Global Terrain Technology for Flight Simulation

Adam Szofran

Microsoft ACES Game Studio
One Microsoft Way, Redmond, WA 98052

1. Introduction

Near the end of 2006, the ACES game studio at Microsoft will release the tenth version of the long-running Flight Simulator franchise. The technology behind Flight Simulator has evolved drastically over the 23 year history of the product [1] and it continues to evolve today, but the goal has always remained the same: To give the customer the most realistic world possible in which to live out their aviation dreams. Naturally, a huge part of the Flight Simulator world is the Earth itself and all of its varied terrain including mountains, lakes, rivers, forests, cities, roads and other features visible from the windows of the virtual aircraft. This paper explains some of the techniques Microsoft uses to represent the Earth's terrain in Flight Simulator's virtual world.

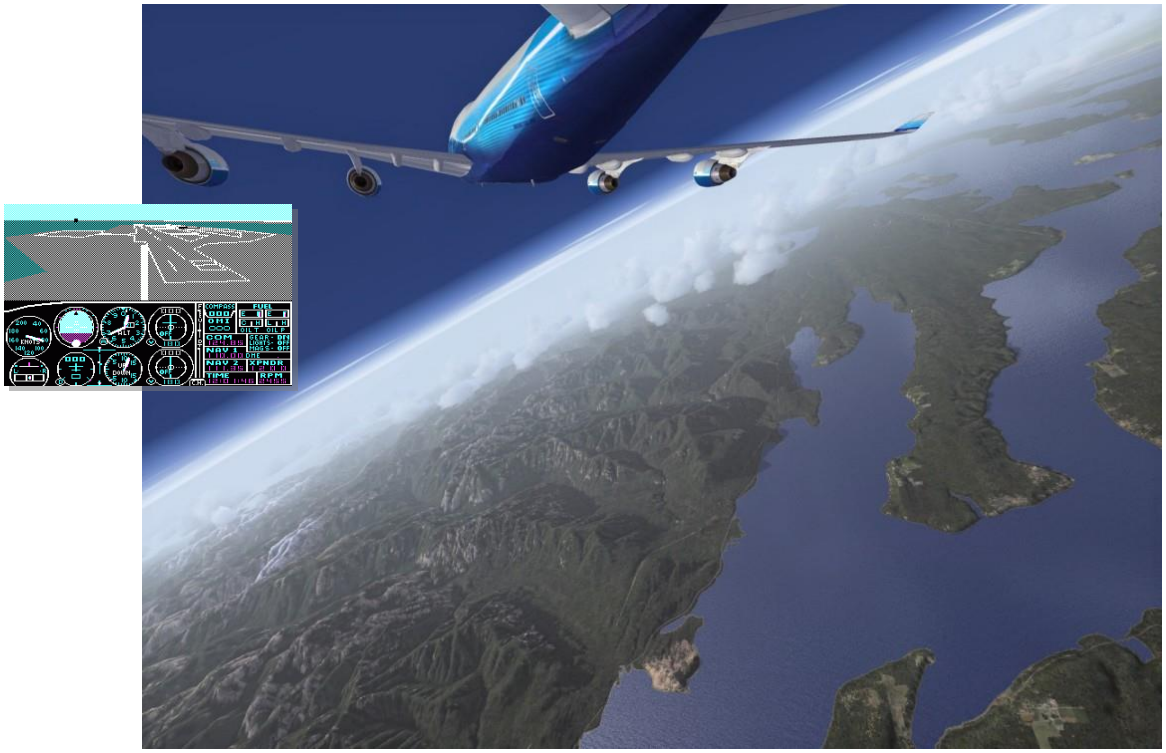

Figure 1: Flight Simulator terrain then (inset c. 1982 [1]) and now (2006)

2. Requirements

The Flight Simulator user community certainly includes the typical 18 to 34 year-old male gamer but it also contains a much wider audience of aviation enthusiasts, male and female, young and old. Because those in this wider audience don't necessarily upgrade their computer

hardware as often as the typical hard-core gamer, Flight Simulator must be highly scalable to run on a wide range of target systems.

The goal of the Flight Simulator terrain engine is to render the Earth from any vantage point whether on the ground, in flight, or in orbit at any time of day in any season. The engine must also support multiple simultaneous viewpoints, rendered in separate windows, while sharing as much data between views as possible. Furthermore, the engine must allow users to extend the scenery database by adding downloadable content.

Because Flight Simulator must do much more than just render the Earth, the terrain engine can't monopolize the CPU resources of the host system. Adequate resources must be reserved for the other aspects of the game such as aircraft and vehicle simulation, artificial intelligence, multiplayer communication, and rendering of the aircraft, buildings, and other non-terrain entities in the scene.

3. Geographic Data Processing

To render the Earth as realistically as possible, the Flight Simulator terrain engine uses a huge amount of real-world geographic data, both vector and raster. Examples of vector data include the centerlines of roads, railways, power lines, streams, and shorelines. It's also used to outline parks, golf courses, and cleared areas used for airports. Raster data consists of elevation grids, aerial imagery, land and water classifications and seasonal data.

The raw data sources themselves total well over a terabyte in size. Before the data can be used with Flight Simulator, it must be combined, culled, and compressed into a form that can fit on the distribution media and that can be used efficiently by the terrain engine at run-time. To avoid being overwhelmed by the amount of data, it's very important to use a good geographic information system (GIS). We use ArcGIS from ESRI [8] which provides excellent functionality out of the box, but also streamlines authoring of custom data manipulation tools through an extensive API layer.

While good tools can automate much of the geographic content authoring process, there are still thorny issues which can only be solved by manually editing the data. For example, vector river data and raster elevation data may not match exactly if they were produced by different vendors at different resolutions. If the mismatch is too visually jarring, like when a river climbs a mountain ridge instead of following the valley, a good GIS editing system is indispensable for making spot fixes to the data.

To help fit all of our data on the distribution media, we employ a variety of compression techniques, both lossless and lossy, always choosing the method that gets the best results for a given set of inputs. One method that we've used heavily recently is the Progressive Transform Codec (PTC), a proprietary lossy technique developed by Microsoft Research [10]. PTC works especially well with both aerial imagery and elevation data because it can handle multi-channel integer and floating-point data with a variety of bit depths per channel. It also can achieve very high compression ratios without an objectionable loss of quality. Like JPEG2000 or wavelet compression, the quality is adjustable with lower quality resulting in smaller output. However, the PTC code is more compact than JPEG2000 and also runs

faster. We also use the PTC entropy encoder on the back end of our predictive delta compressor for vector data.

4. Subdividing the surface of the Earth

To help organize and manage the terrain data at run time, we subdivide the surface of the Earth into cells organized into a quad tree. There are several schools of thought concerning how best to do this. One of the simplest is to subdivide the globe along geographic lines of latitude and longitude. However, because the lines of longitude converge at the poles, the scale of the cells varies with latitude. Polyhedral subdivisions of the globe can reduce the scale variation, but at the cost of added complexity [2, 3, 4].

