3D Engine Design for Virtual Globes

Patrick Cozzi and Kevin Ring

Editorial, Sales, and Customer Service Office

A K Peters, Ltd.
5 Commonwealth Road, Suite 2C
Natick, MA 01760
www.akpeters.com

Library of Congress Cataloging-in-Publication Data

To be determined

Printed in the United States of America
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

Try This

Run the night-lights example with a frame-rate utility. Note the frame rate when viewing just the daytime side of the globe and just the night-time side. Why is the frame rate higher for the nighttime side? Because the night-lights texture is a lower resolution than the daytime texture and does not require any lighting computations. This shows using dynamic branching to improve performance.

Virtual globe applications use real-world data like this night-light texture derived from satellite imagery. On the other hand, video games generally focus on creating a wide array of convincing artificial data using very little memory. For example, EVE Online takes an interesting approach to rendering night lights for their planets [109]. Instead of relying on a night-light texture whose texels are directly looked up, a texture atlas of night lights is used. The spherically mapped texture coordinates are used to look up surrogate texture coordinates, which map into the texture atlas. This allows a lot of variation from a single texture atlas because sections can be rotated and mirrored.

Rendering night lights is one of many uses for multitexturing in globe rendering. Other uses include cloud textures and gloss maps to show specular highlights on bodies of waters [57,147]. Before the multitexturing hardware was available, effects like these required multiple rendering passes. STK, being one of the first products to implement night lights, uses a multiple-pass approach.

4.3 GPU Ray Casting

GPUs are built to rasterize triangles at very rapid rates. The purpose of ellipsoid-tessellation algorithms is to create triangles that approximate the shape of a globe. These triangles are fed to the GPU, which rapidly rasterizes them into shaded pixels, creating an interactive visualization of the globe. This process is very fast because it is embarrassingly parallel; individual triangles and fragments are processed independently, in a massively parallel fashion. Since tessellation is required, rendering a globe this way is not without its flaws:

- No single tessellation is perfect; each has different strengths and weaknesses.
- Under-tessellation leads to a coarse triangle mesh that does not approximate the surface well, and over-tessellation creates too many

triangles, negatively affecting performance and memory usage. View-dependent level-of-detail algorithms are required for most applications to strike a balance.

- Although GPUs exploit the parallelism of rasterization, memories are not keeping pace with the increasing computation power, so a large number of triangles can negatively impact performance. This is especially true of some level-of-detail algorithms where new meshes are frequently sent over the system bus.

Ray tracing is an alternative to rasterization. Rasterization starts with triangles and ends with pixels. Ray tracing takes the opposite approach: it starts with pixels and asks what triangle(s), or objects in general, contribute to the color of this pixel. For perspective views, a ray is cast from the eye through each pixel into the scene. In the simplest case, called *ray casting*, the first object intersecting each ray is found, and lighting computations are performed to produce the final image.

A strength of ray casting is that objects do not need to be tessellated into triangles for rendering. If we can figure out how to intersect a ray with an object, then we can render it. Therefore, no tessellation is required to render a globe represented by an ellipsoid because there is a well-known equation for intersecting a ray with an ellipsoid's implicit surface. The benefits of ray casting a globe include the following:

- The ellipsoid is automatically rendered with an infinite level of detail. For example, as the viewer zooms in, the underlying triangle mesh does not become apparent because there is no triangle mesh; intersecting a ray with an ellipsoid produces an infinitely smooth surface.

- Since there are no triangles, there is no concern about creating thin triangles, triangles crossing the poles, or triangles crossing the IDL. Many of the weaknesses of tessellation algorithms go away.

- Significantly less memory is required since a triangle mesh is not stored or sent across the system bus. This is particularly important in a world where size is speed.

