PUTTING LIFE ON MARS: Using Computer Graphics to Render a Living Mars

Kevin M. Gill '11G*
Senior Software Engineer, Thunderhead.com, Manchester, NH

Keywords: Computer Graphics, Mars, Life, Planetary Science, OpenGL

Abstract

This article describes the software, algorithms & decisions that went into the development of the Living Mars image project. This includes topics related to computer graphics, software development, astronomy, & planetary science. The purpose of the project was to create a visualization of the planet Mars as could look with a living biosphere. This makes no distinction as to whether this biosphere would represent an ancient or future, possibly terraformed planet.

1 Background

Mars, named for the Roman god of war. Ancient civilizations have forever associated the planet with fear, war, and destruction. It is the color of blood, and "one of a handful of planets visible to the naked eye, and the only one of marked color, so the planet demanded attention (Pyle, 2012)." Ever since man has noticed it, there have been dreams and visions of life on Mars, from Giovanni Schiaparelli and Percival Lowell describing channels and canals to Robert A. Heinlein's science fiction. Lowell, in particular famous for fantastic writings of Mars, asked "are physical forces alone at work there, or has evolution begotten something more complex, something not unakin to what we know on Earth as life?" (Lowell, 1895) Even more recent discoveries by NASA's Curiosity rover have found proof that liquid water once flowed billions of years ago positing an environment that could have served host to life (Brown, 2013).

The planet lies in the so called "Goldilocks Zone" were solar radiation from the sun is at a point to make liquid water possible. That zone is not the only factor affecting the possibly of water, however. The Martian atmosphere today is only about a tenth of Earth's, and with the lower pressure, any liquid water can only exist for a moment before boiling and evaporating into the air. Conversely, on Venus, also in the Goldilocks Zone, the atmosphere is a crushing nine hundred pounds per square inch and runaway carbon dioxide greenhouse gasses make for an extremely hot environment.

There *is* evidence of water on Mars, even today in trace amounts. Findings of neutral PH clays have pointed to likely ancient stream beds and planetary scientists have also found ancient dry rivers cut into the terrain. With the evidence of flowing water, people's imaginations have been running wild, not the least of which was mine.

2 Introduction

Late in 2012, I downloaded the dataset created by the Mars Orbiter Laser Altimeter (MOLA) instrument that is aboard the Mars Global Surveyor (MGS) spacecraft. This instrument was tasked with mapping the Martian surface from orbit and send home altimetry data which can then be used to generate elevation models. It is this data that is used to generate a lot of the computer graphics simulations of the planet seen in print and television.

ISSN 1559-9388 (online version), ISSN 1559-9396 (CD-ROM version).

I had been using this data to test an open-source digital elevation modeling application that I have been writing called jDem846, or Java Digital Elevation Modeler, 8/4/2006 (my son's birthday). Testing with this data allowed me to make comparisons and fine tune the rendering when creating visualization of different planets, namely Earth and Mars, and to make sure the relative terrain features are scaled correctly given the different size, flattening, etc, of the planets. It also allowed me to create interesting scientifically backed visualizations that I would be able to share out on social media and other places.

I was watching a documentary on Mars one weekend afternoon with my son and it struck me to not show Mars how it is, but rather how it could have been or could one day be. What if instead of red, what if it was blue, like our own? It would have received a much different legacy and name than those it enjoys today. Perhaps instead of Mars, it would have been called Neptune, the Roman god of water and seas, long before the planet that currently bears the name would have been discovered.

I knew how to use my software to render a planet and some basic clouds, but had not yet created hand drawn textures (I am no artist by any stretch of the imagination) nor had I done atmospheres. Both were beyond the scope of what I had been writing the application to do. This project would require more time and changes to my code base than spent actually putting together the model of Mars.

In planning the model, I determined what I would need to go into the model. I started with the MOLA altimetry data which allowed me correctly represent the terrain features of Mars. Then was to use that to create a texture map on which I could paint a virtual biosphere, oceans, deserts, and all the features one finds on a habitable planet like ours. I would need a way to render global clouds in a visually appealing way. If one has clouds, one likely has an atmosphere which means light scattering, a halo, and the appearance of gradual thickening (or opacity) of the gasses when viewed from smaller angles.

2.1 About the Software

jDem846 is an open-source software project that provides an interface for creating digital elevation models for GIS (Geographic Information Systems) and scientific consumption. It provides a highly configurable and extensible set of rendering and visualization algorithms in an effort to support a broad range of geographic applications. In addition, a rich API is exposed via a scripting interface which allows users to inject logic and extend models beyond the out-of-box functionality of the program. With a focus on highly detailed models and management of massive input datasets, it is not optimized for high performance rendering in terms of execution time. Rendering is done using an off-screen buffer (onscreen rendering is not yet supported, but possible), primarily due to the desire to support a broad range of client applications ranging from the desktop to clusters and server-side processes. It does not seek to provide competition to other professional GIS packages such as Google Earth or NASA WorldWind. It specializes in the creation of single-frame images; however rendering for video

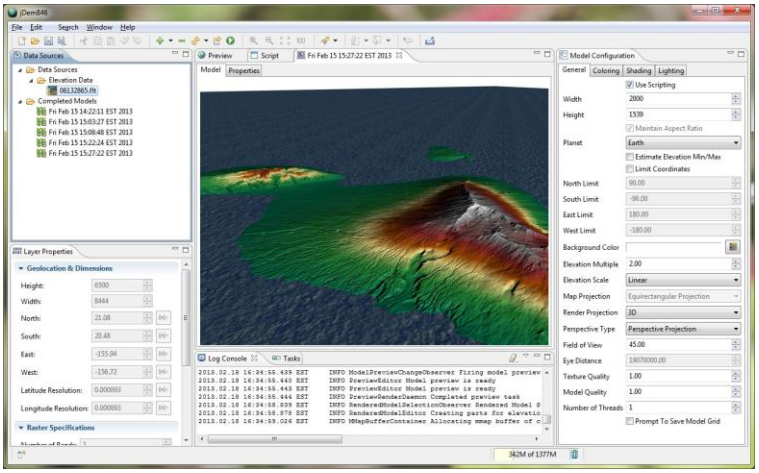


Figure 1: jDem846 Screenshot

consumption has been achieved. Scripting is available using Groovy, Scala, or JavaScript, though the two former options are slated for removal.

The software is written in Java and uses the Eclipse Rich Client Platform for the user interface. By design, the core library is stand-alone to the user interface and can be integrated into other client application frameworks. Originally started as a project at Rivier University's Master's level Professional Seminar course when it was written in C & C++, it has gone through a number of ports, rewrites and other changes. Models are rendered using OpenGL for hardware acceleration, but a software rendering engine is available for systems where OpenGL may not be available.

A major consideration within the core library is the management of massive data sources. It is not uncommon to be required to generate models with input datasets that exceed tens or hundreds of gigabytes which can very quickly overwhelm available memory for both the Java Virtual Machine (JVM) and the computer. Thus the software provides mechanisms for the efficient retrieval and caching of data in a way that will not result in excessive CPU overhead and expensive disk IO.

The software currently supports several input formats in varying degrees of completeness. A basic raster data infrastructure was created for supporting a high number of custom formats and is currently the base of the two existing raster formats that are supported.

Raster Data Formats:

- GridFloat; Gridded 32 bit IEEE floating point values
- BIL16Int; Gridded 16 bit signed integers
- Custom defined rasters using specifications:
    - Width
    - Height
    - Number of bands
    - Header Size (bytes)
    - Data Type (# bytes)
        - Byte (1)
        - Unsigned Short Integer (2)
        - Signed Short Integer (2)
        - Unsigned Integer (4)
        - Signed Integer (4)
        - 32 Bit IEEE Floating Point (4)
        - 64 Bit IEEE Floating Point Double Precision (8)
        - C-Type Signed Short Integer (2)
        - C-Type Signed Integer (4)
        - C-Type 32 Bit IEEE Floating Point (4)
        - C-Type 64 Bit IEEE Floating Point Double Precision (8)
    - Byte Order
        - Little Endian
        - Big Endian
    - Coordinates

        - North, South, East, West
    - Grid Resolution
        - Latitude, Longitude
    - No-Data Value Constant

Image Formats:

- JPEG
- PNG

Shape Formats:

- ESRI ShapeFile

2.2 Scope

On a large rotating object such as Mars and Earth, ocean level is set by a balance of gravity and centrifugal forces. According to scientists (Fraczek, 2010), the seas are controlled first by gravity, which pulls the water towards the center of the planet, and by centrifugal forces generated by the rotation of the planet as it moves along its orbit. These centrifugal forces work to pull the water out and towards the plane of rotation. If we had no centrifugal forces working against gravity (if the planet did not rotate) the oceans would all flow towards the poles. This would happen due to the true shape of Earth and Mars: They are not perfect spheres; they are oblate spheroids. Being an oblate spheroid, there is a certain degree of polar flattening and as a result of that flattening the distance from the surface of the planet to the center is shorter at the poles than it is on the equator. Representing the oblate nature of the planet would not be difficult, but determining the effect of the equilibrium between gravity and centrifugal forces would be. For this reason, I chose to keep the shorelines where they fell based on the data alone.

