I

3

A WebGL Globe Rendering Pipeline

Patrick Cozzi and Daniel Bagnell

3.1 Introduction

WebGL brings hardware-accelerated graphics based on OpenGL ES 2.0 to the web. Combined with other HTML5 APIs, such as gamepad, fullscreen, and web audio, the web is becoming a viable platform for hardcore game development. However, there are also other killer applications for WebGL; one we are particularly interested in is mapping. The web has a long history of 2D maps such as MapQuest, OpenStreetMap, and Google Maps. WebGL enables web mapping to move from flat 2D maps to immersive 3D globes.

In this chapter, we present a WebGL globe rendering pipeline that integrates with hierarchical levels of detail (HLOD) algorithms used to manage high resolution imagery streamed from standard map servers, such as Esri or OpenStreetMap. Our pipeline uses screen-space techniques, including filling cracks between adjacent tiles with different LODs with a masked Gaussian blur, filling holes in the north and south pole with masking and ray casting, and avoiding depth fighting with vector data overlaid on the globe by rendering a *depth plane*.

We use these techniques in Cesium, http://cesium.agi.com, our open-source WebGL globe and map engine. We found them to be pragmatic, clean, and light on the CPU.

3.2 Rendering Pipeline Overview

Imagery is commonly served from map servers using 256×256 RGB tiles. Tiles are organized hierarchically in a quadtree, where the root node covers the entire globe, $-180°$ to $180°$ longitude and $-90°$ to $90°$ latitude,[1] at a very low resolu-

[1] As we'll see in Section 3.4 the latitude bounds are actually $\approx \pm 85°$ for the most common projection.

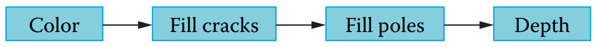


Figure 3.1. Our globe rendering pipeline.

tion. As we continue down the tree, tiles remain 256 × 256 but cover a smaller longitude-latitude extent, thus increasing their resolution. The extent of one of the root's child tiles is −180° to 0° longitude and 0° to 90°, and the extent of one of its grandchild tiles is −180° to −90° longitude and 45° to 90° latitude.

A 2D map can easily request tiles for rendering based on the visible longitude-latitude extent clipped to the viewport, and the zoom level. In 3D, when tiles are mapped onto a WGS84 ellipsoid representing the globe, more general HLOD algorithms are used to select geometry, i.e., tessellated patches of the ellipsoid at different resolutions, and imagery tiles to render based on the view parameters and a pixel error tolerance. HLOD algorithms produce a set of tiles, both geometry and texture, to be rendered for a given frame.

Our pipeline renders these tiles in the four steps shown in Figure 3.1. First, tiles are rendered to the color buffer only; the depth test is disabled. Next, to fill cracks between adjacent tiles with different geometric LODs, a screen-space Gaussian blur is performed. A fragment is only blurred if it is part of a crack; therefore, most fragments are not changed.

Next, since most map servers do not provide tiles near the poles, two viewport-aligned quads are rendered, and masked and ray-casted to fill the holes in the poles.

Finally, a depth-only pass renders a plane perpendicular to the near plane that slices the globe at the horizon. A ray is traced through each fragment and discarded if it does not intersect the globe's ellipsoid. The remaining fragment's depth values are written, allowing later passes to draw vector data on the globe without z-fighting or tessellation differences between tiles and vector data rendering later.

Let's look these steps in more detail.

3.3 Filling Cracks in Screen Space

Cracks like those in Figure 3.2 occur in HLOD algorithms when two adjacent tiles have different geometric LODs. Since the tile's tessellations are at different resolutions, together they do not form a watertight mesh, and gaps are noticeable.