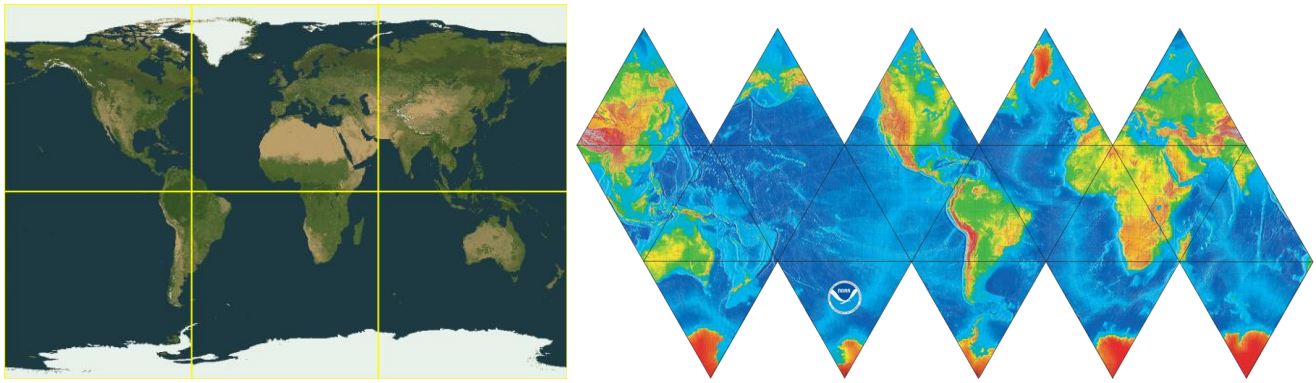

Figure 2: Geographic and polyhedral [6] subdivisions of the globe

The biggest disadvantage of polyhedral subdivision from our perspective is that it's simply too difficult for content producers to author tiling textures that fit the often triangular or parallelogram shaped cells. The geographic subdivision method, in spite of the convergence issues at the poles, simply wins on the basis of simplicity and familiarity. Besides, while Flight Simulator allows trans-polar flight, most of the action still occurs at the middle latitudes where the majority of the world's population is concentrated.

To minimize scale distortion where most of our customers live and fly, we designed our quad tree so the ratio of north-south to east-west extent of the cells is nearly 1:1 at +/- 45 degrees of latitude. This was done by designating six root cells, three north of the equator and three south of it. Each root cell covers 90 degrees of latitude and 120 degrees of longitude.

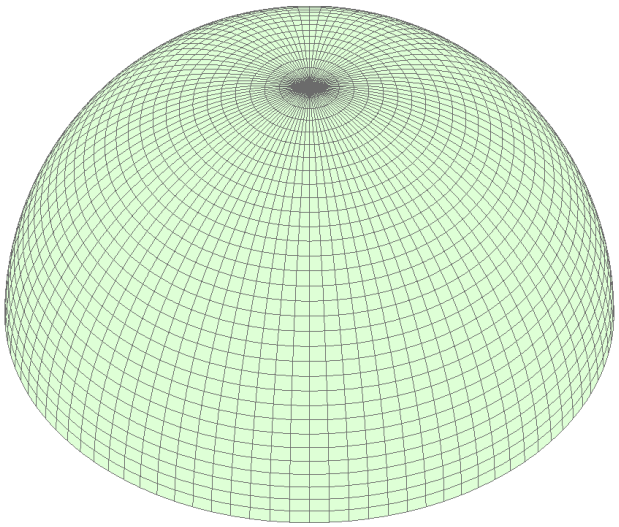

Figure 3: One level of Flight Simulator's global quad tree

Each cell in the quad tree is assigned a unique identifier composed from the cell's depth in the tree and its distance in cells south of the North Pole and east of the 180th meridian. With this scheme, only simple integer operations are required to find the identifier of any cell's parent, siblings, and children. We use the cell identifier as a search key when retrieving the data for each cell from disk and to cache and find cells in hash tables at run time. Explicit links between quad tree cells are only used where necessary for speed but otherwise we save on link storage and maintenance by using the cell identifiers as implicit links. This type of implicit quad tree design is sometimes called a linear quad tree [13].

5. Scene graph

The terrain scene graph is designed for rendering with as much fidelity as possible but without having to load more data than necessary. Therefore, we only want to use the data with the highest level of detail (LOD) near the viewpoint and use progressively less detail as distance from the viewpoint increases. Small cells deep in the tree contain data with the highest LOD so we want to concentrate them near the viewpoint. The larger cells near the root of the tree hold data with low LOD so we want them to be visible from greater distances than the smaller cells. To accomplish this, we cache cells from all levels of the tree in the scene graph but we use a geometrically decreasing geographic radius as the depth of the cells increases. Because the cache radius decreases at about the same rate as cell size, the number of cells cached in each level of the scene graph remains relatively constant at all levels.

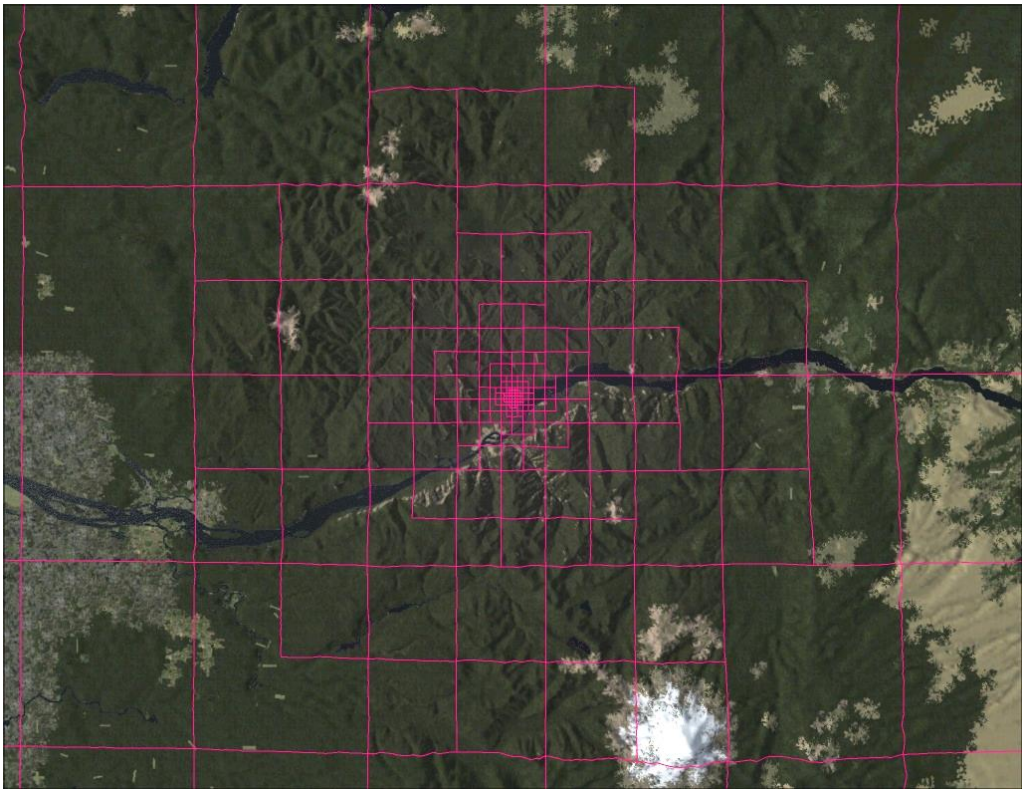

Figure 4: Outlines of quad tree cells in the scene graph superimposed on the terrain

Ideally, the scene graph radius for each level of the quad tree would be determined by screen resolution, with higher resolutions requiring larger radii, but we leave the choice up to the individual user so they can find a balance between view quality and the capabilities of their machine. In practice, we allow a user-selectable radius of between 2.5 and 4.5 cells, which seems to be a good balance between keeping the total number of cells low, while still allowing enough granularity for streaming in new content as the viewpoint moves through the world.