On a planet with flowing water, winds or any number of geological processes, the surface will experience a certain amount of erosion and surface renewal. As it is, Mars can have high winds and massive dust storms which grind away at objects on the surface. Mars does lack one major factor of surface renewal process experienced here on Earth: Plate tectonics. Mars is geologically dead. This is why an object such as Olympus Mons, comparable as a shield volcano to Mauna Kea in Hawaii, is so much larger. Through its history, Olympus Mons has only been in one place on the planet and has had only one place in which to deposit its lava. In contrast, due to plate tectonics, Mauna Kea has been dragged from near Russia and has left a long line of islands and sea mounts making the island that it is today smaller.

Additionally, with a much thinner atmosphere, objects that would otherwise burn up or explode in Earth's atmosphere, get a relatively unimpeded path to the Martian surface. Though the number of impacts has been greatly reduced in the billions of years since the late heavy bombardment, they still do occur. This means changes to landscapes that would otherwise be smoother with newer lava flows. With a more substantial atmosphere, thus less ground impacting meteorites, an ancient Mars would appear to have a smoother surface.

So, given the complexity of determining erosion and the effect of differing geological processes, I chose to accept the data as is and not do any smoothing of the terrain or try to simulate sediment

deposits. This decision results in a lot more visible creators than would be likely in that environment, but they do help in orienting the viewer and making objects and locations recognizable.

3 Creation of the Main Texture Image

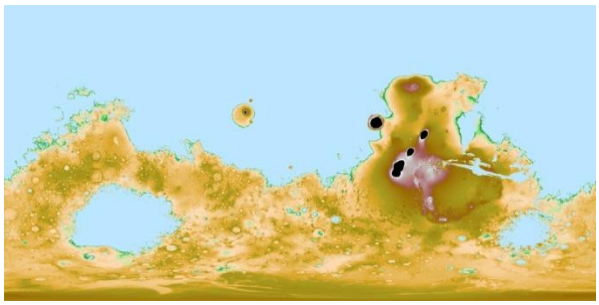


Figure 2: Initial elevation model of the Martian surface

The first thing I would need to create is a flat two-dimensional image on which I could paint the oceans and biosphere. This didn't need to be fancy, just enough to give me an idea of the terrain while I put in the new colors so I could figure out what types of flora would be required. Creating this took the form of a simple flat digital elevation model with a combination hypsometric and bathymetric gradient. It was as this stage that I set the sea levels and determine the location of shorelines. I wanted to oceans to relatively large, just high enough to put shorelines up to the cliffs that edge Olympus Mons, and to flood, but not completely fill or cover the Valles Marineris. This put the base sea level at about 2000 meters. I scripted the initial model to apply a single elevation to the areas under sea level (see Code Sample 1). This would provide a single, smooth color which I would use to guide the coloration of dry land. The initial coloring I used was a hypsometric gradient based on the one used in the ETOPO1 visualizations of Earth created by J. Varner and E. Lim for the National Oceanic and Atmospheric Administration (NOAA). This type of false coloring method is commonly used as a visualization of relative elevations by assigning different colors depending on where a point falls. It is similar to topographical maps but instead of lines to display contour, colors are used to display elevations. This is why some Martian features appear black on the initial image: They are taller than anything on Earth thus lacked a corresponding color gradient. I rendered this at 10,000 by 5,000 pixels so I could support high-resolution versions of the final model. Memory problems affecting my software at the time prevented me from creating it at any higher resolution.

```
/**
 *
 * @param latitude
 * @param longitude
 * @param elevation
 * @returns The elevation, script modified or not.
 */
function onGetElevationAfter(latitude, longitude, elevation)
{
    var scaledSeaLevel = scriptContext.scaleElevation(2000);
    if (elevation <= scaledSeaLevel) {
        elevation = scaledSeaLevel;
    }
    return elevation
}
```

Code Sample 1: Scripted elevation modifier. Scaling allows the method to automatically handle elevation exaggerations configured by the user when generating the model.

Now was the time to create a living biosphere. The image I had created didn't look very appealing, and would only suggest a planet with oceans and otherwise sandy, desert-like land masses. It needed some realistic looking forests, deserts, highlands, lowlands, tundra, etc. I wanted the final product to look as much like an actual photograph as I could, so I chose to use NASA's Blue Marble Next Generation images of Earth (see Figure 3) as a source of those colors and patterns.

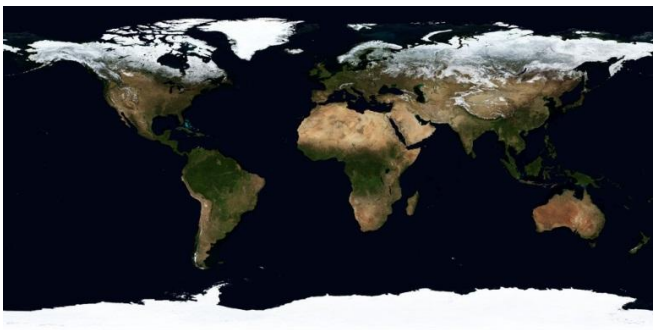


Figure 3: Blue Marble NG. ***Courtesy of NASA Earth Observatory***

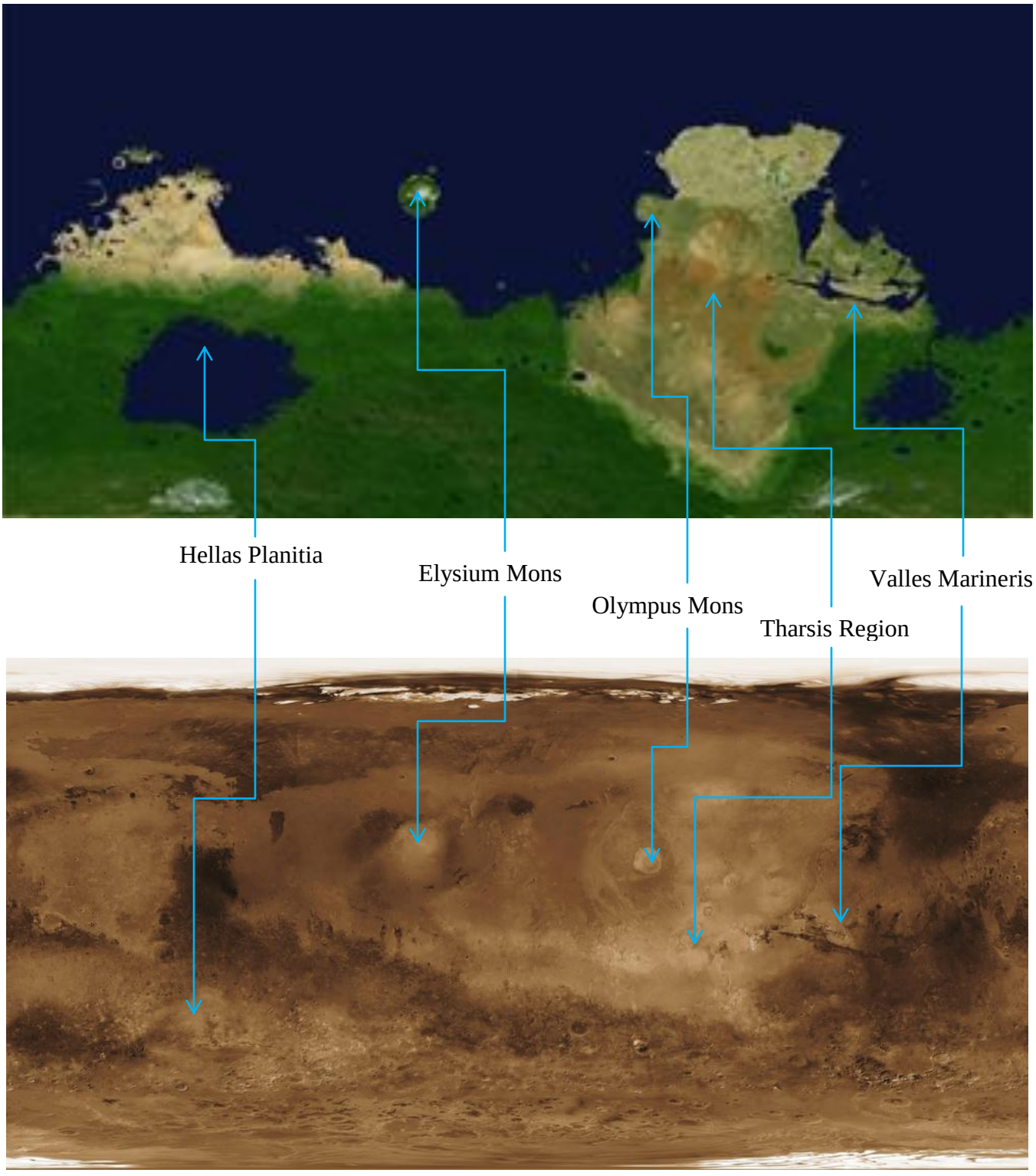
As soon as you look at the surface of Earth, you get an insight into how absolutely diverse an environment we have, and that's just what's visible above the oceans. This is the type of thing that I needed to represent on Mars.

I would start with some initial assumptions:

- It would portray a summertime Mars with less ice caps visible. The current polar ice caps on Mars are a "mix of a thin layer carbon dioxide ice, or dry ice, and water ice. (Pyle, 2012)". The CO_2 portion of the caps in a warmer climate would be unable to form.
- Much of the Tharsis Region rises so high as to effectively poke out of the atmosphere. This is likely too high for much ice to form, thus no glaciation. Also, with the volcanic activity and the bulging of the region, I suspect the ground temperature would too high and the terrain too young to form much ice. Also, being newer volcanic soil, it would probably be rich in nutrients, but too new and/or active for much forestation to take hold.
- Like Earth, forest woodlands would get darker as they get closer to the poles. They would also be increasingly broken up by tundra areas.