There are a wide array of geometric techniques for filling cracks. A key observation is that cracking artifacts need to be removed, but adjacent tiles don't necessarily need to line up vertex-to-vertex. For example, in terrain rendering, it is common to drop flanges, ribbons, or skirts vertically or at a slight angle to minimize noticeable artifacts [Ulrich 02]. This, of course, requires creating extra geometry and determining its length and orientation. Other geometric

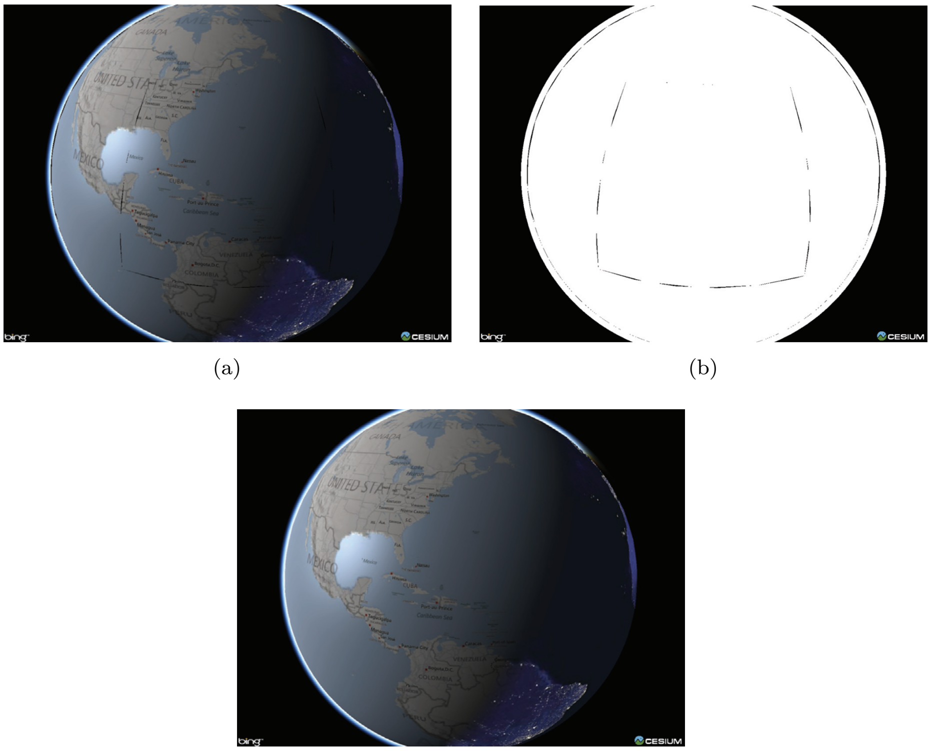


Figure 3.2. (a) Cracking between adjacent tiles with different geometric LODs (exaggerated for clarity). (b) An alpha mask that is white where tiles are rendered. (c) Cracks are detected using the alpha mask, and filled using a Gaussian blur.

techniques involve matching vertices in adjacent tiles by marking triangles that overlap a boundary as restricted during decimation [Erikson et al. 01].

Instead of filling cracks geometrically, we fill them in screen space. Cracks need to be filled with something plausible, but not necessarily extra geometry. To shade a fragment in a crack, we use a Gaussian blur that only includes samples from surrounding fragments not in the crack. The result is visually plausible, simple to implement, and well-suited to WebGL since it requires no extra geometry and is light on the CPU.

In the first rendering pass, each tile's color is rendered, and the alpha channel is set to 1.0. As shown in Figure 3.2(b), the alpha mask is 1.0 where tiles were rendered, and 0.0 in cracks and the sky.

To fill the cracks, a bounding sphere encompassing the ellipsoid is projected into screen space, and the bounding rectangle is found to reduce fragment work-load. A viewport-aligned quad is rendered in two passes, one vertical and one horizontal, to perform a masked Gaussian blur. Using two passes instead of one reduces the number of texture reads from n^2 to $n + n$ for an $n \times n$ kernel [Rákos 10].

The fragment shader first reads the alpha mask. If the alpha is 1.0, the color is passed through since the fragment is part of a tile. Otherwise, two other values are read from the alpha mask. For the vertical pass, these are the topmost and bottommost texels in the kernel's column for this fragment. For example, for a 7 × 7 kernel, the texels are $(0, \pm 3)$ texels. If the alpha for both texels is 0.0, sky is detected, and the color is passed through. Otherwise, the blur is performed including only texels in the kernel with an alpha of `1.0`. Essentially, pixels surrounding cracks are bled into the cracks to fill them.

Kernel size selection presents an important tradeoff. An $n \times n$ kernel can only fill cracks up to $n - 2 \times n - 2$ pixels because of the sky check. We found that a 7×7 kernel works well in practice for our engine. However, we do not restrict geometry such that cracks will never exceed five pixels. It is possible cracks will not be completely filled, but we have found these cases to be quite rare.

