.Ticks < StartTime.Ticks)

{

throw new ArgumentException("StartTime must be less than StopTime");

}

if (String.IsNullOrWhiteSpace(SatelliteName))

{

throw new ArgumentException("Satellite Name must be defined");

}

if (Inclination > 180 || Inclination < 0)

{

throw new ArgumentException("Inclination range is 0 < inc < 180");

}

}

}

}

1. Add a new folder called “Calculations”. To that folder add a new class called “PropagateSatelliteState”.
2. The PropagateSatelliteState calculation class will require some STK Component libraries. Add these dll libraries to the StkComponentsCalculations project references:

C:\CesiumTraining\installs\STKComponentsForDotNet2015r2\Assemblies\ AGI.Foundation.Cesium.dll

C:\CesiumTraining\installs\STKComponentsForDotNet2015r2\Assemblies\AGI.Foundation.Core.dll

C:\CesiumTraining\installs\STKComponentsForDotNet2015r2\Assemblies\AGI.Foundation.Models.dll

C:\CesiumTraining\installs\STKComponentsForDotNet2015r2\Assemblies\ AGI.Foundation.Platforms.dll

1. Edit the “PropagateSatelliteState” class like below:

using System.Drawing;

using System.IO;

using StkComponentsCalculations.Models;

using AGI.Foundation.Celestial;

using AGI.Foundation.Coordinates;

using AGI.Foundation.Propagators;

using AGI.Foundation.Platforms;

using AGI.Foundation.Geometry;

using AGI.Foundation;

using AGI.Foundation.Time;

using AGI.Foundation.Cesium;

namespace StkComponentsCalculations.Calculations

{

public class SatellitePropagator

{

public SatellitePropagator() { }

#region Properties, Variables

private SatelliteState \_state;

private Platform \_satellite;

private EarthCentralBody \_earth;

public SatelliteState State

{

get

{

return \_state;

}

}

public Platform Satellite

{

get

{

return \_satellite;

}

}

#endregion

#region Public Calculation Methods

public void Propagate(SatelliteState state)

{

state.Verify();

\_state = state;

\_satellite = PropagatePlatform();

StylePlatform();

}

public string ReturnCzml()

{

CzmlDocument czml = new CzmlDocument();

czml.Name = "Simple Example";

czml.Description = "Simple Web Service Example";

czml.RequestedInterval = new TimeInterval(

new JulianDate(\_state.StartTime),

new JulianDate(\_state.StopTime));

czml.Clock = new Clock

{

Interval = czml.RequestedInterval,

Multiplier = 60.0

};

czml.ObjectsToWrite.Add(\_satellite);

StringWriter stringWriter = new StringWriter();

czml.WriteDocument(stringWriter);

return stringWriter.ToString(); ;

}

#endregion

#region Private Calculations

private Platform PropagatePlatform()

{

if(\_earth == null)

{

\_earth = CentralBodiesFacet.GetFromContext().Earth;

}

KeplerianElements keplerianElements = new KeplerianElements(

\_state.SemiMajorAxis,

\_state.Eccentricity,

Trig.DegreesToRadians(\_state.Inclination),

Trig.DegreesToRadians(\_state.ArgOfPeriapsis),

Trig.DegreesToRadians(\_state.RAAN),

Trig.DegreesToRadians(\_state.TrueAnomaly),

WorldGeodeticSystem1984.GravitationalParameter);

TwoBodyPropagator propagator = new TwoBodyPropagator(

new JulianDate(\_state.StartTime),

\_earth.InertialFrame,

keplerianElements);

Platform satellite = new Platform();

satellite.Name = \_state.SatelliteName;

satellite.LocationPoint = propagator.CreatePoint();

satellite.OrientationAxes = new AxesVehicleVelocityLocalHorizontal(

\_earth.InertialFrame,

satellite.LocationPoint);

return satellite;

}

private void StylePlatform()

{

PointGraphicsExtension pointExtension = new PointGraphicsExtension(new PointGraphics

{

PixelSize = 10,

Color = new ConstantCesiumProperty<Color>(Color.White)

});

\_satellite.Extensions.Add(pointExtension);

LabelGraphicsExtension labelExtension = new LabelGraphicsExtension(new LabelGraphics