I would also pick out areas of Earth that best represented their particular form of biota:

- For rain forests, I would use those found in Brazil in South America and the Congo in Africa.
- For desert, I would use areas from the Sahara in Africa and much of Australia.
- For the transition from desert to woodland, I would use areas of the Sub-Saharan Africa, mostly an area of semi-arid tropical savanna known as the Sahel (Contributors, 2013).
- Temperate woodlands would be mostly derived from the eastern United States and Europe.
- Arctic woodland, glaciers, glaciated mountain peaks, and tundra would mostly be derived from northern Siberia in Russia with some filler from northern Canada.
- I would attempt shallow tropical seas by using areas surrounding The Bahamas, Cuba and Florida, but found them difficult to place them in a realistic manner.


Hellas Planitia
Elysium Mons
Olympus Mons
Tharsis Region
Valles Marineris

4 Creating Clouds

Clouds can be a very challenging aspect of any computer graphics rendering. They don't generally follow any easily replicable pattern and they are highly dependent on atmospheric conditions and terrain properties. As such, I chose to take a bit of a shortcut here. I pulled the global clouds composite image (Figure 4) from the NASA Blue Marble NG project. It was white on black, but using GIMP to transform the black into a transparency was not difficult. It is a very useful image and it doesn't make it too obvious that it's Earth based by not making too many surface properties evident (the exception being overcast clouds surrounding the Andes in South America).

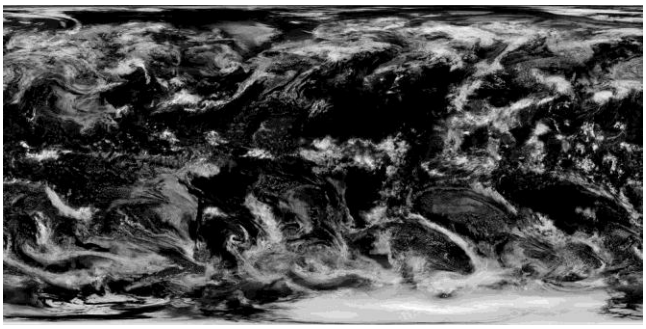


Figure 4: Blue Marble NG. ***Courtesy of NASA Earth Observatory***

The clouds image was strictly two dimensional; however, I sought to add a visible textured and three dimensional look to them rather than just blanket them over the Martian surface as-is. To establish a textured look, I would need to put them at different altitudes, just as they are in real life and apply lighting to ensure the sides facing away from the sun were shadowed. Additionally, I wanted to ensure any terrain visible below the clouds would be in their shadow.

Figure 5: The type of cloud look I was looking for. Note the fluffy look and the shadows below them. ***Courtesy of Chris Hadfield, CSA***

So, to achieve a three dimensional look with a two dimensional image, I took the approach that I would derive cloud altitude from their relative transparency. One useful by-product of the conversion from the original white on black image from Blue Marble was a gradient of transparency within the clouds: The thicker a cloud, the whiter and opaque it is, and vice versa. I picked a high and low range for cloud elevations of 10,000 meters to 18,000 meters, based mostly on visual appearance than meteorological reality. Because the color values are coming from an image file stored in a 4 byte Red, Green Blue, Alpha (RGBA) format, the transparency of any given point would be within the range of 0 to 255. Determining the altitude of a point would simply be a function of the alpha value within the RGBA range converted to the altitude range (Code Sample 1). Using this method, clouds assumed a rather round, puffy look. With OpenGL lighting applied, they took on a realistic look. The only drawback was a limit of the available resolution of the source cloud image. With the image I used, the clouds looked realistic when viewing the planet as a whole. Zoom in to a particular region of feature, and they looked like strange, unattractive blobs.

```
// Cloud layer configuration
var cloudLayer = {
    cloudImagePath : "F:/bluemarble/cloud_combined_8192.b.png",
    altitudeMinimum : 10000.0,
```

```
    altitudeMaximum : 18000.0,
    resetImage : false,
    coords : {
        north : 90.0,
        south : -90.0,
        east : 180.0,
        west : -180.0
    }
};


function initialize()
{
    ...
    // Determines the location of the sun as a vector with X/Y/Z
    // coordinates.
    sunsource = scriptContext.getSolarPosition()

    // jDem has a key/value pair storage container available for scripts
    // which allows users to load resources into memory once and store them
    // for retrieval on the next execution. This can save a considerable
    // amount of time when performing preview renders which are
    // expected to complete in a relatively small amount of time.
    // Resources are available to future executions of the script until
    // either it is removed or the application is shut down.
    // This fetch method accepts the resource identifier key, a function
    // which is called to load the resource if it is not yet present
    // in the container, and a boolean that if “true” will force the
    // container to call the load callback regardless of whether the
    // resource exists in memory.
    cloudsGeoImage = scriptContext.storage.get(cloudLayer.cloudImagePath, function()
    {
        var img = scriptContext.loadImage(cloudLayer.cloudImagePath
                        , cloudLayer.coords.north
                        , cloudLayer.coords.south
                        , cloudLayer.coords.east
                        , cloudLayer.coords.west
                        , true);
        return img;
    }, cloudLayer.resetImage);

    // Scales the lower and upper altitudes of the cloud layers.
    cloudLayer.altitudeMinimum = scriptContext.scaleElevation(8000.0);
    cloudLayer.altitudeMaximum = scriptContext.scaleElevation(9000.0);
    ...
}

function getCloudAlpha(latitude, longitude)
{
    var color = cloudsGeoImage.getColor(latitude, longitude);
    var alpha = (color != null) ? color.getAlpha() : 0;
    return alpha;
}
```

```
function getCloudAltitude(latitude, longitude, altitudeMinimum, altitudeMaximum)
{
   var alpha = getCloudAlpha(latitude, longitude);
   return altitudeMinimum
                          + ((alpha / 255.0)
                                * (altitudeMaximum - altitudeMinimum));
}
```

Code Sample 2: Determining the altitude at one point in the cloud and rendering.

```
function renderCloudImage(cloudsGeoImage)
{
    var lightSpecular = 0.1;
    var lightDiffuse = 2.6;
    var lightEmmissive = 0.0;
    var lightAmbient = 0.0;
    var lightShininess = 50;

    renderer.setLighting(sunsource
                         , lightEmmissive
                         , lightAmbient
                         , lightDiffuse
                         , lightSpecular
                         , lightShininess);

    var emmisive = lightingOptionModel.getEmmisive();
    var ambient = lightingOptionModel.getAmbient();
    var diffuse = lightingOptionModel.getDiffuse();
    var specular = lightingOptionModel.getSpecular();
    var shininess = lightingOptionModel.getSpotExponent();

    renderer.setMaterial(new Color(emmisive, emmisive, emmisive, 1.0)
                               , new Color(ambient, ambient, ambient, 1.0)
                               , new Color(diffuse, diffuse, diffuse, 1.0)
                               , new Color(specular, specular, specular, 1.0)
                               , shininess);

    var texture = cloudsGeoImage.getAsTexture();

    var cloudLatitudeResolution =
        scriptContext.modelDimensions.getModelLatitudeResolution();
    var cloudLongitudeResolution =
        scriptContext.modelDimensions.getModelLongitudeResolution();

    // If the model resolution is more detailed then the cloud image, adjust
    // use the cloud image resolution. This will save time and unnessecary
    // processing.
    if (cloudLatitudeResolution <
        cloudsGeoImage.getImageDefinition().getLatitudeResolution()) {
       cloudLatitudeResolution =
        cloudsGeoImage.getImageDefinition().getLatitudeResolution();
    }
```

```
    if (cloudLongitudeResolution <
        cloudsGeoImage.getImageDefinition().getLongitudeResolution()) {
       cloudLongitudeResolution =
        cloudsGeoImage.getImageDefinition().getLongitudeResolution();
    }

    // The TextureRenderer class takes in a texture and related properties
    // and handles the actual render operation. As a last parameter, it
    // optionally accepts a callback function used to retrieve the elevation
    // at each point. In this case, the function retrieves the cloud altitude
    // passes that back as the elevation. The texture renderer, none the wiser
    // that it is rendering clouds rather than a surface terrain, will place
    // clouds at the altitudes computed in the getCloudAltitude() method.
    var cloudRenderer = new TextureRenderer(texture
              , renderer
              , view
              , cloudLatitudeResolution
              , cloudLongitudeResolution
              , scriptContext.globalOptionModel
              , new TextureMapConfiguration(true
                   , TextureMapConfiguration.InterpolationTypeEnum.LINEAR
                   , TextureMapConfiguration.TextureWrapTypeEnum.REPEAT)
                   , function(latitude, longitude) {
                     return getCloudAltitude(latitude
                                           , longitude
                                           , cloudLayer.altitudeMinimum
                                           , cloudLayer.altitudeMaximum);
                   });

    cloudRenderer.render();
}
```