To support the requirement for multiple independent views, we create a separate scene graph for each viewpoint. When the scene graphs for multiple views overlap spatially, they share as much view-independent data as possible such as textures and raw, untriangulated elevation data.

6. Terrain mesh geometry

The latest cutting-edge terrain mesh triangulation and LOD computation algorithms are being designed to run in vertex shaders on 3D hardware [9]. However, since Flight Simulator must run on both older and newer hardware, we run our mesh triangulation algorithms in software using techniques pioneered by Lindstrom [5], Pajarola [7], Duchaineau [11], and others, collectively referred to here as the restricted quad tree triangulation, or RQT. The RQT generates a seamless mesh whose complexity depends upon the roughness of the terrain and is scalable by a simple screen-space error metric. By allowing the end user to set the error metric, Flight Simulator can easily adapt the terrain mesh triangle count to different user preferences and machine configurations.

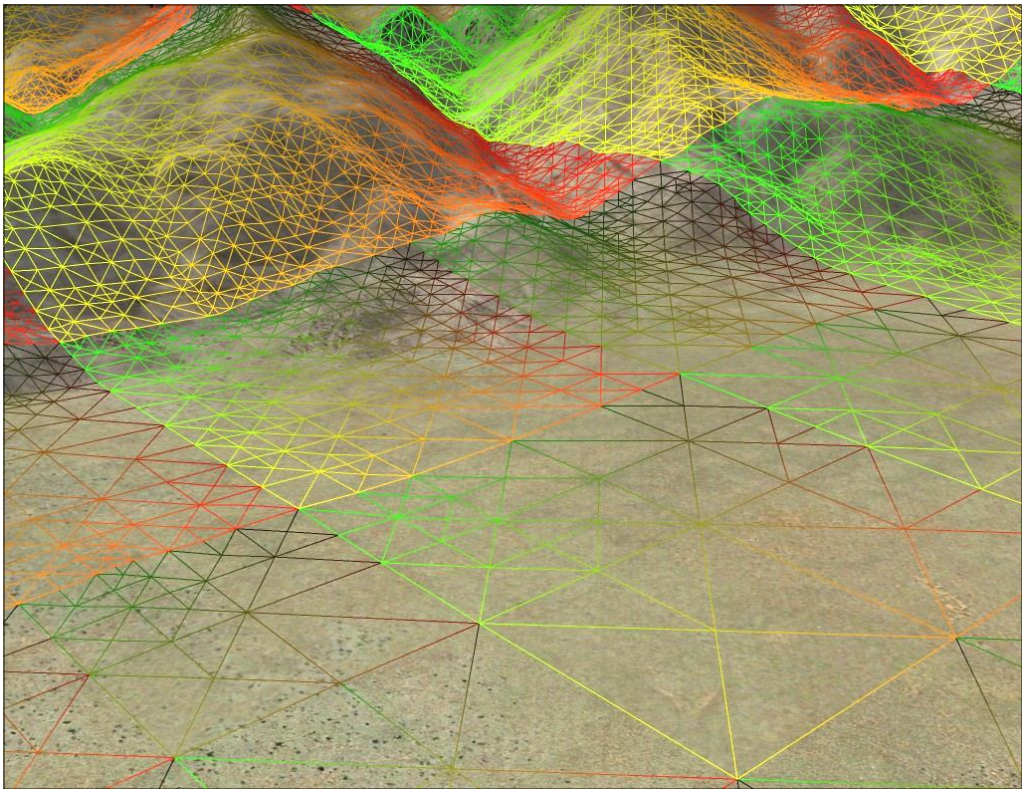

Figure 5: Section of terrain mesh triangulated with the RQT algorithm

The RQT algorithm, as published, is really intended for “flat” Earth terrain where the sea level elevation reference is assumed to be an infinite plane. Since Flight Simulator models the Earth’s curvature, we had to modify the algorithm a bit. The main problem is that the unmodified RQT tends to eliminate nearly all the mesh vertices over large open expanses of ocean which it interprets as being completely flat. The solution is to increase the error metric for each point in the mesh by the height of the Earth’s curvature above the line connecting the point’s opposite neighbors. The result is an increase in the error metric roughly proportional to the distance to its neighbors. The larger the error metric, the less likely a point will be eliminated from the triangulation. This helps preserve the Earth’s curvature in large areas with the same elevation.

To avoid monopolizing CPU resources, the terrain engine does not refine the entire mesh every frame. Instead, the mesh is refined and triangulated incrementally and the resulting geometry is added to the scene graph when it becomes available using a double buffering scheme.

After triangulation, the mesh vertices must be transformed to account for the Earth’s curvature. Flight Simulator uses the ellipsoidal Earth model defined by the World Geodetic System 1984, or WGS 84 [12]. WGS 84 uses a geocentric rectangular coordinate system with the Earth’s center of mass at the origin. The x-, y-, and z-axes are shown in Figure 6. Each axis originates at the Earth's center of mass. The positive x-axis passes through the equator at the prime meridian. The positive y-axis intersects the equator at 90° east longitude. The positive z-axis passes through the North Pole. Distances along each axis are measured in meters.

Longitude, λ, is measured as the angle in the x-y plane from the x-axis toward the y-axis. The geodetic latitude, φ, at a point is the angle between x-y plane and the normal of the reference ellipsoid through that point. Altitude, h, is the height in meters above the ellipsoid surface, which we use as our sea level reference.