3.4 Filling Poles in Screen Space

Most standard map servers provide tiles in the Web Mercator projection. Due to the projection, tiles are not available above 85.05112878° latitude and below −85.05112878°. Not rendering these tiles results in holes in the poles as shown in Figures 3.3(a) and 3.4(a). For many zoomed-in views, the holes are not visible. However, for global views, they are obvious.

There are several geometric solutions to fill the holes. Tiles can be created above and below the latitude bounds, essentially creating the same geometry as if image tiles were available. This is simple but can lead to over-tessellation at the poles. To avoid this, a projection tailored to the poles can be used similar to Miller and Gaskins [Miller and Gaskins 09]. However, this requires a good bit of code, and cracking between the two different tessellation methods needs to addressed. Alternatively, a coarsely tessellated globe can be rendered after the imagery tiles, but tessellation and lighting discontinuities can be noticeable.

Instead of a geometric solution, holes can be filled in screen space, avoiding any concerns about over-tessellation or cracking. We render viewport-aligned quads covering the poles, and detect and shade the holes in a fragment shader.

For each pole, first compute a bounding sphere around the pole's longitude-latitude extent; for example, the north pole extent is −180° to 180° longitude, and 85.05112878° to 90° latitude. The sphere can be frustum and occlusion culled like any other geometry. If visible, the bounding rectangle of the sphere projected into screen space is computed, and a viewport-aligned quad is rendered as shown in Figure 3.3(b).

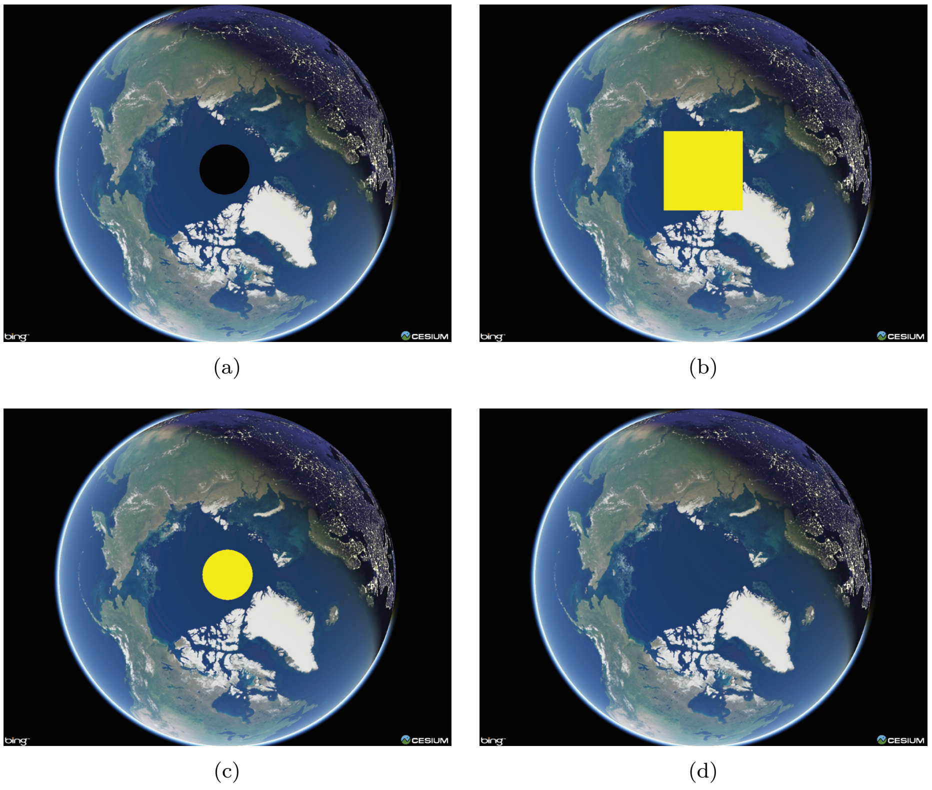


Figure 3.3. (a) A hole at the north pole since tiles are not available outside of $\approx \pm 85°$ latitude. (b) A rectangle covering the hole in screen space. (c) Masking and ray casting to detect the hole. (d) Shading the hole.