{

Text = new ConstantCesiumProperty<string>(\_satellite.Name),

FillColor = new ConstantCesiumProperty<Color>(Color.White),

PixelOffset = new ConstantCesiumProperty<Rectangular>(new Rectangular(0, 0))

});

\_satellite.Extensions.Add(labelExtension);

PathGraphicsExtension pathExtension = new PathGraphicsExtension(new PathGraphics

{

Show = true,

Width = 2,

TrailTime = 10000,

LeadTime = 10000,

Material = new PolylineOutlineMaterialGraphics

{

Color = Color.White,

OutlineColor = Color.Black,

OutlineWidth = 0.5

}

});

\_satellite.Extensions.Add(pathExtension);

CesiumReferenceFrameExtension refExtension = new CesiumReferenceFrameExtension();

refExtension.CesiumReferenceFrame = \_earth.InertialFrame;

\_satellite.Extensions.Add(refExtension);

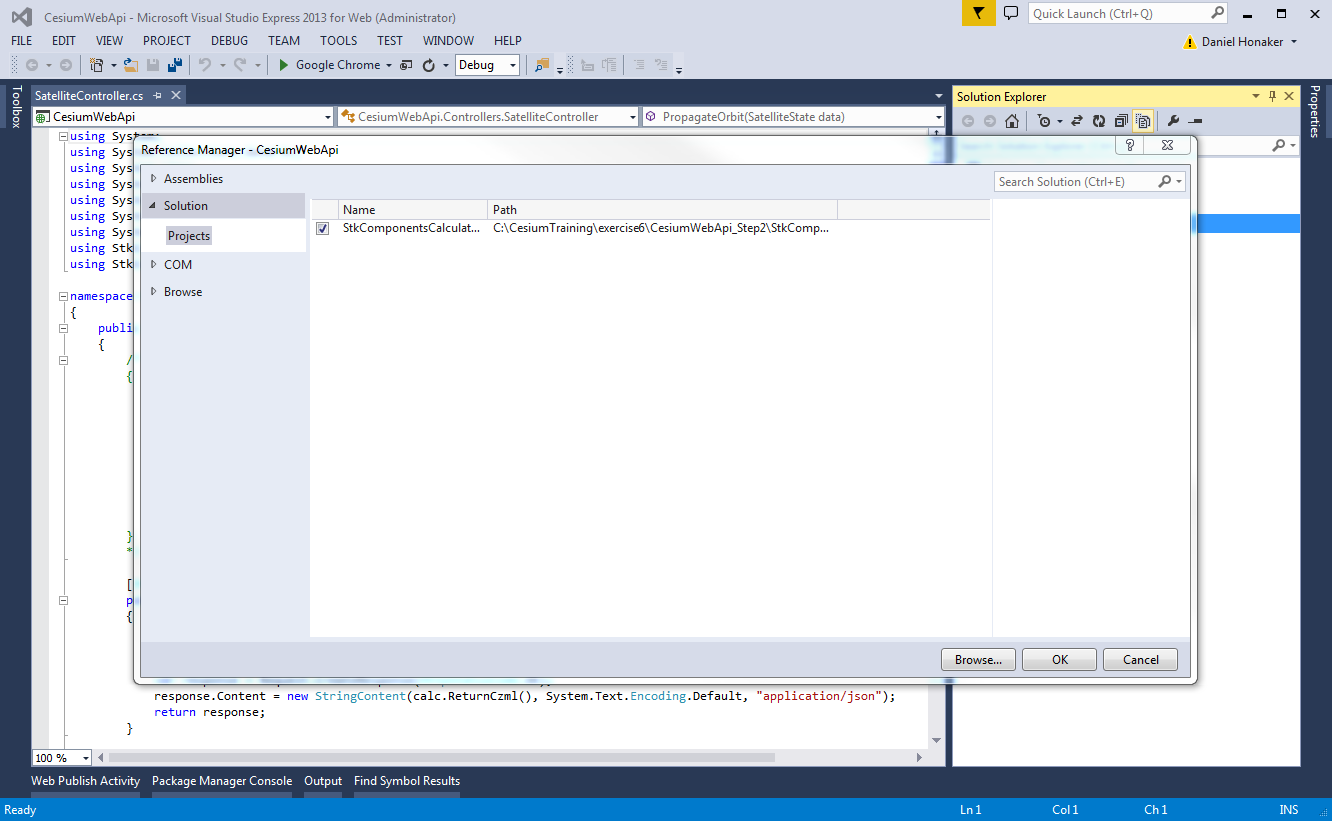
}

#endregion

}

}

1. Make sure that both the new classes are “public” classes.
2. Rebuild the solution.
3. Now add the StkComponentsCalculations project as a reference to the CesiumWebApi.