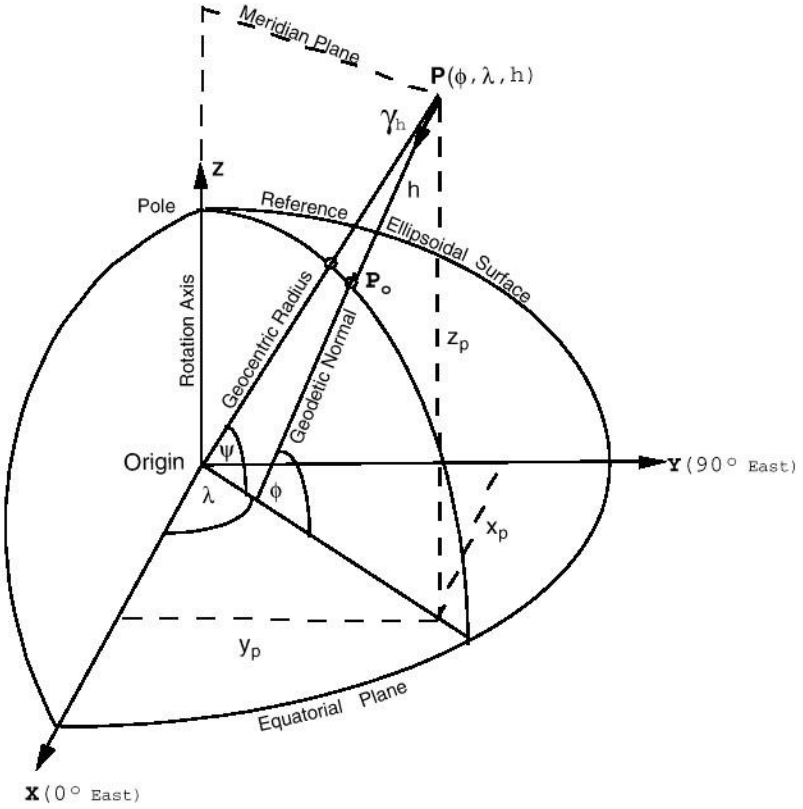

Figure 6: WGS 84 coordinate system [12]

The conversion from latitude, longitude, and altitude above sea-level to geocentric Cartesian coordinates is relatively straightforward using equation 4-14 from the WGS 84 specification:

$$x = (N + h)\cos\phi\cos\lambda$$
$$y = (N + h)\cos\phi\sin\lambda$$
$$z = \left(\left(\frac{b^2}{a^2}\right)N + h\right)\sin\phi$$

Where:

$$N = \frac{a}{\sqrt{1 - e^2\sin^2\phi}}$$

a = 6378137.0 meters (Earth's equatorial radius)
b = 6356752.3142 meters (Earth's polar radius)
e = 8.1819190842622 x 10^{-2} (ellipsoid eccentricity)

The equations must be evaluated using 64-bit floating point to prevent a loss of precision. The seven significant digits of 32-bit floats aren't enough to represent a point with better than sub meter accuracy. Since the rendering pipeline (Direct3D in this case) uses 32-bit floats, the final coordinates used for rendering must be converted to 32-bit offsets from a 64-bit local origin no more than several thousand meters away (see Figure 7). It would be simpler if each cell had its own local origin and world-to-camera matrix, but then small cracks would be visible between cells due to floating-point imprecision. Therefore, all the cells must share the same local origin and matrix.

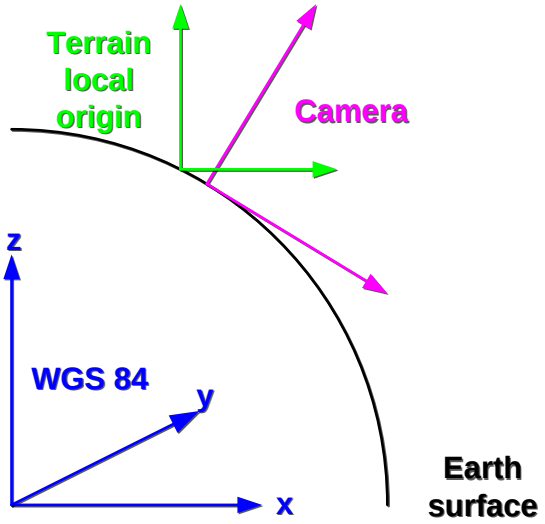

Figure 7: Diagram showing the WGS-84, local origin, and camera axes.

As the viewpoint moves through the world, the local origin must be periodically updated to keep it within a few kilometers of the viewpoint. Otherwise, a loss of precision results making the terrain vertices appear to wiggle around as the viewpoint moves, an interesting but obviously undesirable outcome. Whenever we update the local origin, all the vertex buffers in every cell of the quad tree must be recomputed relative to the new origin. As this can take several frames, the new vertex data is double buffered and then made active only when all are ready to avoid visible cracks.

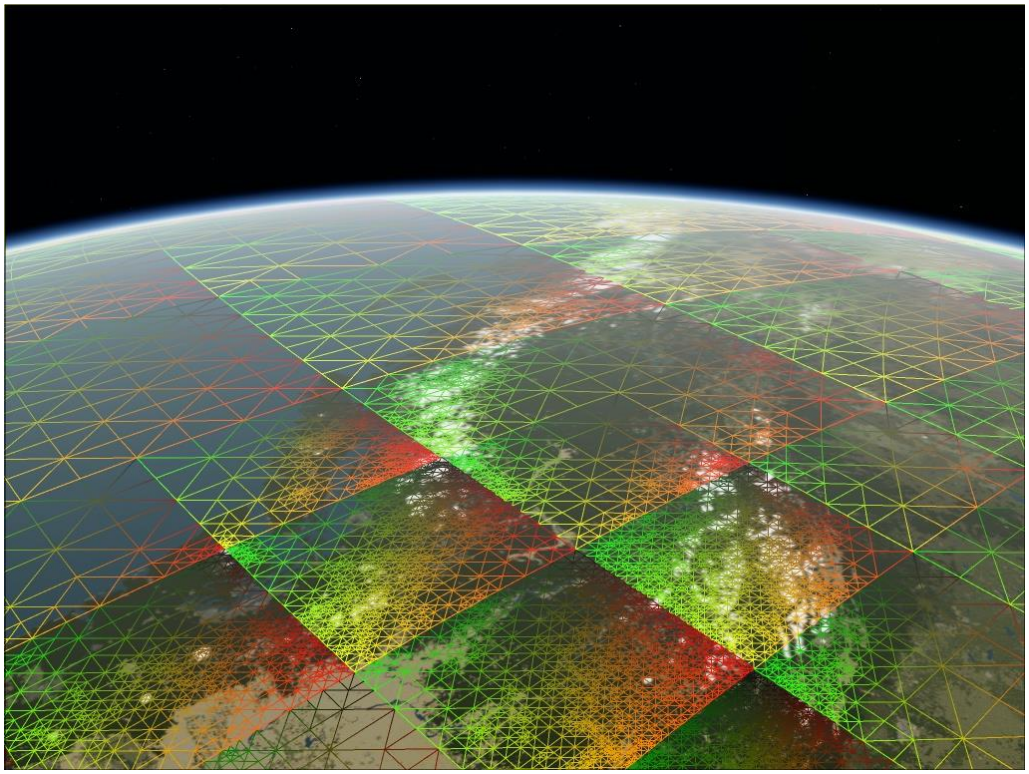

Figure 8: A section of the fully tessellated globe.

7. Synthesis of aerial imagery

Someday, Flight Simulator may stream all of its terrain imagery from a central server such as Microsoft's Virtual Earth (http://ve.msn.com). However, that day hasn't arrived yet for a variety of reasons [20]. For one, Flight Simulator does not require a broadband Internet connection, so we can't rely on a network streaming solution. Second, raw aerial photographs, even high resolution ones, look surprisingly poor when draped over terrain. Some of the problems include contrast and color balance inconsistencies, visible seams between adjacent images, parallax or perspective artifacts caused by tall buildings, cloud obscuration, shadows, lack of seasonal variation, and incomplete coverage. The resources required to correct all of these problems for a worldwide database of imagery are prohibitive.