Code Sample 3: Rendering the cloud layer

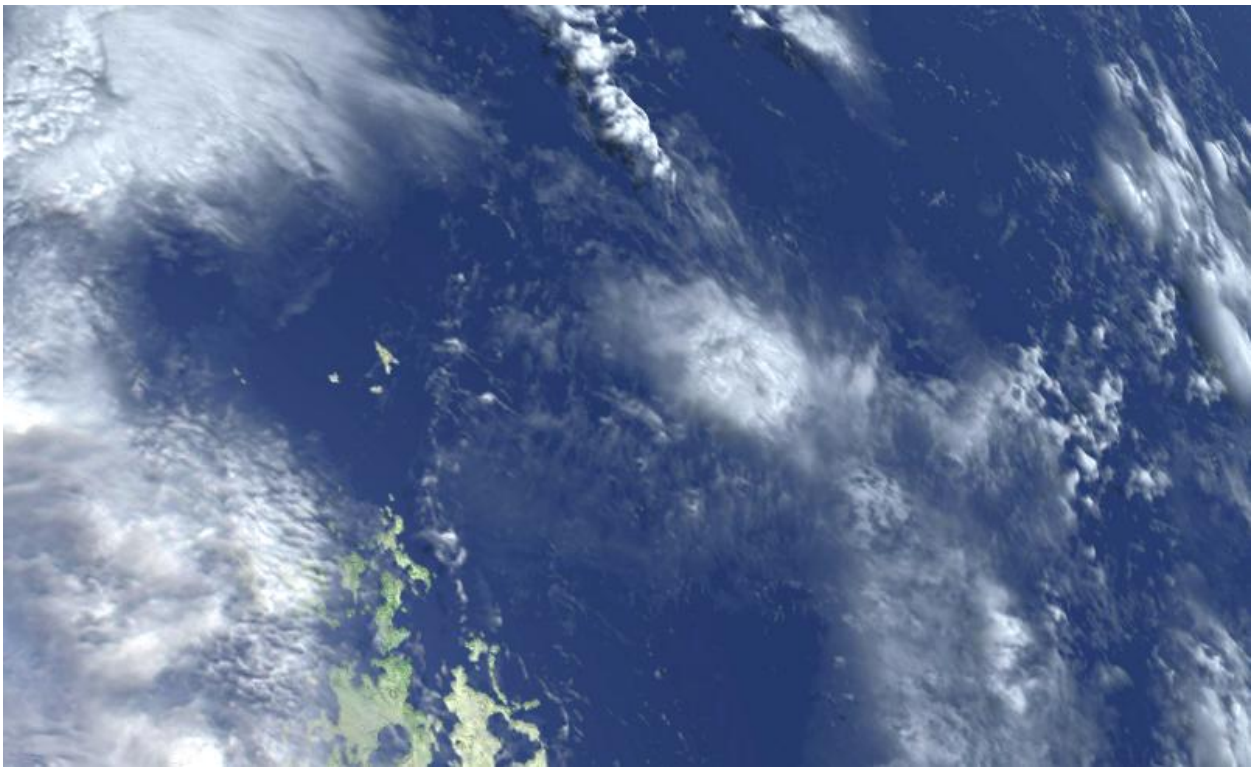


Figure 6: Martian clouds rendered as I had envisioned them. In actuality, in the source image, these clouds are just southeast of Taiwan.

5 Creating an Atmosphere

Putting an atmosphere on Mars took some trial and error to get right. To achieve the effect I wanted, I broke the atmosphere into two different components: the halo surrounding the visible rim (horizon) of the planet (what I called the upper atmosphere) and the varying degrees of opaqueness when viewed against the surface of the planet (what I called the lower atmosphere).

The lower atmosphere portion follows a pretty basic concept: The closer the eye is to the surface, the clearer the atmosphere is. Additionally, the less atmosphere between the viewer and the surface there is, the clearer it is. This is apparent when a distance is viewed from the top of a mountain: Objects closer are clearly visible, were objects far in the distance are partially obscured by our atmosphere, both by light scattering and the water vapors present in the air.

Figure 7: Atmosphere over Earth. ***Courtesy of Chris Hadfield, CSA***

My first attempt at rendering this lower atmosphere consisted of rendering a semitransparent sphere around the planet. It was the bluish color I had chosen for the Martian atmosphere, but as the surface pointed towards the user, I made it more and more transparent. Though visually appealing by itself, it had a habit of producing blending issues in OpenGL when done alongside the clouds and upper atmosphere.

The second attempt at the lower atmosphere took a simpler approach: OpenGL fog effect. In OpenGL, you are able to place a fog within a frame by defining its color along with some properties. Basically, when OpenGL renders a pixel, it blends the fog color with the true pixel color based on the calculation you specify. A basic linear method specifies start and stop points and blends in the fog color linearly as the distance from the views increases away. I used the EXP2 method which blends in the fog color given the following formula (the EXP method is the same just without the exponent being raised by 2):

$$f = e^{-(density*z)^2}$$

To handle density, I found a working value and made sure to scale it properly when the model was rendered at different sizes.

```
function preRender(renderer, view)
{
    var density = 0.0015;
    var w = scriptContext.globalOptionModel.getWidth();
    var d = density * (390.0 / w);

    var forColor = new Color(ColorUtil.rgbaToInt(104, 138, 176, 255));
    renderer.enableFog(forColor, FogModeEnum.EXP2, d, 1, 10000);
}
```

Code Sample 4: Configuring fog prior to the start of the render process.

The upper atmosphere provides a completely different technique. It had to mimic looking through an atmosphere from the side while fading as it got to higher altitudes. Doing this with fog or a semi-transparent sphere wouldn't work since they can't handle the fading in the higher altitudes.

The method I would end up going with was to draw a ring around the planet. In a method known as "billboarding," the ring would be drawn facing directly at the viewer and placed with no real regard as to the rotations of the planet which it surrounded or the angle of the viewer's perspective.

Since the fade of the halo is not a direct, linear fade, I did it in two stages. The first, lower ring would be a solid color. The second, higher ring would provide the fade from the base atmospheric color to full transparency. Rendering the rings by themselves is not much of a challenge, however placing them is, and this is where I took a bit of a shortcut.

When you look at a globe, depending on your distance from the surface, you don't see the whole picture. One would think you'd see the entire half of the sphere, but the closer to the sphere you are, the closer the horizon is to you and the less of the sphere you can actually see before the surface drops out of view (Figure 9: Portion of sphere visible to a viewer. If you render the halo at the exact

Figure 8: The atmospheric halo. ***Courtesy of Chris Hadfield, CSA***

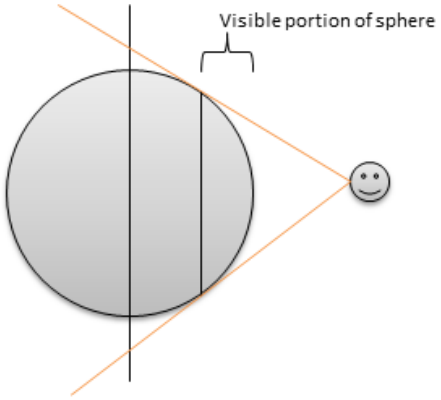


Figure 9: Portion of sphere visible to a viewer

midpoint of the sphere, then it won't be visible, so it is at this horizon that the halo needs to be rendered.

The horizon can be determined using the following equations. They are a close-enough method and assume a completely spherical planet, which is generally acceptable given the small degree of flattening.

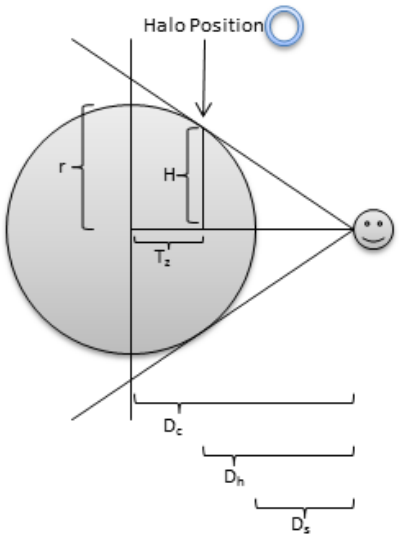


Figure 10: Halo positioning dimensions

The halo positioning dimensions are defined by the following parameters:

Radius: r
Eye distance to surface of planet: D_s
Eye distance to center of planet: $D_c = D_s + r$
Eye distance to horizon (horizontally):

$$D_h = \sqrt{D_c \cdot (2r + D_c)}$$

Translation Amount from Origin: $T_z = D_c - D_h$

We will also need to scale the halo some to adjust for the fact that the halo's radius will be smaller because it exists on a part of the planet closer than the center:

Horizon Height from Horizontal Plane:

$$H = \sqrt{r^2 - T_z}$$

Scale Amount: $S_{xyz} = H/r$

Since the planet is being rendered at the origin coordinates (0, 0, 0), the value for the distance of the horizon from the center of the planet can be used within a translation matrix for the Z axis. This will move the OpenGL matrix to the horizon were the halo will be rendered.

```
var lowerBandAtmosphere = {
   elevationMinimum : -5000,
   elevationMaximum : 25000,
   colorLower : new Color(ColorUtil.rgbaToInt(104, 138, 176, 255)),
   colorUpper : new Color(ColorUtil.rgbaToInt(104, 138, 176, 255)),
```

```
    enableMaterial : true,
    emissive : new Color(ColorUtil.rgbaToInt(0, 0, 0, 255)),
    ambient : new Color(ColorUtil.rgbaToInt(15, 22, 30, 255)),
    diffuse : new Color(ColorUtil.rgbaToInt(104, 138, 176, 255)),
    specular : new Color(ColorUtil.rgbaToInt(104, 138, 176, 255)),
    shininess : 1.0,
    enabled : true
};

var upperBandAtmosphere = {
    elevationMinimum : 25000,
    elevationMaximum : 52000,
    colorLower : new Color(ColorUtil.rgbaToInt(104, 138, 176, 255)),
    colorUpper : new Color(ColorUtil.rgbaToInt(104, 138, 176, 0)),
    enableMaterial : true,
    emissive : new Color(ColorUtil.rgbaToInt(0, 0, 0, 255)),
    ambient : new Color(ColorUtil.rgbaToInt(15, 22, 30, 255)),
    diffuse : new Color(ColorUtil.rgbaToInt(104, 138, 176, 255)),
    specular : new Color(ColorUtil.rgbaToInt(104, 138, 176, 255)),
    shininess : 1.0,
    enabled : true
};

function initialize()
{
    lightingOptionModel = scriptContext.getLightingOptionModel();
    sunsource = scriptContext.getSolarPosition();

    ...
    lowerBandAtmosphere.elevationMinimum = scriptContext.scaleElevation(-5000);
    lowerBandAtmosphere.elevationMaximum = scriptContext.scaleElevation(25000);
    upperBandAtmosphere.elevationMinimum = scriptContext.scaleElevation(25000);
    upperBandAtmosphere.elevationMaximum = scriptContext.scaleElevation(52000);
    ...

}

// Called from the postRender handler
function renderAtmosphereHalos()
{
    if (lowerBandAtmosphere.enabled) {
        renderAtmosphereHalo(lowerBandAtmosphere);
    }

    if (upperBandAtmosphere.enabled) {
        renderAtmosphereHalo(upperBandAtmosphere);
    }
}


function renderAtmosphereHalo(atmosphere)
{
    var viewAngle = scriptContext.globalOptionModel.getViewAngle();
```

```
var eyeZ = view.eyeZ();
var radius = view.radius();

var fieldOfView = scriptContext.globalOptionModel.getFieldOfView();
var width = scriptContext.globalOptionModel.getWidth();
var height = scriptContext.globalOptionModel.getHeight();
var aspectRatio = width / height;
var nearClipDistance = 0.1;
var farClipDistance = 100000;
// Push matrices onto the projection and modelview stacks so
// we can make modifications and not effect any other
// render operations done after this.
renderer.matrixMode(MatrixModeEnum.PROJECTION);
renderer.pushMatrix();
renderer.loadIdentity();

renderer.perspective(fieldOfView
            , aspectRatio
            , nearClipDistance
            , farClipDistance);

renderer.matrixMode(MatrixModeEnum.MODELVIEW);
renderer.pushMatrix();
renderer.loadIdentity();

renderer.lookAt(0     // Eye X
      , 0    // Eye Y
      , eyeZ    // Eye Z
      , 0    // Center X
      , 0    // Center Y
      , 0    // Center Z
      , 0    // Up X
      , 1    // Up Y
      , 0);     // Up Z

renderer.translate(viewAngle.getShiftX() * radius
          , viewAngle.getShiftY() * radius
          , viewAngle.getShiftZ() * radius);

var distanceToSurface = view.nearClipDistance();
var distanceToHorizon = view.farClipDistance();
var distanceToCenter = view.getZoomDistanceFromCenter();
var trans = distanceToCenter - distanceToHorizon;
var horizonHeightFromPlane = MathExt.sqrt(MathExt.sqr(radius);
                                    - MathExt.sqr(trans));
var distanceToSurfaceHorizon = MathExt.sqrt(MathExt.sqr(distanceToHorizon)
                                  + MathExt.sqr(horizonHeightFromPlane));

var scale = horizonHeightFromPlane / radius;
```

```
    renderer.setLighting(sunsource
                , atmosphere.emissive
                , atmosphere.ambient
                , atmosphere.diffuse
                , atmosphere.specular
                , atmosphere.shininess);
    renderer.setMaterial(atmosphere.emissive
                        , atmosphere.ambient
                        , atmosphere.diffuse
                        , atmosphere.specular
                        , atmosphere.shininess);
    renderer.disableMaterial();

    renderer.translate(0.0, 0.0, trans);
    renderer.rotate(90.0, AxisEnum.X_AXIS);

    renderer.begin(PrimitiveModeEnum.TRIANGLE_STRIP)
    for (var angle = 0.0; angle <= 360.0; angle+=1.0) {
        var latitude = 0.0;
        renderAtmosphereVertex(latitude
                    , angle
                    , atmosphere.elevationMinimum
                    , atmosphere.colorLower
, viewAngle);

        renderAtmosphereVertex(latitude
                    , angle
                    , atmosphere.elevationMaximum
                    , atmosphere.colorUpper
       , viewAngle);
    }
    renderer.end();

    // Pop the temporary projection and modelview matrices
    // back off of the stack.
    renderer.matrixMode(MatrixModeEnum.PROJECTION);
    renderer.popMatrix();

    renderer.matrixMode(MatrixModeEnum.MODELVIEW);
    renderer.popMatrix();
}

var atmosphereVector = new Vector();
var atmosphereNormal = new Vector();
function renderAtmosphereVertex(latitude, longitude, elevation, color, viewAngle)
{
    // Fetch the X/Y/Z coordinates from the View object
    view.project(latitude
             , longitude
             , elevation
             , atmosphereVector);

    view.getNormal(latitude
```

```
                , longitude
                , atmosphereNormal
                , false);

        // Rotate normal vector to match the model as rotated by the renderer.
        Vectors.rotate(viewAngle.getRotateX()
           , viewAngle.getRotateY()
           , viewAngle.getRotateZ()
           , atmosphereNormal);

        // Since the normal is being determined as if it wrapped around the equator,
        // we again rotate it to match it's vertical nature.
        Vectors.rotate(90, 0, 0, atmosphereNormal);

        // Set the OpenGL lighting surface normal
        renderer.normal(atmosphereNormal);

        // Set the OpenGL color
        renderer.color(color);

        renderer.vertex(atmosphereVector.x
                , atmosphereVector.y
                , atmosphereVector.z);

    }
```

Code Sample 5: Rendering the upper and lower band atmospheric halo.

6 Per-Vertex Lighting Adjustments

When modeling an environment using computer graphics is becomes clear that not every surface uses the same lighting properties. For this reason, there are a lot of options available for modifying lighting in terms of sources, colors, materials, shininess, and more. With a single planet, sourcing the light is relatively straight forward: Determine the location of the sun in relation to the planet and define that at the light source. The complexity, however, is a property of the surface features.

The model needed to be mindful of a couple things:

- Bodies of water are shiny
- Landmasses, for the most part, are more matte
- Clouds project shadows
- The clouds themselves and the atmosphere have their own lighting considerations.

For any terrain above sea level, I modified the material to a more matte lighting and left anything below that remain shiny. This was a modification of the spot exponent, which controls shininess, and the brightness of the specular light component.

A check was also made at each terrain point as to whether they are under a cloud cover. If they are, I reduced the diffuse and specular lighting relative to the transparency of the cloud cover. That alone results in most of the planet being in shadow since the cloud layer has clouds at just about every point at one level or another. To address this, I set a minimum transparency amount where any cloud level below it would not cause the terrain to be shadowed.

```
function onBeforeVertex(latitude, longitude, elevation, renderer, view)
{
       var scaledSeaLevel = scriptContext.scaleElevation(2000);
    var emmisive = lightingOptionModel.getEmmisive();
    var ambient = lightingOptionModel.getAmbient();
    var diffuse = lightingOptionModel.getDiffuse();
    var specular = lightingOptionModel.getSpecular();
    var shininess = lightingOptionModel.getSpotExponent();
    var minAlpha = 30.0;

    if (elevation > scaledSeaLevel) {
        shininess = 10;
        specular *= 0.5;
    }

    var alpha = getCloudAlpha(latitude, longitude);
    var cloudAlphaAdjust = (1.0 - ((alpha - minAlpha) / (255.0 - minAlpha)));

    if (alpha > minAlpha) {
        specular *= cloudAlphaAdjust;
        diffuse *= cloudAlphaAdjust;
    }

    renderer.setMaterial(new Color(emmisive, emmisive, emmisive, 1.0)
                         , new Color(ambient, ambient, ambient, 1.0)
                         , new Color(diffuse, diffuse, diffuse, 1.0)
                         , new Color(specular, specular, specular, 1.0)
                         , shininess);
}
```

Code Sample 6: Light adjustments prior to each point being rendered.

7 Tying it All Together

The final model used Mars as the target planet for rendering and Earth for time-based calculations. I also limited the coordinates to avoid some unnecessary rendering time. Elevations were multiplied by 5 and the viewer is positioned 10,070 kilometers away from the surface of the planet. To improve rendering time, both for previewing and for final rendering, I used jDem846 to first process all terrain elevation and colors (excludes any clouds or lighting) into a grid file. This grid file can be used to replace the raw input data and image layers so as to allow the render process to skip raster processing.

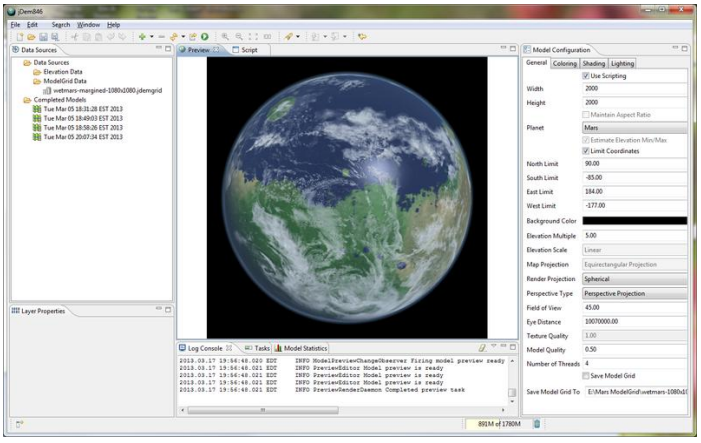


Figure 11: jDem846 with the Living Mars project. The pre-processed model grid file is loaded instead of the raw data and image layer.