Since current GPUs are built for rasterization, you may wonder how to efficiently ray cast a globe. In a naïve CPU implementation, a nested `for` loop iterates over each pixel in the scene and performs a ray/ellipsoid intersection. Like rasterization, ray casting is embarrassingly parallel. Therefore, a wide array of optimizations are possible on today's CPUs, including casting each ray in a separate thread and utilizing single instruction multiple data (SIMD) instructions. Even with these optimizations, CPUs

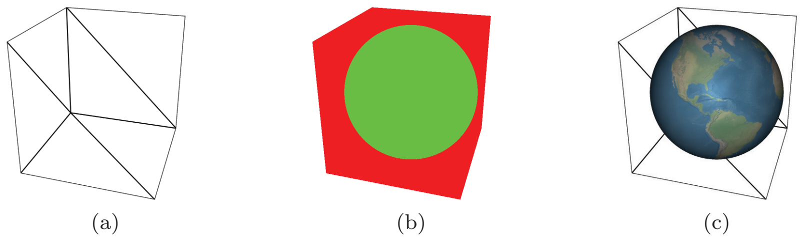

Figure 4.17. In GPU ray casting, (a) a box is rendered to (b) invoke a ray-casting fragment shader that finds the ellipsoid's visible surface. When an intersection is found, (c) the geodetic surface normal is used for shading.

do not support the massive parallelism of GPUs. Since GPUs are built for rasterization, the question is how do we use them for efficient ray casting?

Fragment shaders provide the perfect vehicle for ray casting on the GPU. Instead of tessellating an ellipsoid, create geometry for a bounding box around the ellipsoid. Then, render this box using normal rasterization and cast a ray from the eye to each fragment created by the box. If the ray intersects the inscribed ellipsoid, shade the fragment; otherwise, discard it.

The box is rendered with front-face culling, as shown in Figure 4.17(a). Front-facing culling is used instead of back-face culling so the globe still appears when the viewer is inside the box.

This is the only geometry that needs to be processed to render the ellipsoid, a constant vertex load of 12 triangles. With front-face culling, fragments for six of the triangles are processed for most views. The result is that a fragment shader is run for each fragment we want to cast a ray through. Since the fragment shader can access the camera's world-space position through a uniform, and the vertex shader can pass the vertex's interpolated world-space position to the fragment shader, a ray can be constructed from the eye through each fragment's position.[4] The ray simply has an origin of `og_cameraEye` and a direction of normalize(`worldPosition` − `og_cameraEye`).

The fragment shader also needs access to the ellipsoid's center and radii. Since it is assumed that the ellipsoid is centered at the origin, the fragment shader just needs a uniform for the ellipsoid's radii. In practice, intersecting a ray with an ellipsoid requires $\frac{1}{\text{radii}^2}$, so that should be precomputed once on the CPU and passed to the fragment shader as a uniform. Given the

[4]In this case, a ray is cast in world coordinates with the ellipsoid's center at the origin. It is also common to perform ray casting in eye coordinates, where the ray's origin is the coordinate system's origin. What really matters is that the ray and object are in the same coordinate system.

ray and ellipsoid information, Listing 4.16 shows a fragment shader that colors fragments green if a ray through the fragment intersects the ellipsoid or red if the ray does not intersect, as shown in Figure 4.17(b).

This shader has two shortcomings. First, it does not do any actual shading. Fortunately, given the position and surface normal of the ray intersection, shading can utilize the same techniques used throughout this chapter, namely `LightIntensity`() and `ComputeTextureCoordinates`(). Listing 4.17 adds shading by computing the position of the intersection along the ray using `i.Time` and shading as usual. If the ray does not intersect the ellipsoid, the fragment is discarded. Unfortunately, using discard has the adverse effect of disabling GPU depth buffer optimizations, including fine-grained early-z and coarse-grained z-cull, as discussed in Section 12.4.5.

```
in vec3 worldPosition;
out vec3 fragmentColor;
uniform vec3 og_cameraEye;
uniform vec3 u_globeOneOverRadiiSquared;

struct Intersection
{
  bool Intersects;
  float Time;       // Time of intersection along ray
};

Intersection RayIntersectEllipsoid(vec3 rayOrigin,
    vec3 rayDirection, vec3 oneOverEllipsoidRadiiSquared)
{ // ... }

void main()
{
  vec3 rayDirection = normalize(worldPosition - og_cameraEye);
  Intersection i = RayIntersectEllipsoid(og_cameraEye,
      rayDirection, u_globeOneOverRadiiSquared);
  fragmentColor = vec3(i.Intersects, !i.Intersects, 0.0);
}
```