The quad is larger than the hole behind it. For horizon views, it extends high above the ground. The fragment shader discards fragments that are not in front of the hole by first checking the alpha mask; if it is 1.0, a tile was rendered to this fragment, and it is not a hole. To discard fragments above ground, a ray is cast from the eye through the fragment to the ellipsoid. If the ray doesn't hit the ellipsoid, the fragment is above the ground, and discarded. At this point, if the fragment was not discarded, it covers the hole as in Figure 3.3(c).

We shade the fragment by computing the geodetic surface normal on the ellipsoid where the ray intersects, and use that for lighting with a solid diffuse color. Texture coordinates can also be computed and used for specular maps and other effects. Given that the poles are mostly uniform color, a solid diffuse color looks acceptable as shown in Figures 3.3(d) and 3.4(b).

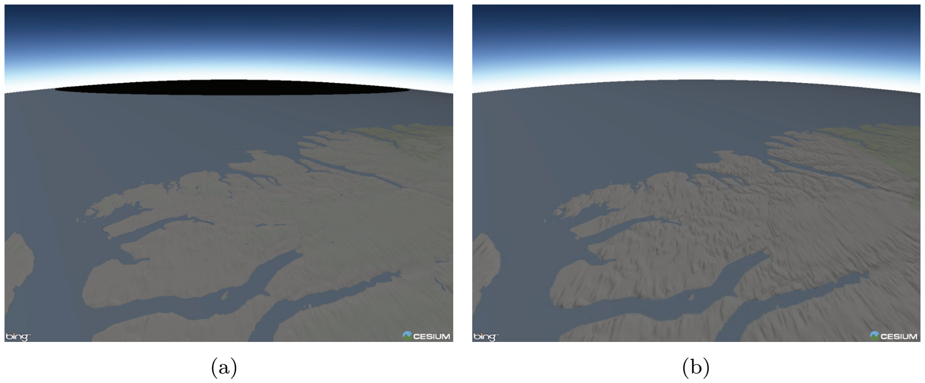


Figure 3.4. Horizon views. (a) A hole at the north pole. (b) The filled hole.

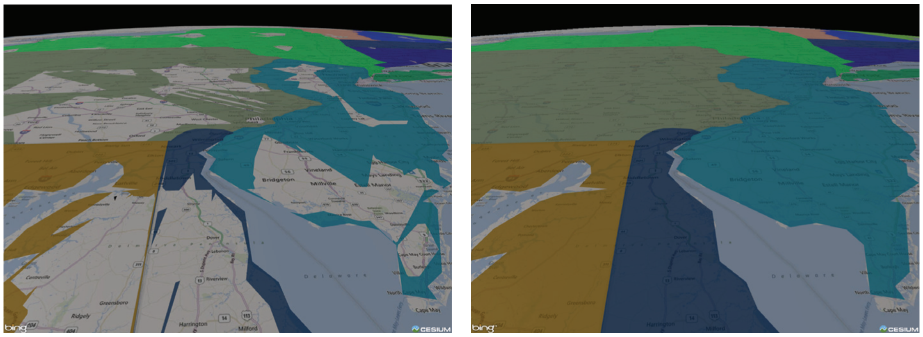
Compared to geometric solutions, our screen-space approach has excellent visual quality with no tessellation or geometric cracking artifacts, uses very little memory, and is light on the CPU, requiring only the bounding rectangle computations, culling, and two draw calls. However, its fragment load can be higher given that many fragments are discarded from extreme horizon views,[2] early-z and hierarchical-z are disabled due to using `discard`, and that a ray/fragment intersection test is used. Given the speed difference between JavaScript and C++, and CPUs and GPUs, we believe this is a good tradeoff.

3.5 Overlaying Vector Data

Maps often overlay vector data, i.e., points, polylines, and polygons, on top of the globe. For example, points may represent cities, polylines may represent driving directions, and polygons may represent countries. Either raster or vector techniques can be used to render this data [Cozzi and Ring 11]. Raster techniques burn vector data into image tiles with an alpha channel. These tiles are then overlaid on top of the base map imagery. This is widely used; it keeps the rendering code simple. However, as the viewer zooms in, aliasing can become apparent, it is slow for dynamic data, and does not support points as viewport-aligned labels and billboards.