1. In the SatelliteController, add “using” for the calculation library and model library.

using StkComponentsCalculations.Models;

using StkComponentsCalculations.Calculations;

1. Now we can create a new Web Api Contoller Action to accept SatelliteState data, and return results from the PropagateSatelliteState calculation class.
2. Add a new POST method/action named “Propagate” to your SatelliteController like below:

[HttpPost]

public HttpResponseMessage Propagate(SatelliteState data)

{

SatellitePropagator satellitePropagator = new SatellitePropagator();

satellitePropagator.Propagate(data);

string czml = satellitePropagator.ReturnCzml();

var response = Request.CreateResponse(HttpStatusCode.OK);

response.Content = new StringContent(czml, System.Text.Encoding.Default, "application/json");

return response;

}

This controller will receive satellite state data from the client, propagate the satellite, and return czml.

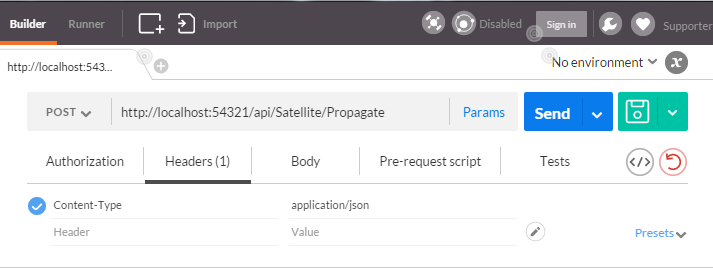
## Run and Test

Now we’re ready to test our web service!

1. Build the project and run in debug mode.
2. Open Postman so that we can test the HTML POST action we created.
3. Change the request type to POST and add the url for the Propagate action:

<http://localhost:54321/api/Satellite/Propagate>

1. Add the header “Content-Type” and select “application/json”



1. In the body we will send json data formatted to match the SatelliteState class that our controller expects. Here is a sample json object that you can use:

{

"StartTime": "01-Jul-2015 00:00:00.000",

"StopTime": "02-Jul-2015 00:00:00.000",

"SatelliteName": "ScienceSat",

"SemiMajorAxis": 7000000.000,

"Eccentricity": 0.001,

"Inclination": 85.0,

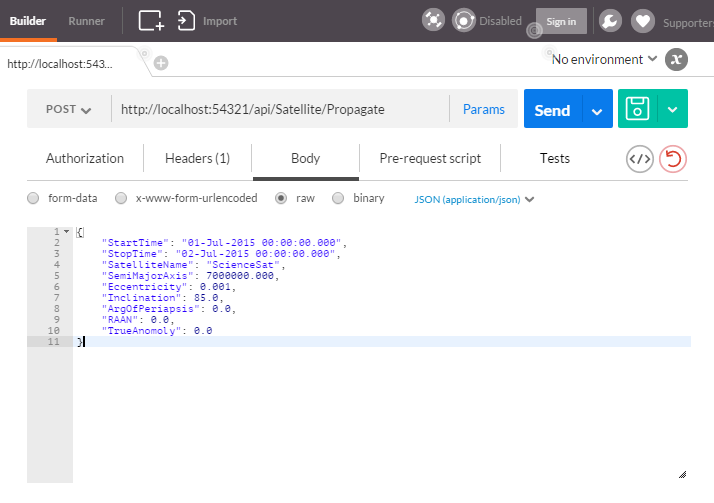
"ArgOfPeriapsis": 0.0,

"RAAN": 0.0,

"TrueAnomoly": 0.0

}

You are now ready to send the request



You should receive CZML data as a json object from your web service.

# Bringing it Together – Components and Cesium

## Adding Cesium to Asp.Net

We now have a functioning Web Api that returns CZML data. Let’s now build a Cesium based web client to interact with the service. Start by adding the Cesium libraries to the CesiumWebApi project.