Our solution to the problems mentioned above is to synthesize aerial imagery on the fly from a set of high quality image tiles representing the most common classifications of land coverage. The determination of which tiles appear in a given area is made by sampling a worldwide map of land and water classification information and seasonal data. To account for cultural and environmental variation around the globe, we subdivide the world into 23 distinct regions with each region having its own set of representative textures for each land class. Blended into the synthesized imagery are roads, rail lines and bodies of water pulled from real-world databases. This approach allows us to cover the terrain with very realistic looking high-resolution imagery at a fraction of the cost and without many of the problems of real aerial imagery.

One of the disadvantages of on-the-fly texture synthesis is that it can take a significant amount of processing, particularly if high quality results are desired. This problem is tempered

somewhat by the fact that once the imagery for the area around the viewpoint is ready, synthesis of new textures is only required when the viewpoint moves or significant changes occur to the local environment such as seasonal changes. Furthermore, we only synthesize higher LOD textures at run time, which in this case are those with a resolution of better than about 300 meters per pixel. The set of lower LOD textures is actually small enough that we can simply synthesize them as a preprocessing step, compress them, and ship them on the distribution media to be loaded as if they were aerial imagery.

Due to the complexity of the synthesis algorithm, we execute it in software rather than using video hardware. This of course requires us to use a fast, high-quality 2D software rasterizer. We chose Anti-Grain Geometry (AGG), by Maxim Shemanarev [14] because it is fast, free, and comes with full source code making it fully customizable to our needs.

The basic synthesis algorithm uses the following steps for each quad tree cell in the scene graph.

1. Build a per-pixel map of land classes in the cell
   a. Blend or mask between different classes within the cell
   b. Use terrain slope to modulate the class map
   c. Apply vector data to the class map
2. Apply each land class texture using the class map as a stencil
3. Render linear vector features
4. Build light maps

The figures that illustrate the steps in the synthesis process were all created for the location outlined in pink in Figure 9.

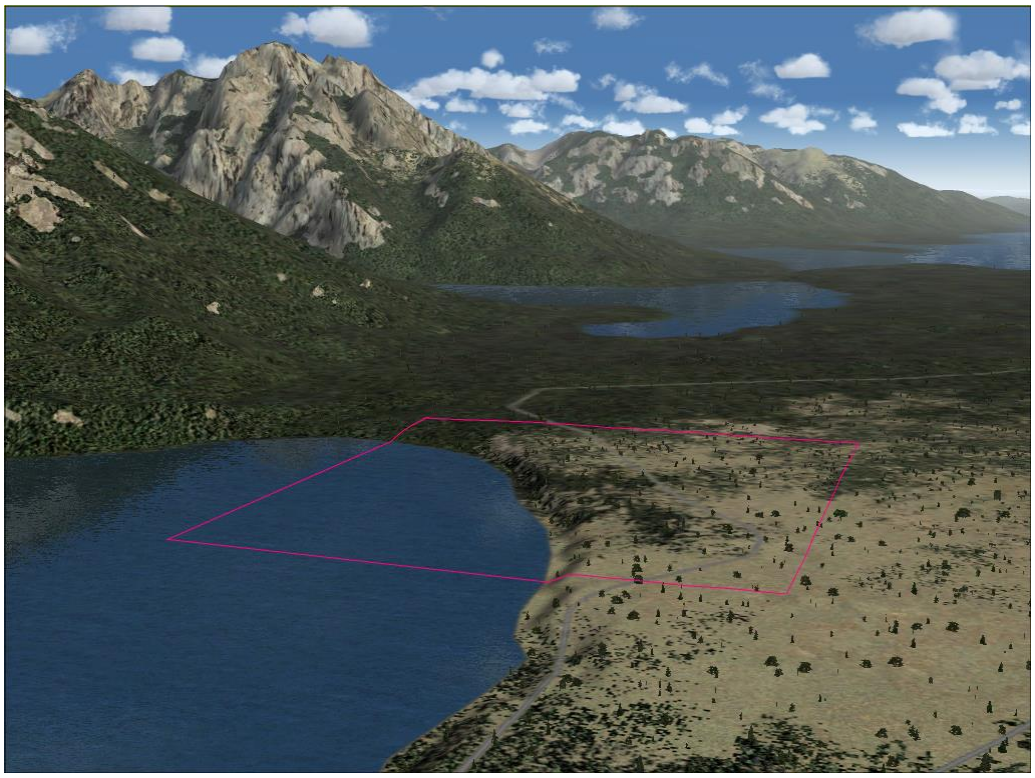

Figure 9: The location chosen to illustrate the aerial imagery synthesis process.

7.1. Building the class map

The first step in the texture synthesis process is to build the class map, a 2D lookup table of the land class textures used for each pixel of the final texture. We use the class map as a sort of digital “paint by numbers” diagram for building the final texture.

The initial list of textures is assembled by sampling the land classification and season raster data that overlaps the cell. The land classification and season data are authored at a resolution of approximately 1 kilometer such that the samples fall exactly on the corners of the cells in the quad tree. A lookup table is used to map the land class and season samples to a matching texture tile. If more than one land class is present in a cell, then we use a set of 1-bit masks to transition from one class to another.

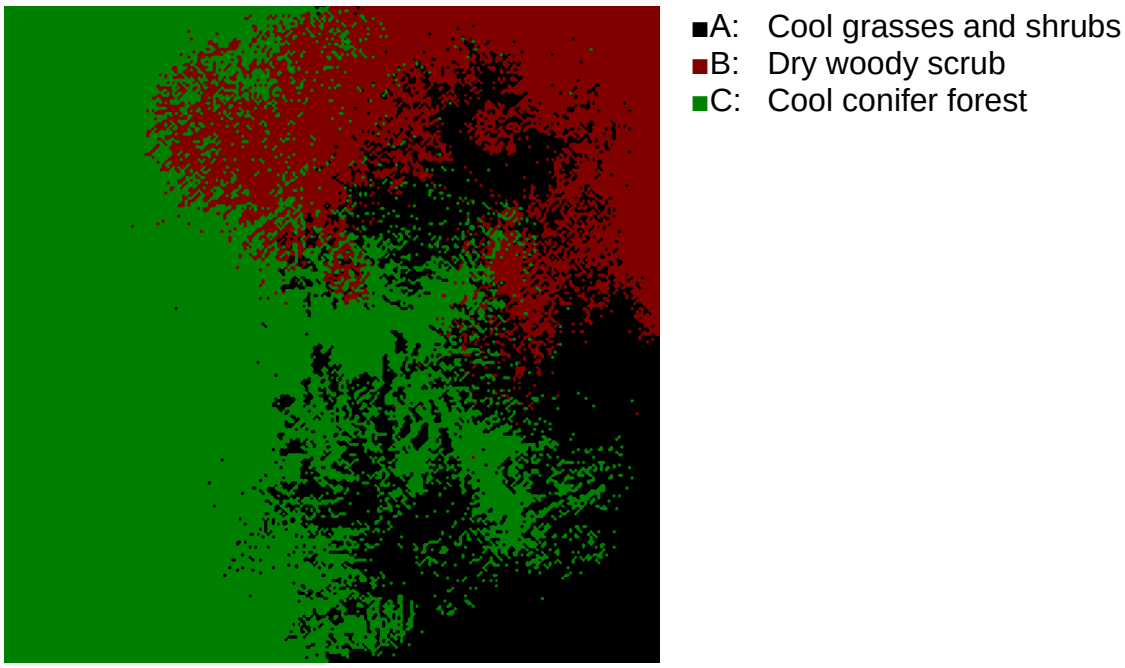

Figure 10: The class map after masking between different land classes at each corner.