8 Conclusion

I posted the first image in November of 2012 on to Google+. That was followed up with version two, posted to the Space community of Google+ in January, 2013. With the visibility afforded to posts to the social network, the images quickly went viral across the Internet. They found their way across websites such as Universe Today, Discovery News, The Atlantic, the Huffington Post, the Smithsonian blog, and Fox News. I attributed some of that to some lucky timing as there was a growing interest in Mars with the recent arrival of the Curiosity rover and discoveries of water evidence from the planet.

In March, 2013, I posted an animation to YouTube titled "A Day in the Life of a Living Mars" that showed the planet rotating so as to display all sides. The original images took several hours to render, but with the move to OpenGL and a number of optimizations I was able to get the render time down to around eighty seconds. The shorter render time allowed me to render the number of frames needed to put together the animation, which totaled about 1440, one for each minute of the day. I also did some post-render color adjustments using Adobe Lightroom 4. This animation didn't get the attention that the original images received, which I expected, but it didn't do too badly.

I was very pleased how the images came out in the end. I put in a lot of revisions since the first image was posted up on Google+, not the least of them being the move from a custom software-based renderer to the use of OpenGL. I will probably keep making revisions to the Living Mars project, and most certainly will continue development on jDem846. Both are available free on the Internet to anyone interested.

9 Images

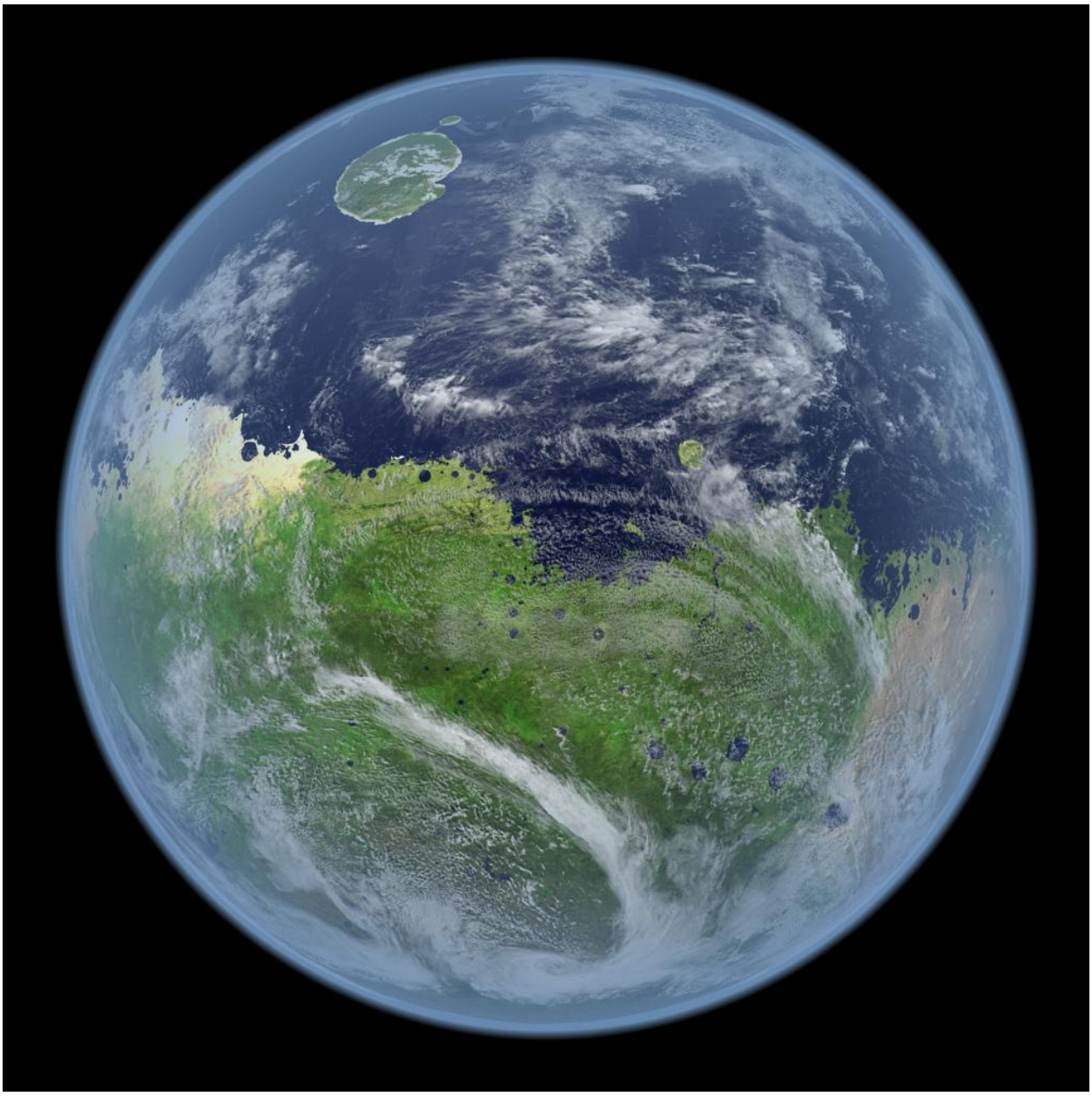


Figure 12: The original "Living Mars" image. Posted November 8, 2012.

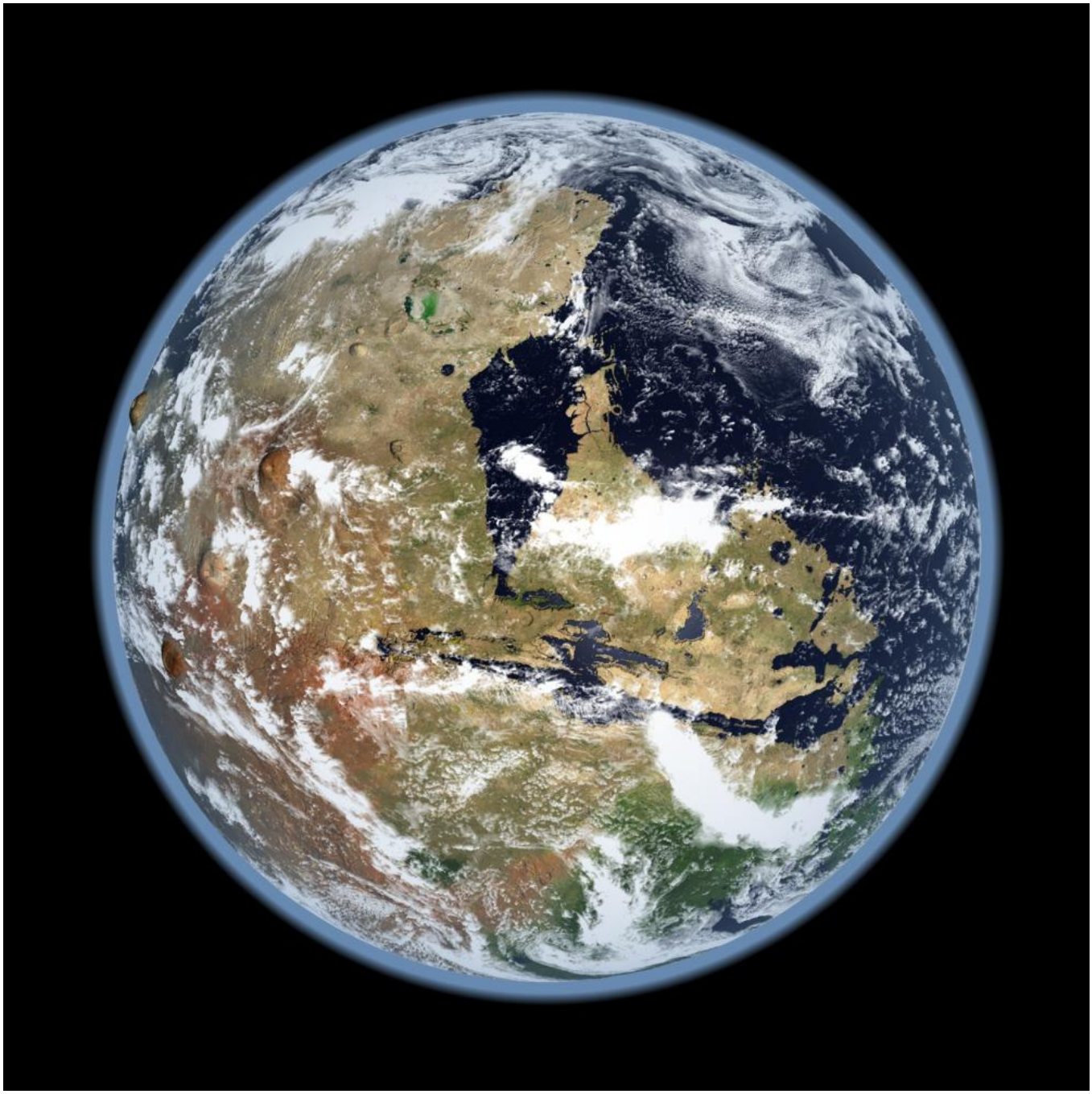


Figure 13: Version two displaying Valles Marineris and the Tharsis Region. I had reduced the opaqueness of the atmosphere, boosted the terrain elevation exaggeration and adjusted the cloud lighting. Posted on the Internet on January 2nd, 2013.