To overcome these limitations, we render vector data using point, line, and triangle primitives, which requires subdividing polylines and polygons to approximate the curvature of the globe. Given that these polygonal representations only represent the true globe surface when infinitely subdivided, the primitives are

[2]This could be reduced by using a screen-space rectangle that is the intersection of the pole's projected rectangle and the ellipsoid's projected rectangle.

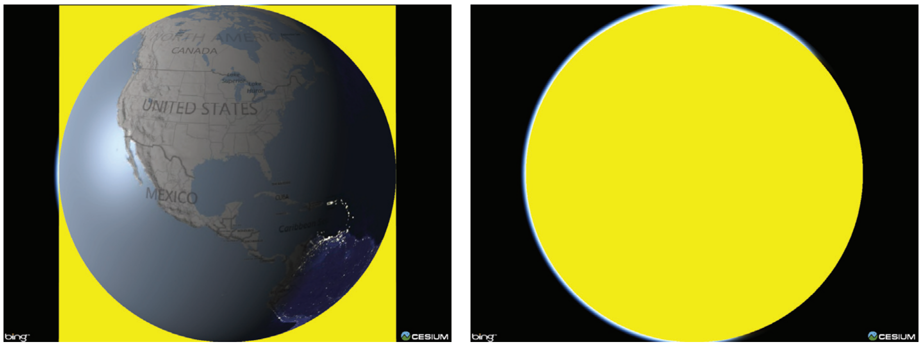


(a) (b)

Figure 3.5. (a) Vector data drawn without the depth plane result in artifacts. (b) With the depth plane.

actually under the true globe, which itself is approximated by triangles. With standard depth testing, parts of the vector data will fail the depth test and z-fight with the globe as shown in Figure 3.5.

We solve this using a method that is well-suited for JavaScript and WebGL; it uses very little CPU and does not rely on being able to write `gl_FragDepth`.[3] The key observation is that objects on or above the backside of the globe should fail the depth test, while objects on—but actually under—the front side of the globe should pass. We achieve this by rendering a *depth plane*, shown in Figure 3.6(a),



(a) (b)

Figure 3.6. (a) The depth plane intersects the globe at the horizon. (b) A ray is sent through each fragment in the plane to determine which fragments intersect the globe, and, therefore, need to write depth.

[3] We expect an extension for writing depth in the future.

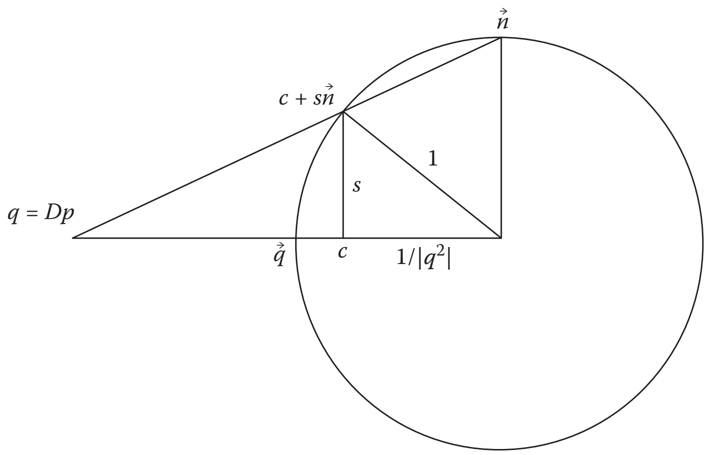


Figure 3.7. Computing the depth plane: a cross section of the ellipsoid scaled to a sphere as viewed from east of the camera.

in a depth-only pass that is perpendicular to the near plane and intersects the globe at the horizon. This plane is computed by determining the visible longitude and latitude extents given the camera position.