On top of the blended land class textures, we draw the filled outlines of any bodies of water that intersect the cell. See Figure 11.

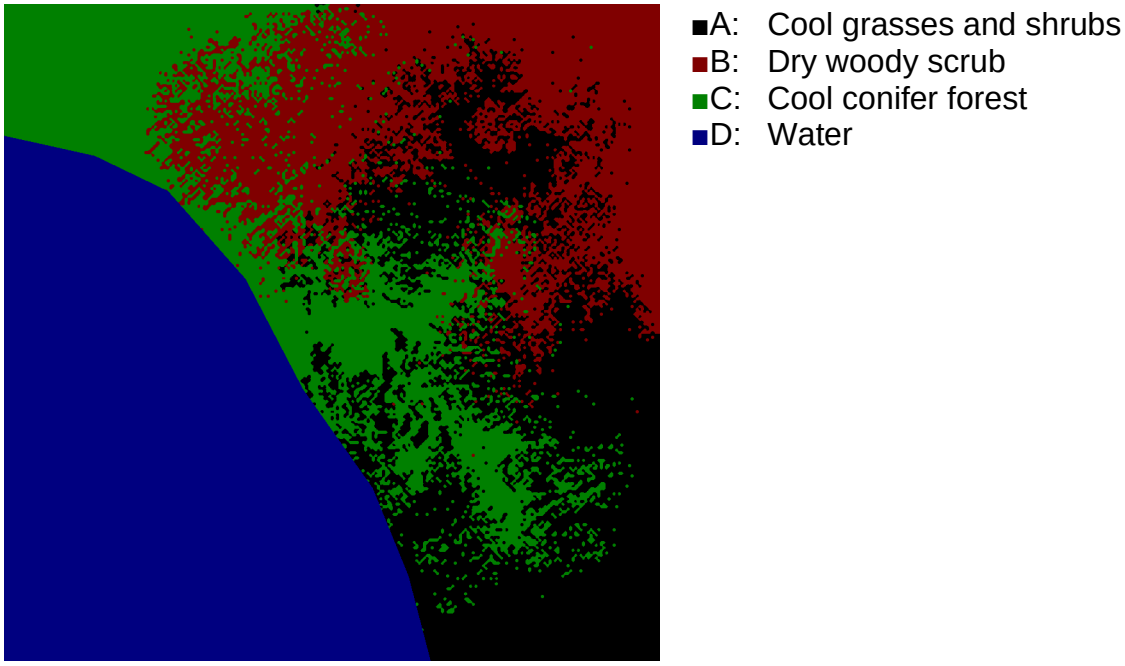

Figure 11: The class map after adding water.

Since our land classification and season data is fairly low resolution it is helpful to modulate the class map with the elevation data which is of much higher resolution. For example, in forested

mountainous areas, the carpet of trees is often interrupted by the steepest parts of the terrain. Likewise, in agricultural areas, the land being farmed is usually relatively flat and steeper slopes are left untouched. To simulate this effect, we compute the slope of each pixel in the class map and use a lookup table to convert the initial land class to something more appropriate in the steeper areas.

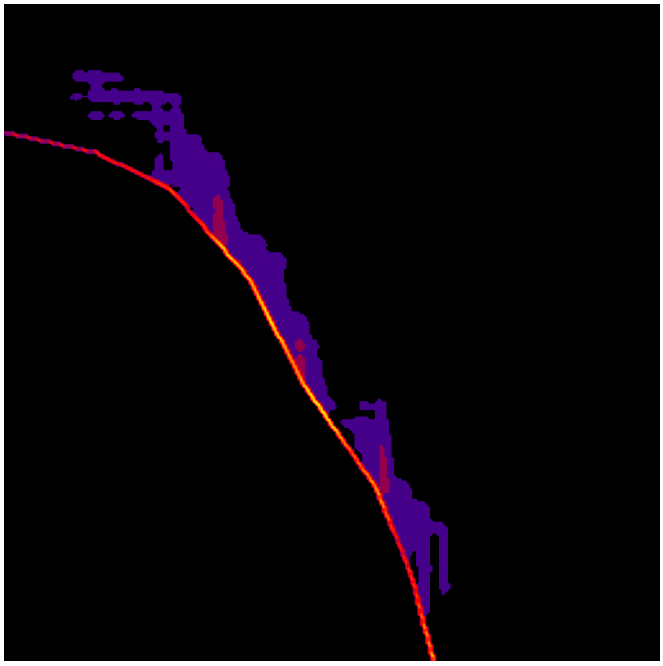

Figure 12: The slope map. Brightly colored areas have steeper slope. The steepest area is along the shoreline with more gradual slopes climbing up from the water to the right of the shore line.

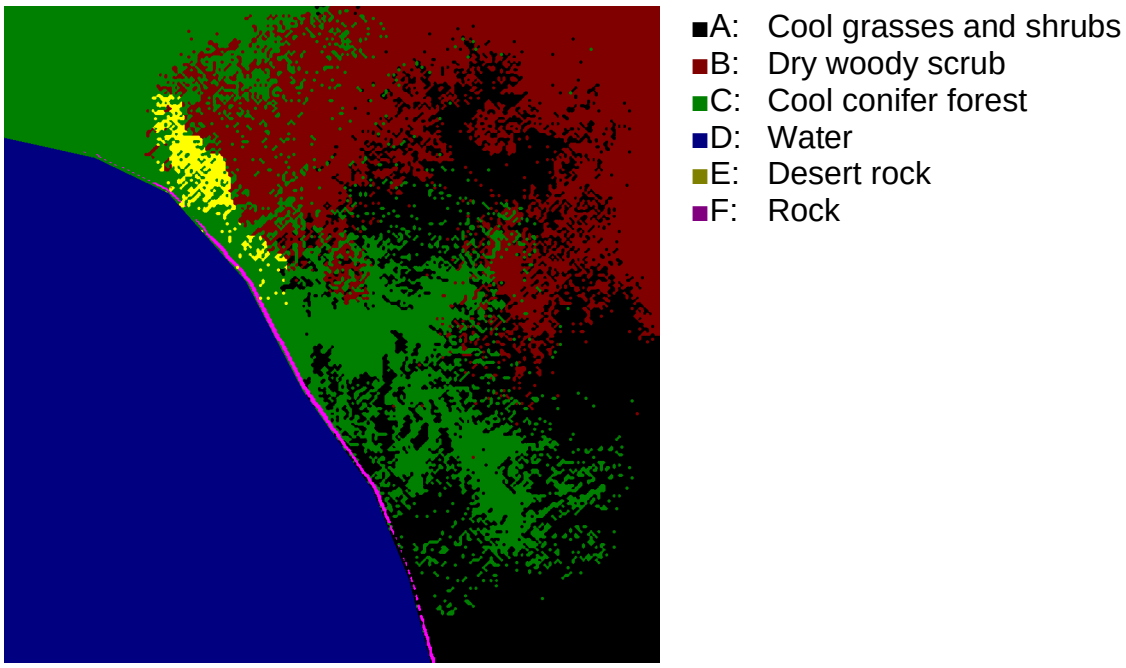

Figure 13: The class map after applying slope effects. Note the addition of desert rock where the dry woody scrub descends toward the lake. Also note the rock added along steepest parts of the shore line.