Listing 4.16. Base GLSL fragment shader for ray casting.

```
// ...
vec3 GeodeticSurfaceNormal(vec3 positionOnEllipsoid,
                           vec3 oneOverEllipsoidRadiiSquared)
{
  return normalize(positionOnEllipsoid *
                   oneOverEllipsoidRadiiSquared);
}

void main()
{
  vec3 rayDirection = normalize(worldPosition - og_cameraEye);
  Intersection i = RayIntersectEllipsoid(og_cameraEye,
      rayDirection, u_globeOneOverRadiiSquared);
  if (i.Intersects)
  {
    vec3 position = og_cameraEye + (i.Time * rayDirection);
    vec3 normal = GeodeticSurfaceNormal(position,
        u_globeOneOverRadiiSquared);

    vec3 toLight = normalize(og_cameraLightPosition - position);
    vec3 toEye = normalize(og_cameraEye - position);
    float intensity = LightIntensity(normal, toLight, toEye,
        og_diffuseSpecularAmbientShininess);

    fragmentColor = intensity * texture(og_texture0,
        ComputeTextureCoordinates(normal)).rgb;
  }
  else
  {
    discard;
  }
}
```

Listing 4.17. Shading or discarding a fragment based on a ray cast.

```
float ComputeWorldPositionDepth(vec3 position)
{
  vec4 v = og_modelViewPerspectiveMatrix * vec4(position, 1);
  v.z /= v.w;
  v.z = (v.z + 1.0) * 0.5;
  return v.z;
}
```

Listing 4.18. Computing depth for a world-space position.

The remaining shortcoming, which may not be obvious until other objects are rendered in the scene, is that incorrect depth values are written. When an intersection occurs, the box's depth is written instead of the ellipsoid's depth. This can be corrected by computing the ellipsoid's depth, as shown in Listing 4.18, and writing it to gl_FragDepth. Depth is computed by transforming the world-space positions of the intersection into clip coordinates, then transforming this z-value into normalized device coordinates and, finally, into window coordinates. The final result of GPU ray casting, with shading and correct depth, is shown in Figure 4.17(c).

Since this algorithm doesn't have any overdraw, all the red pixels in Figure 4.17(b) are wasted fragment shading. A tessellated ellipsoid rendered with back-face culling does not have wasted fragments. On most GPUs, this is not as bad as it seems since the dynamic branch will avoid the shading computations [135, 144, 168], including the expensive inverse trigonometry for texture-coordinate generation. Furthermore, since the branches are coherent, that is, adjacent fragments in screen space are likely to take the same branch, except around the ellipsoid's silhouette, the GPU's parallelism is used well [168].

To reduce the number of rays that miss the ellipsoid, a viewport-aligned convex polygon bounding the ellipsoid from the viewer's perspective can be used instead of a bounding box [30]. The number of points in the bounding polygon determine how tight the fit is and, thus, how many rays miss the ellipsoid. This creates a trade-off between vertex and fragment processing.

GPU ray casting an ellipsoid fits seamlessly into the rasterization pipeline, making it an attractive alternative to rendering a tessellated approximation. In the general case, GPU ray casting, and full ray tracing in particular, is difficult. Not all objects have an efficient ray intersection test like an ellipsoid, and large scenes require hierarchical spatial data structures for quickly finding which objects a ray may intersect. These types of linked data structures are difficult to implement on today's GPUs, especially for dynamic scenes. Furthermore, in ray tracing, the number of rays quickly explodes with effects like soft shadows and antialiasing. Nonetheless, GPU ray tracing is a promising, active area of research [134, 178].

4.4 Resources

A detailed description of computing a polygonal approximation to a sphere using subdivision surfaces, aimed towards introductory graphics students, is provided by Angel [7]. The book is an excellent introduction to computer graphics in general. A survey of subdivision-surface algorithms is presented in *Real-Time Rendering* [3]. The book itself is an indispensable survey of real-time rendering. See "The Orange Book" for more information on using multitexturing in fragment shaders to render the Earth [147]. The book is generally useful as it thoroughly covers GLSL and provides a wide range of example shaders.

An ellipsoid tessellation based on the honeycomb [39], a figure derived from a soccer ball, may prove advantageous over subdividing platonic solids, which leads to a nonuniform tessellation. Another alternative to the tessellation algorithms discussed in this chapter is the HEALPix [65].

A series on procedurally generating 3D planets covers many relevant topics, including cube-map tessellation, level of detail, and shading [182]. An experimental globe-tessellation algorithm for NASA World Wind is described by Miller and Gaskins [116].

The entire field of real-time ray tracing is discussed by Wald [178], including GPU approaches. A high-level discussion on ray tracing virtual globes, with a focus on improving visual quality, is presented by Christen [26].