Following Figure 3.7, for an ellipsoid, $\frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2} = 1$, and camera position in WGS84 coordinates, p, we compute

$$D = \begin{pmatrix} \frac{1}{a} & 0 & 0 \\ 0 & \frac{1}{b} & 0 \\ 0 & 0 & \frac{1}{c} \end{pmatrix},$$

$$q = Dp,$$

where D is a scale matrix that transforms from the ellipsoid to a unit sphere and q is the camera position in the scaled space. Next, we compute the east vector, $\vec{e}$; the north vector, $\vec{n}$; the center of the circle where the depth plane intersects the unit sphere, c; and the radius of that circle, s:

$$\vec{e} = (0,0,1) \times \vec{q},$$
$$\vec{n} = \vec{q} \times \vec{e},$$
$$c = \frac{q}{|q|^2},$$
$$s = \sin \arccos \frac{1}{|q|^2}.$$

Finally, we compute the corners of the visible extent in WGS84 coordinates.

$$\begin{aligned}
\text{upper left} &= D^{-1}\left(c + s\left(\vec{n} - \vec{e}\right)\right)\\
\text{upper right} &= D^{-1}\left(c + s\left(\vec{n} + \vec{e}\right)\right)\\
\text{lower left} &= D^{-1}\left(c + s\left(-\vec{n} - \vec{e}\right)\right)\\
\text{lower right} &= D^{-1}\left(c + s\left(-\vec{n} + \vec{e}\right)\right)
\end{aligned}$$

In the fragment shader, a ray is cast from the eye through each fragment in the depth plane. If the fragment does not intersect the ellipsoid, the fragment is discarded as shown in Figure 3.6(b). The result is the globe's depth is replaced with the depth plane's depth, which allows objects on the front side of the globe to pass the depth test and those on the backside to fail without z-fighting or tessellation differences between image tiles and vector data. This is like backface culling, except it doesn't require the primitives to be backfacing; for example, a model of a satellite works with the depth plane but does not work with backface culling alone.

We originally used a technique based on backface culling. First, we rendered the tiles without depth. Next, we rendered polygons and polylines on the ellipsoid's surface without depth, and with backface culling implemented by discarding in the fragment shader based on the ellipsoid's geodetic surface normal. Finally, we rendered the tile's depth. Like the depth plane, this did not require writing `gl_FragDepth`; however, it had created a shortcoming in our API. Users needed to specify if a polygon or polyline was on the surface or in space. The depth plane works in both cases except for the rare exception of polylines normal to and intersecting the ellipsoid. The backface-culling technique also relies on two passes over the tiles, which increases the number of draw calls. This is a major WebGL bottleneck.

3.6 Conclusion

As long-time C++ and desktop OpenGL developers, we have found JavaScript and WebGL to be a viable platform for serious graphics development. We hope this chapter provided both inspiration for what is possible with WebGL, and concrete techniques for globe rendering that are well-suited to WebGL. To see these techniques in action, see our live demos at http://cesium.agi.com.

3.7 Acknowledgments

We thank Matt Amato, Norm Badler, Wolfgang Engel, Scott Hunter, and Kevin Ring for reviewing this chapter. We especially thank Frank Stoner for deriving the equations for the depth plane.

Bibliography

[Cozzi and Ring 11] Patrick Cozzi and Kevin Ring. *3D Engine Design for Virtual Globes.* Boca Raton: CRC Press, 2011. (Information at http://www.virtualglobebook.com.)

[Erikson et al. 01] Carl Erikson, Dinesh Manocha, and William V. Baxter III. “HLODs for Faster Display of Large Static and Dynamic Environments.” In *Proceedings of the 2001 Symposium on Interactive 3D Graphics*, pp. 111–120. New York: ACM, 2001. (Available at http://gamma.cs.unc.edu/POWERPLANT/papers/erikson2001.pdf.)

[Miller and Gaskins 09] James R. Miller and Tom Gaskins. “Computations on an Ellipsoid for GIS.” *Computer-Aided Design* 6:4 (2009), 575–583. (Available at http://people.eecs.ku.edu/~miller/Papers/CAD_6_4__575-583.pdf.)

[Rákos 10] Daniel Rákos. “Efficient Gaussian Blur with Linear Sampling.” *RasterGrid Blogosphere*, http://rastergrid.com/blog/2010/09/efficient-gaussian-blur-with-linear-sampling/, 2010.

[Ulrich 02] Thatcher Ulrich. “Rendering Massive Terrains Using Chunked Level of Detail Control.” *SIGGRAPH 2002 Super-Size It! Scaling Up to Massive Virtual Worlds Course Notes.* http://tulrich.com/geekstuff/sig-notes.pdf, 2002.