The last pass over the class map is to account for the effects of features such as streams, roads, and clearings around airports. For example, a stream through an arid landscape usually provides enough moisture for vegetation to grow on its banks. Likewise, road building usually affects the land immediately adjacent to the roadway because native vegetation is cut down and removed. Land cleared for airports is similarly affected. To simulate these effects, we rasterize a slightly enlarged footprint of these features into the class map and choose the modified land class from a lookup table. See Figure 14.

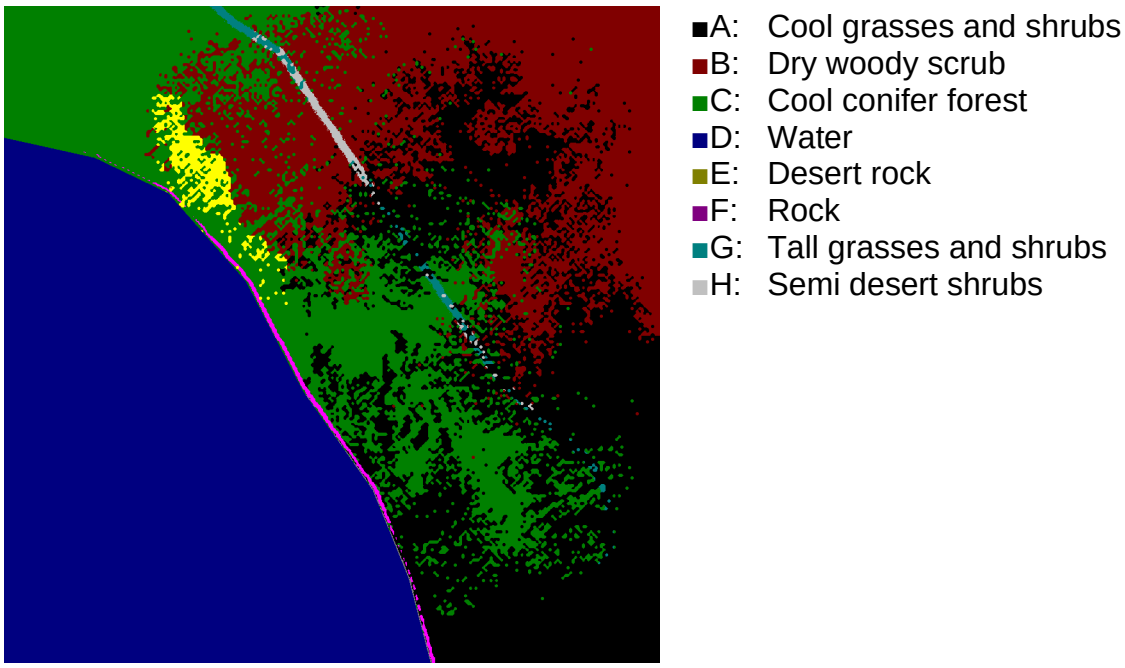

Figure 14: The class map after modification by vector features. In this example, a road replaced forest and scrub with grasses and small shrubs.

7.2. Apply land class textures

All land classes have a diffuse or “daytime” texture that represents what it looks like with the sun directly overhead. Some land classes also have an emissive or “nighttime” texture that is dark except for the effects of outdoor illumination like street lights. We need to keep the diffuse and emissive separated so we create two target images: one diffuse and one emissive.

Applying the land class textures to the synthesized diffuse and emissive images is fairly straightforward. We simply step through the class map’s texture list and blit them one-by-one onto the final image while using the class map as a stencil mask.

Cool grasses and shrubs

Dry woody scrub

Cool conifer forest

Water

Desert rock

Rock

Tall grasses and shrubs

Semi desert shrubs

Figure 15: Individual diffuse texture tiles for each land class in the class map.

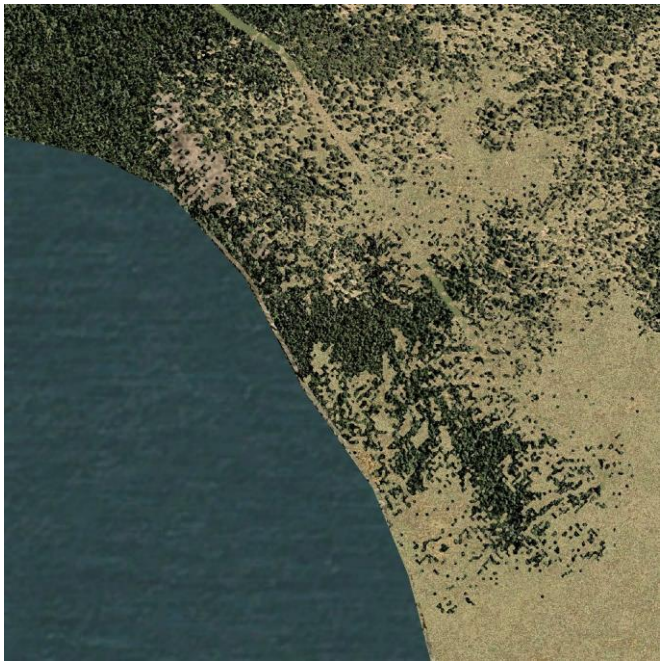

Figure 16: Diffuse texture synthesized by applying each of the individual land class texture tiles with the class map as a stencil mask.

7.3. Render linear vector features

At this point, we render a variety of linear vector features representing things like roads, streams, and shorelines. All are drawn into the target image as wide textured lines. Any roads with street lights get rendered in two passes, diffuse and emissive. The shorelines, usually just rendered into the diffuse target, help soften the edge between the land and water that was created when the water bodies were rasterized into the class map.

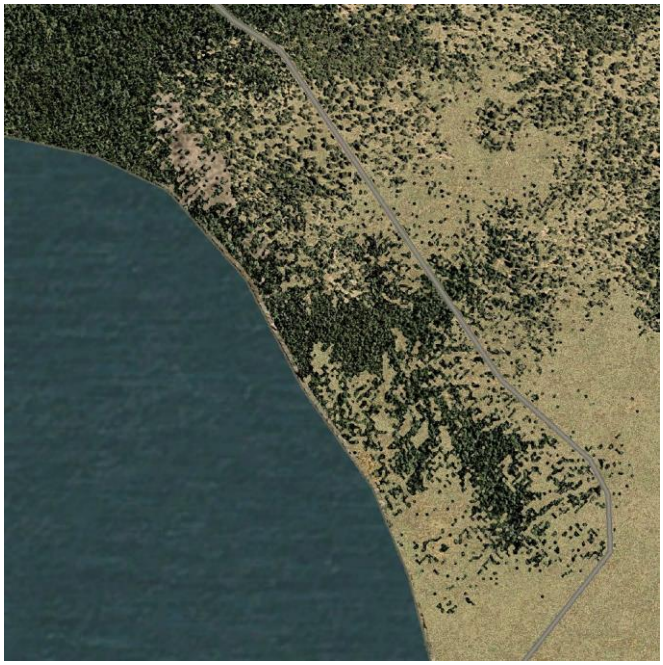

Figure 17: Diffuse texture after applying vector features.