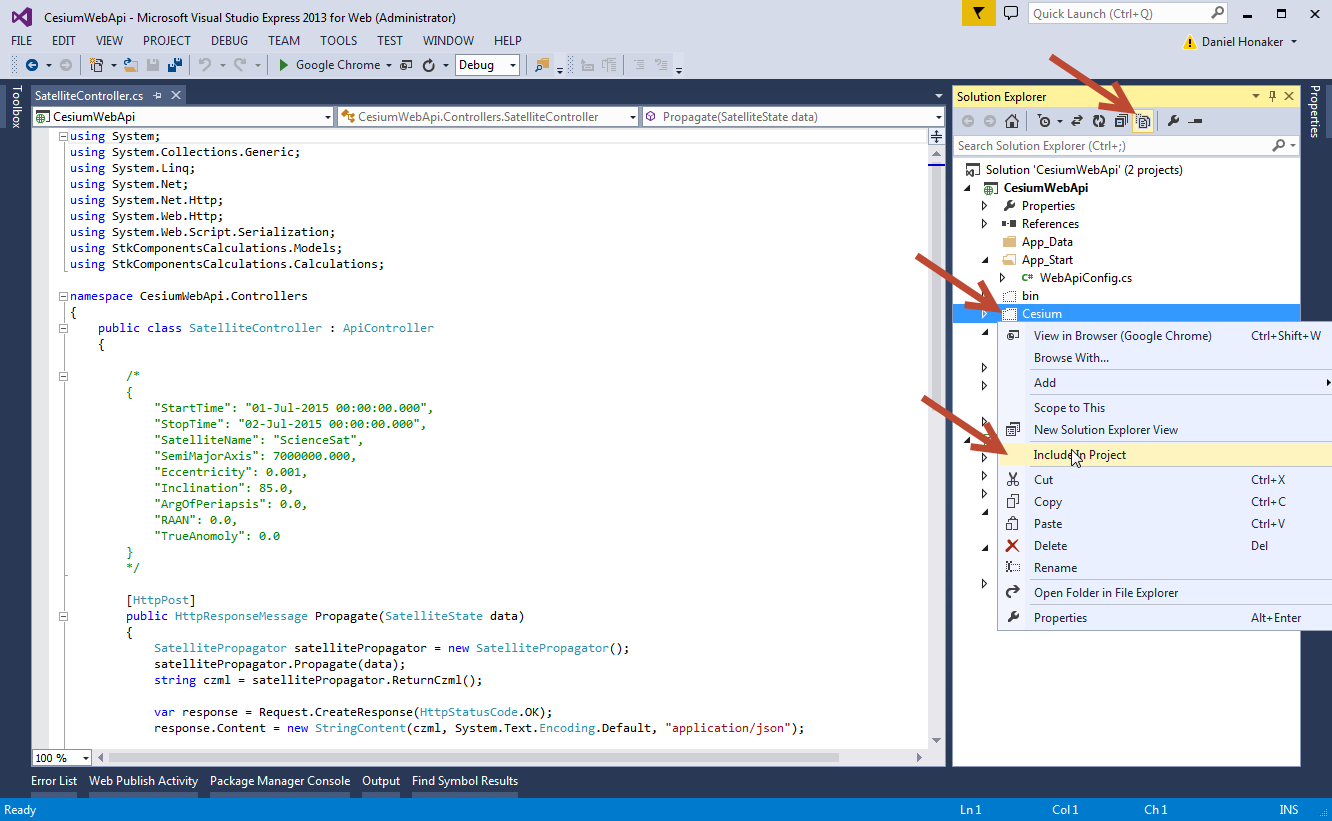
1. Copy the Cesium build folder:

**C:\CesiumTraining\installs\Cesium-1.13\Build\Cesium**

To the CesiumWebApi project folder:

**C:\CesiumTraining\exercise6\CesiumWebApi\_Step3\CesiumWebApi**

1. Click on the CesiumWebApi project in the Solution Explorer, and choose to “Show All Files”
2. Right click on the Cesium folder and select to “Include in Project”



Now we will add a new HTML file to the CesiumWebApi project to act as the default HTML page in which we’ll embed the Cesium Viewer.