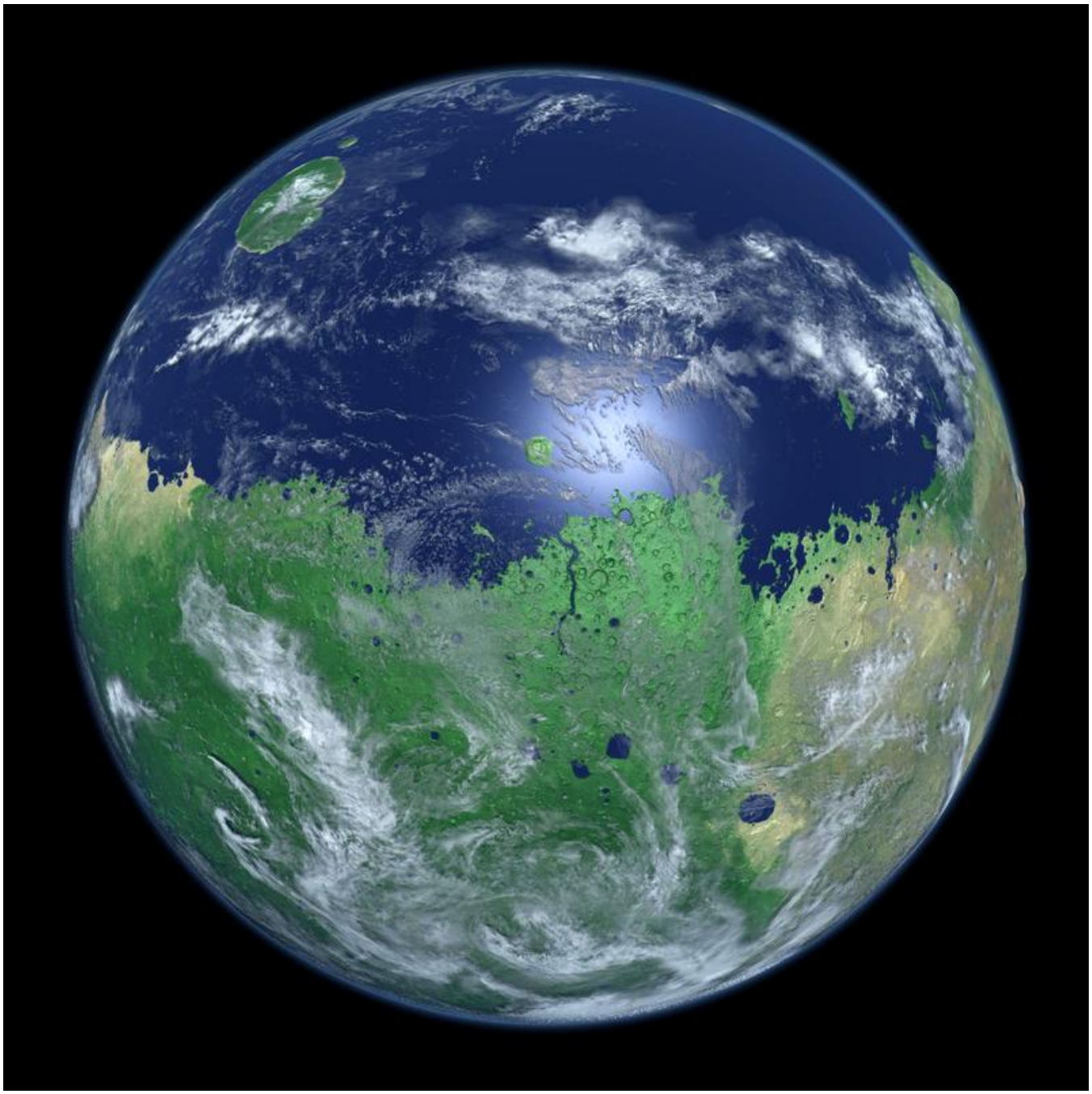


Figure 14: Version 3 returns to the original perspective to display the improvements of the OpenGL renderer. Other changes included a reduced atmospheric halo, improved clouds, and sunlight reflection off of the ocean. Posted on the Internet on February 20th, 2013.

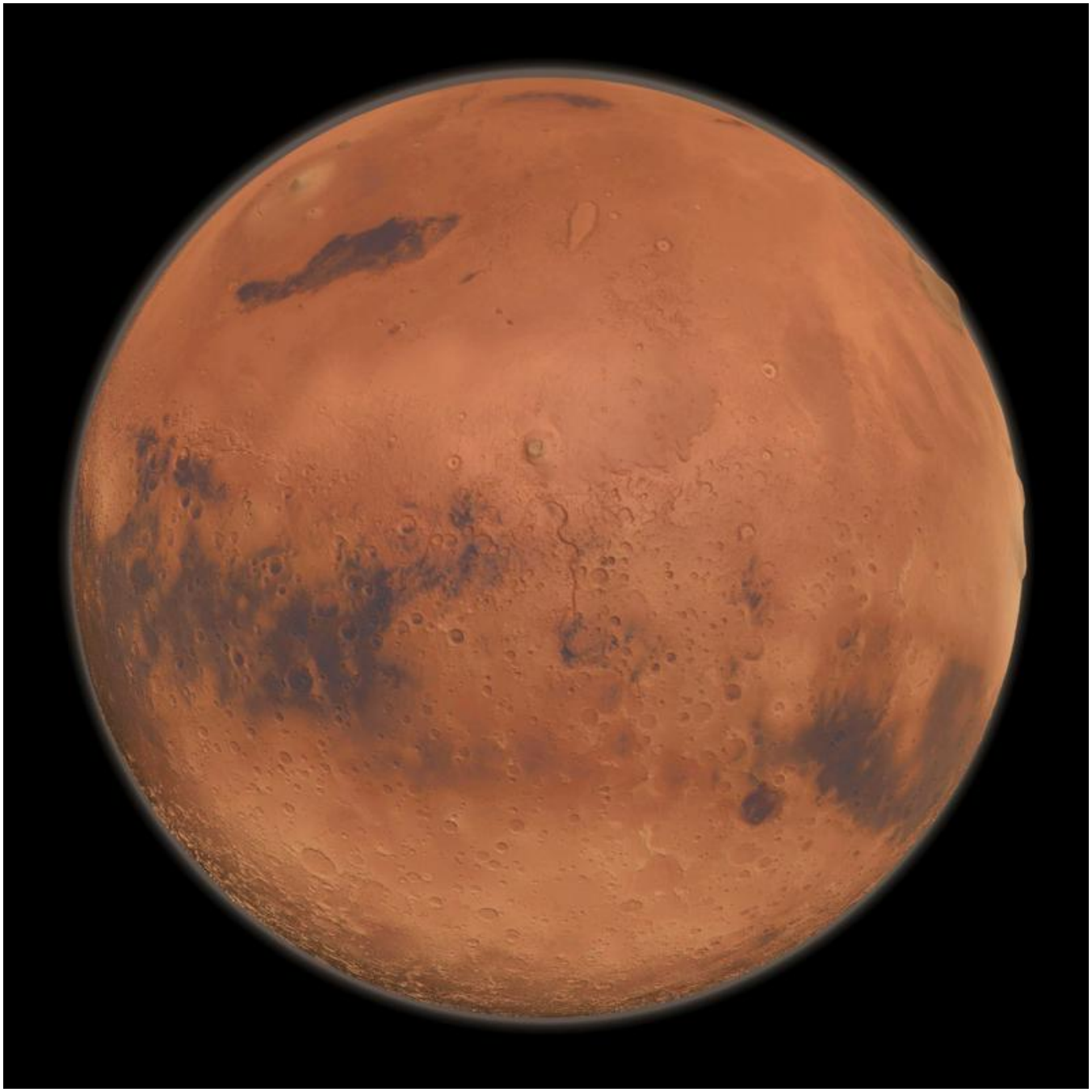


Figure 15: True-color model of Mars using the same perspective as Living Mars versions 1 & 3. Posted on the Internet on February 20th, 2013.