7.4. Build light maps

To accurately light the synthesized texture, we need more than just the emissive texture created above. We also need to incorporate the light from the sun and moon and cast shadows across the terrain. All of these lighting components can be merged together into a single light map.

First we need to know the positions of the sun and moon. We also need to know the phase of the moon because it reflects more light when full. Believe it or not, Flight Simulator has a mini planetarium program running inside it to compute the positions of the sun, moon and nearly 10,000 stars based upon the date and time set in the game.

From the position of the brightest light source (sun during the day and moon at night), we determine which pixels of the synthesized texture are in shadow. Then we compute the amount of ambient and directional light at each pixel.

We modulate the ambient light by a simple radiosity computation approximating each pixel's exposure to the sky dome. The effect is subtle, but it tends to darken valleys and brighten pinnacles. The directional light is modulated by its dot product with the normal vector of the terrain. If a pixel is in shadow, then the directional light is completely attenuated. Finally, we add the emissive light for each pixel from the emissive image synthesized previously.

For best results, the resulting light map should be combined with the diffuse texture by the video hardware. However, previous versions of Flight Simulator combined the diffuse and light maps in software to save video memory and bus bandwidth.

8. Regarding threads, fibers, and multi-core architectures

Many of the tasks performed by the terrain engine, including the ones described in this paper, cannot be started and run to completion in the interval between consecutive rendered frames. Some of them would take several seconds to run even if 100% of the processor time was devoted to them. What's the best way to execute these tasks without negatively impacting the frame rate or its stability?

One possible solution is to run them on threads separate from the main render loop. In practice, however, threads alone may not be the best solution. The main reason is because the terrain engine's tasks are processor intensive and they can easily starve the game's main loop of processing time. Worse yet, the terrain tasks come and go intermittently which would cause the frame rate to fluctuate.

In theory, lowering the priority of the terrain threads should prevent the main game thread from being starved. However, the game thread is also pretty busy so that would increase the risk of starving the lower priority terrain threads.

A better solution, based on our experience, is to run most of the terrain engine tasks on Win32 fibers. Fibers are similar to threads in that they each have their own context (i.e. register state and call stack). However, the scheduling of fibers is entirely up to the application using them. With fibers, Flight Simulator can schedule the terrain tasks to run in the interval between iterations of the main game loop and not during rendering or other time critical operations.

Fibers do have their down side, however. Flight Simulator's fiber scheduler is cooperative, which means the fiber tasks must periodically call back into the scheduler to see if it's time to yield. While this takes some getting used to when writing code, it does simplify the job of synchronizing data between fibers and the main game loop because all of the possible points of preemption are known.

Recognizing the increased availability of dual-core processors, we plan to execute some fibers in a single thread on the second core. Fiber tasks that exchange data only with other fibers on the same core can still lightweight synchronization, but any data exchange with the main game loop or other fibers running on the primary core must be fully synchronized obviously.

9. Acknowledgements

Some of the techniques described in this paper were either adapted from published sources or designed from scratch by the following current and former Microsoft employees:

Jason Dent was the original developer of the terrain engine used in Flight Simulator. He devised the quad tree and cell addressing schemes and helped devise many of the texture synthesis techniques.

Jason Waskey, the Art Lead for the ACES Studio, was involved in the design of the texture synthesis, tiling, and masking techniques.

10. References

1. J. Grupping. Flight Simulator History web site. http://fshistory.simflight.com

2. B. Discoe, et al. Virtual Terrain Project web site, section entitled “Spherical Textures”. http://vterrain.org/Textures/spherical.html

3. G. Dutton. (1989). Planetary modeling via hierarchical tessellation, *Proc. Auto-Carto 9* . Falls Church, VA: ACSM-ASPRS, 462-471.

4. G. Fekete. Rendering and Managing Spherical Data with Sphere Quadtrees. In Proceedings of Visualization 90, 1990.

5. P. Lindstrom, D. Koller, W. Ribarsky, L. F. Hodges, N. Faust, and G. A. Turner. "Realtime, continuous level of detail rendering of height fields". In Proceedings SIGGRAPH 96, pages 109--118. ACM SIGGRAPH, 1996. http://citeseer.ist.psu.edu/lindstrom96realtime.html

6. Surface of the Earth Icosahedron Globe. National Geophysical Data Center. National Environmental Satellite, Data and Information Service. National Oceanic and Atmospheric Administration. U.S. Department of Commerce. http://www.ngdc.noaa.gov/mgg/fliers/04mgg02.html

7. R. Pajarola. Large scale terrain visualization using the restricted quadtree triangulation. In Proceedings IEEE Visualization'98, pages 19--24, Research Triangle Park, NC, 1998. IEEE Comp. Soc. Press. http://citeseer.ist.psu.edu/article/pajarola98large.html

8. ESRI ArcGIS Geographic Information System. http://www.esri.com

9. A. Asirvatham, H. Hoppe. Terrain rendering using GPU-based geometry clipmaps. GPU Gems 2, M. Pharr and R. Fernando, eds., Addison-Wesley, March 2005. http://research.microsoft.com/~hoppe/gpugcm.pdf

10. H. Malvar. Fast Progressive Image Coding without Wavelets. In Proceedings IEEE Data Compression Conference, Snowbird, UT, March 2000. http://research.microsoft.com/~malvar/papers/dcc00.pdf

11. M. Duchaineau, M. Wolinsky, D.E. Sigeti, M.C. Miller, C. Aldrich, and M.B. Mineed-Weinstein. Roaming terrain: Real-time optimally adapting meshes. In Proceedings IEEE Visualization'97, pages 81--88, 1997. http://citeseer.ist.psu.edu/duchaineau97roaming.html

12. NIMA Technical Report TR8350.2, Department of Defense World Geodetic System 1984, Its Definition and Relationships With Local Geodetic Systems, Third Edition, 4 July 1997. http://earth-info.nga.mil/GandG/publications/tr8350.2/tr8350_2.html

13. M. Lee, H. Samet. Navigating through Triangle Meshes Implemented as Linear Quadtrees, report CAR--TR--887, Computer Science Department, University of Maryland, 1998. http://citeseer.ist.psu.edu/lee98navigating.html

14. J. Waskey, Art Lead on Flight Simulator. Blog entry about the difficulty of using raw aerial imagery on terrain. December 30, 2005. http://blogs.technet.com/pixelpoke/archive/2005/12/30/416494.aspx

15. M. Shemanarev. Anti-Grain Geometry, High Fidelity 2D Graphics, A High Quality Rendering Engine for C++. http://www.antigrain.com