1. Right click on the CesiumWebApi project and choose to add an HTML page. Name it “index.html”.
2. Edit the <title> tag to:

<title>Cesium Web Api</title>

1. Add the Cesium script and stylesheet to the <head> below the <title>:

<link rel="stylesheet" href="Cesium/Widgets/widgets.css" />

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

1. Create a <div> in the <body>, this will be the DOM element for the Cesium Viewer

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

1. Finally, for a quick test, add this script block below the <div> in the <body>

<script>

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

</script>

1. Run the project in Debug mode.

Notice that the Cesium Viewer does not take up the full screen. We will add a custom stylesheet to correct this. Later we will be adding some UI controls to the HTML with custom styling which will help us to interact with the STK web service. Let’s add the client side libraries and stylesheets that will make this possible.

1. Select Tools – NuGet Package Manager – Manage NuGet Packages for Solution…
2. Under Online – nuget.org, select to install “Bootstrap CSS” and “jQuery” for the CesiumWebApi project.



1. Close the NuGet Package Manager.
2. Right click on the Content folder in the CesiumWebApi project and choose to Add a “style sheet”. Call it “site.css”.
3. Edit the site.css contents like below:

#cesiumContainer {

position: absolute;

top: 0;

left: 0;

height: 100%;

width: 100%;

margin: 0;

overflow: hidden;

padding: 0;

font-family: sans-serif;

}

.buttonHolder {

position: absolute;

top: 12px;

left: 5px;

padding: 7px;

width: 370px;

background-color: rgba(42, 42, 42, 0.8);

border: 2px solid grey;

border-radius: 7px;

vertical-align: middle;

color: grey;

}

.input-group-addon {

width: 150px;

}

html {

height: 100%;

}

body {

padding: 0;

margin: 0;

overflow: hidden;

height: 100%;

}

Now let’s set up the basics of our client side JavaScript application. We will need code to configure the Cesium widget, interact with the HTML Ui Controls, and interact with the STK web service. We will utilize jQuery to interact with the DOM elements and the AJAX methods for working with the web services. First, we will define a single module for the application which we will call from the jQuery ready function as well as the HTML controls.

1. Right click on the Scripts folder and select to add a new JavaScript File. Name it, “app.js”.
2. Edit the contents of “app.js” like below:

var app = (function () {

var viewer = {};

var \_initCesium = function () {

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

};

return {

initCesium: \_initCesium

};

})();

Next we need to call the initCesium() function once the DOM has finished loading. We will use the jQuery ready function for this.

1. Right click on the Scripts folder and select to add a new JavaScript File. Name it, “\_run.js”.
2. Edit the contents of “\_run.js” like below:

$(function () {

app.initCesium();

});

1. Finally, we need to add these scripts and stylesheets to the master HTML page and remove the test script block from the <body>.
2. Edit index.html like below:

<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml">

<head>

<title>Cesium Web Api</title>

<link rel="stylesheet" href="Cesium/Widgets/widgets.css" />

<link rel="stylesheet" href="Content/bootstrap.css" />

<link rel="stylesheet" href="Content/site.css" />

<script src="Scripts/jquery-2.1.4.min.js"></script>

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

<script src="Scripts/app.js"></script>

<script src="Scripts/\_run.js"></script>

</head>

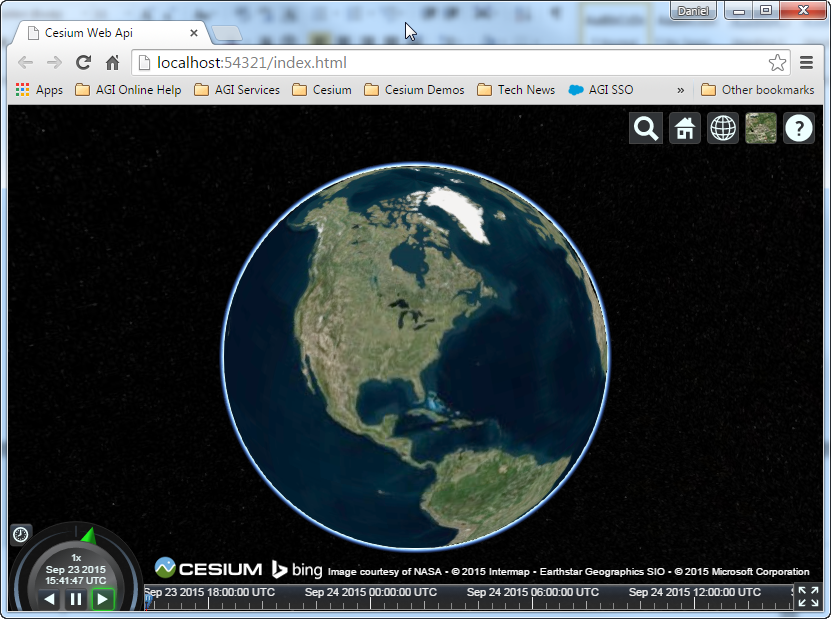
<body>

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

</body>

</html>

1. Run the application in debug to ensure that everything is configured correctly. You should see the Cesium viewer filling 100% of the browser.



## Connection to REST Api

We now need to create the UI controls that will collect the user input for the satellite state, and then send that data to the web service.

1. Modify the <body> of the master HTML document like below. Note that we are using bootstrap’s form controls to stylize the text boxes and buttons.

<body>

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

<div class="buttonHolder">

<div class="input-group">

<span class="input-group-addon">Satellite Name</span>

<input type="text" id="satelliteName" class="form-control" value="ScienceSat"

aria-describedby="satelliteName" />

</div>

<div class="input-group">

<span class="input-group-addon">Start Time</span>

<input type="text" id="startTime" class="form-control" value="01-Jan-2015 00:00:00.000"

aria-describedby="startTime" />

</div>

<div class="input-group">

<span class="input-group-addon">Stop Time</span>

<input type="text" id="stopTime" class="form-control" value="02-Jan-2015 00:00:00.000"

aria-describedby="stopTime" />

</div>

<div class="input-group">

<span class="input-group-addon">Semi Major Axis</span>

<input type="text" id="semiMajorAxis" class="form-control" value="7000000.00"

aria-describedby="semiMajorAxis" />

</div>

<div class="input-group">

<span class="input-group-addon">Eccentricity</span>

<input type="text" id="eccentricity" class="form-control" value="0.005"

aria-describedby="eccentricity" />

</div>

<div class="input-group">

<span class="input-group-addon">Inclination</span>

<input type="text" id="inclination" class="form-control" value="97.6"

aria-describedby="inclination" />

</div>

<div class="input-group">

<span class="input-group-addon">Arg. Of Periapsis</span>

<input type="text" id="argOfPeriapsis" class="form-control" value="0.0"

aria-describedby="argOfPeriapsis" />

</div>

<div class="input-group">

<span class="input-group-addon">RAAN</span>

<input type="text" id="raan" class="form-control" value="0.0"

aria-describedby="raan" />

</div>

<div class="input-group">

<span class="input-group-addon">True Anomaly</span>

<input type="text" id="trueAnomaly" class="form-control" value="0.0"

aria-describedby="trueAnomaly" />

</div>

<br />

<input type="button" class="btn btn-success" value="Propagate" onclick="app.propagateSatellite()" />

</div>

</body>

1. Next we need to add the propagateSatellite() function to app.js. Modify the app.js code like below:

var app = (function () {

var viewer = {};

var url = 'http://localhost:54321/api/Satellite/Propagate';

var \_initCesium = function () {

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

};

var \_propagateSatellite = function () {

var postData = {

SatelliteName: $('#satelliteName').val(),

StartTime: $('#startTime').val(),

StopTime: $('#stopTime').val(),

SemiMajorAxis: $('#semiMajorAxis').val(),

Eccentricity: $('#eccentricity').val(),

Inclination: $('#inclination').val(),

ArgOfPeriapsis: $('#argOfPeriapsis').val(),

RAAN: $('#raan').val(),

TrueAnomoly: $('#trueAnomaly').val()

};

ajaxRequest(url, 'POST', postData).then(addCzmlToViewer);

};

var ajaxRequest = function (url, type, data) {

var options = {

url: url,

headers: {

Accept: "application/json"

},

contentType: "application/json",

dataType: 'json',

cache: false,

type: type,

data: JSON.stringify(data)

}

return $.ajax(options);

};

var addCzmlToViewer = function (data) {

viewer.dataSources.add(Cesium.CzmlDataSource.load(data));

};

return {

initCesium: \_initCesium,

propagateSatellite: \_propagateSatellite

};

})();

1. Run the solution in debug and click the Propagate button to see the results!