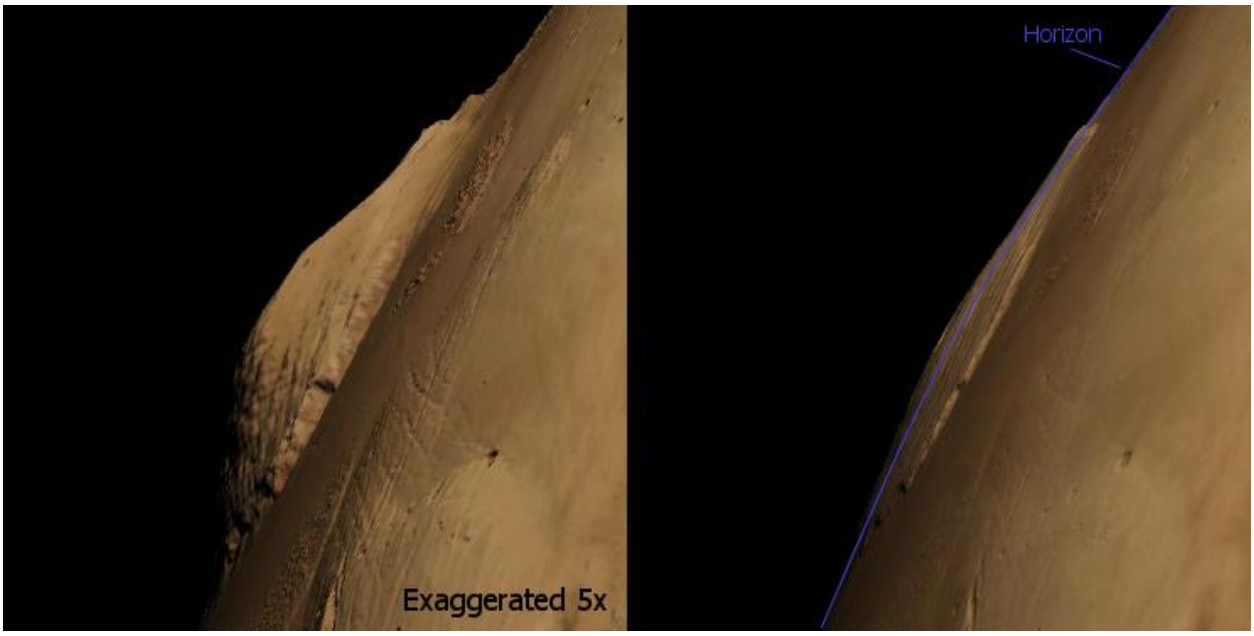


Figure 16: In this graphic I put a 5x exaggerated rendering of O. Mons for detail (left) and a true scale rendering (right). You are seeing the volcano from its eastern side with it on the horizon, with that visible horizon traced in blue. Note the peak of O. Mons visibly poking up above that line. This was created using the data and algorithms developed for the "Living Mars" project. Posted on the Internet on March 12th, 2013.

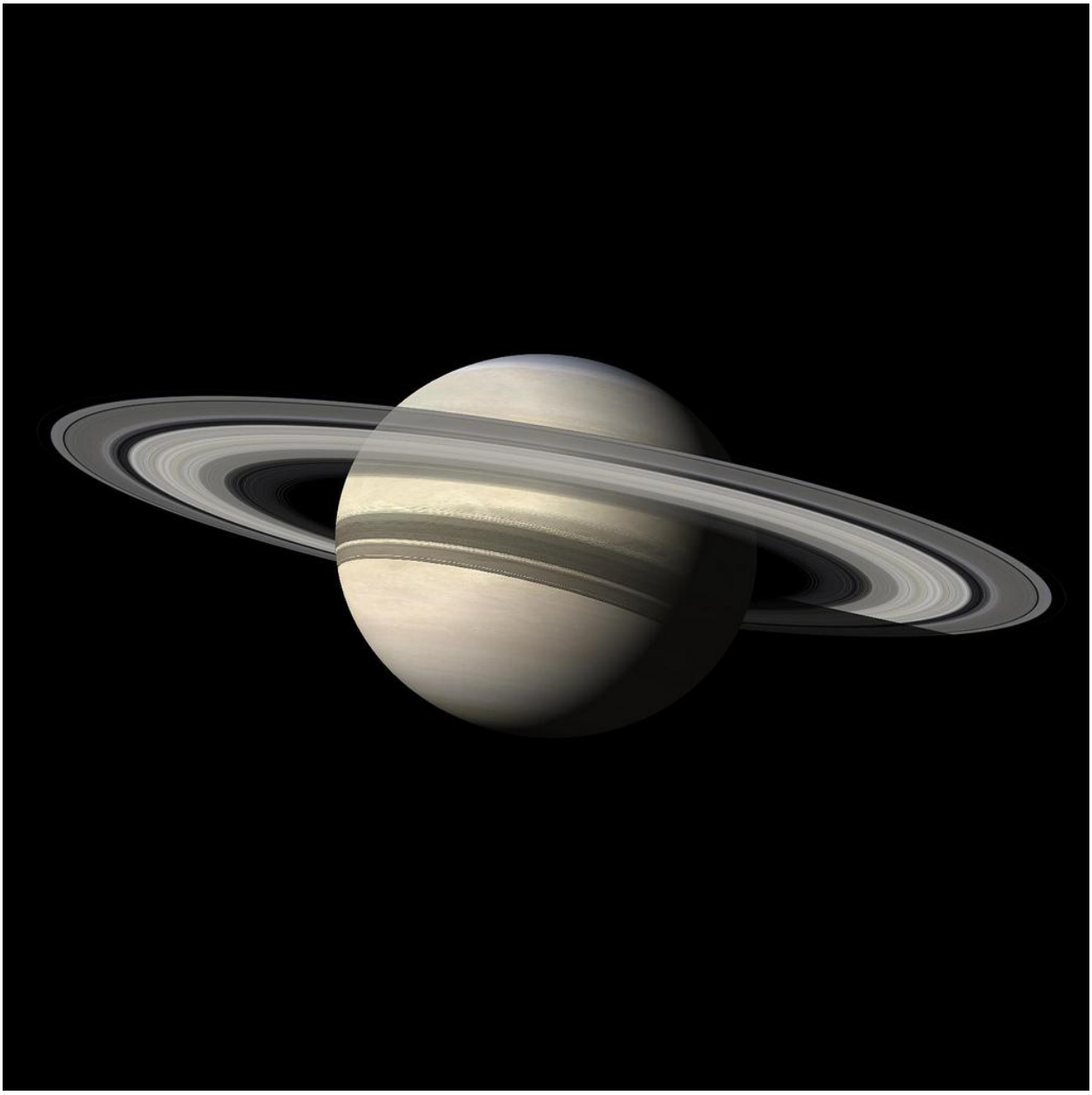


Figure 17: Saturn rendered using jDem846. This was done do utilize the visualization of planet flattening, shadowing, and rings (via the atmospheric halo) originally developed for Living Mars. Posted on the Internet on February 7th, 2013.

10 Bibliography

Brown, D. (2013, 3 12). *NASA Rover Finds Conditions Once Suited for Ancient Life on Mars*. Jet Propulsion Laboratory, California Institute of Technology. Retrieved March 12, 2013, from http://www.jpl.nasa.gov/news/news.php?release=2013-092

Contributors. (2013, 3 2). *Sahel*. Wikipedia, the free online Encyclopedia. Retrieved March 15, 2013, from http://en.wikipedia.org/wiki/Sahel

Encyclopedia Britannica. (2013). *Canals of Mars*. Encyclopedia Britannica Online. Retrieved March 10, 2013, from http://www.britannica.com/EBchecked/topic/366397/canals-of-Mars

Fraczek, W. (2010, Summer). *If the Earth Stood Still*. ArcUser Online. Retrieved March 10, 2013, from http://www.esri.com/news/arcuser/0610/nospin.html

Lowell, P. (1895). *Mars.* Boston, MA: Houghton and Mifflin.

Pyle, R. (2012). *Destination Mars: New Explorations of the Red Planet.* Amherst, NY: Prometheus.

11 General Resources

jDem846 Project Homepage. Retrieved March 10, 2013, from https://code.google.com/p/jdem846/

Blue Marble Next Generation, NASA Earth Observatory. Retrieved March 10, 2013, from http://earthobservatory.nasa.gov/Features/BlueMarble/

Jet Propulsion Laboratory, California Institute of Technology. Retrieved March 10, 2013, from http://www.jpl.nasa.gov/

Living Mars Project Resources. Retrieved March 10, 2013, from https://docs.google.com/folder/d/0B5gEWNRhmLMTYkEyR3FQY2pjRVk/edit?usp=sharing

12 Related Articles

A Day in the Life of a Living Mars. Retrieved March 10, 2013, from http://www.youtube.com/watch?v=7RSY37H_sFU

New Images Show a "Living" Mars. Universe Today. Retrieved March 10, 2013, from http://www.universetoday.com/99248/new-images-show-a-living-mars/

Living Mars: Kevin Gill's Concept Images Show Red Planet Teeming with Life. Huffington Post. Retrieved March 10, 2013, from http://www.huffingtonpost.co.uk/2013/01/04/living-mars-kevin-gills-concept-images_n_2407216.html

Blue, Not Red: Did Ancient Mars Look Like This? Discovery News. Retrieved March 10, 2013, from http://news.discovery.com/space/blue-not-red-did-ancient-mars-look-like-this-130102.htm

A Martian Dream: Here's What the Red Planet Would Look Like With Earth-Like Oceans and Life. The Atlantic. Retrieved March 10, 2013, from http://www.theatlantic.com/technology/archive/2013/01/a-martian-dream-heres-what-the-red-planet-would-look-like-with-earth-like-oceans-and-life/266791/

This is What a Watery Mars May Have Looked Like. Smart News, Smithsonian Magazine. Retrieved March 10, 2013, from http://blogs.smithsonianmag.com/smartnews/2013/01/this-is-what-a-watery-mars-may-have-looked-like/

View of a Living Mars Take the Rouge Off. CNet. Retrieved March 10, 2013, from http://news.cnet.com/8301-17938_105-57562214-1/views-of-a-living-mars-take-the-rouge-off/

A Day in the Life of a Living Mars. Universe Today. Retrieved March 10, 2013, from http://www.universetoday.com/100635/a-day-in-the-life-of-a-living-mars/

* **KEVIN M. GILL** is a Senior Software Engineer at Thunderhead.com. Prior to that he was a software engineer with Fidelity Investments, Inc. and a computer technician in the U.S. Marine Corps. He is a 2011 graduate of Rivier College with a M.S. in Computer Science and a 2009 graduate of the University of Massachusetts Lowell with a B.S. in Information Technology. He resides in Nashua, NH with his wife, Stephanie, his 6 year old son and 1 year old twins. He currently maintains the open source GIS modeling project, jDem846, and can be reached on Google+ or Twitter.