projected onto the surface of the ellipsoid, as shown in Figure 8.15(d), or even at a constant height above the ellipsoid. In fact, we'd like to approximate the intersection of the ellipsoid and the infinite pyramid formed from the ellipsoid's center and the triangle (recall Figure 8.14). This will result in curves between triangle endpoints.

In order to approximate the triangle's projection, we use a subdivision stage immediately after triangulation (see Figure 8.16). This algorithm is very similar to the subdivision-surfaces algorithm discussed in Section 4.1.1, which subdivides a regular tetrahedron into an ellipsoid. Here, the idea is to subdivide each triangle into two new triangles until a stopping condition is satisfied.

We'll use *granularity* as our stopping condition. Consider two points, p and q, forming an edge of a triangle. The granularity is the angle between $\mathbf{p} - \mathbf{0}$ and $\mathbf{q} - \mathbf{0}$. For our stopping condition to be satisfied, the granularity of every edge in a triangle needs to be less than or equal to a given

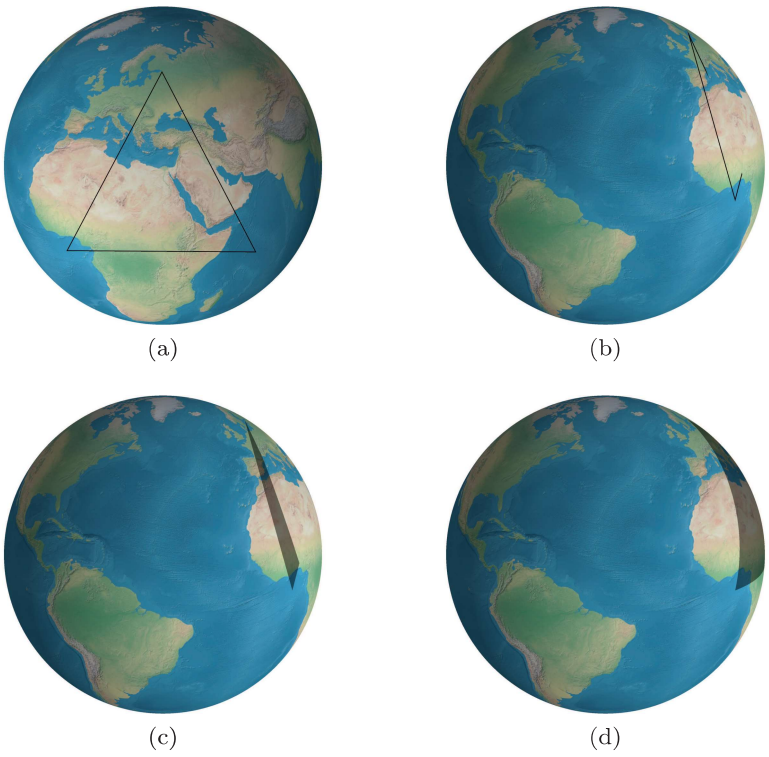


Figure 8.15. (a) From this view, the triangle appears to be on the globe. (b) In reality, only its endpoints are on the globe, as evident in this view. (c) When filled, the triangle is shown to be under the globe and partially clipped. (d) The desired result, which requires subdividing the triangle.

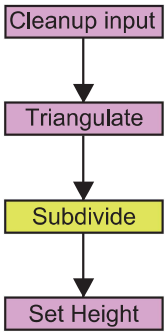


Figure 8.16. The subdivision stage immediately follows triangulation.

granularity (e.g., 1°). If the stopping condition is not satisfied, the triangle is divided into two by bisecting the edge with the largest granularity. This continues recursively until all triangles meet the stopping condition. At this point, the subdivided triangles are still in the same plane as the original triangles. The final stage of our pipeline will raise them to the ellipsoid's surface or above it.

In Section 4.1.2, we presented a recursive implementation to the tetrahedron subdivision algorithm. Here, we will present a queue-based subdivision implementation. The complete code is in `OpenGlobe.Core.TriangleMeshSubdivision`. The method takes positions and indices computed from triangulation and a granularity and returns a new set of positions and indices that satisfy the stopping condition (see Listing 8.7).

```
public class TriangleMeshSubdivisionResult
{
  public ICollection<Vector3D> Positions  {get; }
  public IndicesUnsignedInt Indices { get; }

  // ...
}

public static class TriangleMeshSubdivision
{
  public static TriangleMeshSubdivisionResult Compute(
      IEnumerable<Vector3D> positions,
      IndicesUnsignedInt indices, double granularity)
  {
    // ..
  }
}
```

Listing 8.7. Method signature for subdivision.

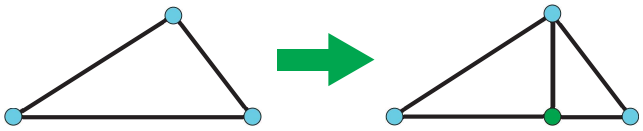


Figure 8.17. Subdividing one triangle into two by bisecting the edge with the largest granularity.

```
public struct Edge : IEquatable<Edge>
{
  public Edge(int index0, int index1) { /* ... */ }

  public int Index0 { get; }
  public int Index1 { get; }
  // ...
}
```

Listing 8.8. An edge of a triangle.

The algorithm works by managing two queues and a list of positions. The `triangle` queue contains triangles that may need to be subdivided, and the `done` queue contains triangles that have met the stopping condition. First, the indices for all input triangles are placed on the `triangles` queue, and the input positions are copied to a new positions list: `subdivided Positions`. While the `triangles` queue is not empty, a triangle is dequeued and the granularity for each edge is computed. If no granularity is larger than the input granularity, the triangle is enqueued on the `done` queue. Otherwise, the center of the edge with the largest granularity is computed and added to `subdividedPositions`. Then, two new subdivided triangles are enqueued onto `triangles`, and the algorithm continues. An example subdivision is shown in Figure 8.17.

This implementation can result in a lot of duplicate positions. For example, consider two adjacent triangles sharing an edge. Since each triangle is subdivided independently, each will subdivide the edge and add the same position to `subdividedPositions`. An easy way to avoid this duplication is to build a cache of subdivided edges and check the cache before subdividing an edge. To implement this, first create a type for a triangle edge. It simply needs to store two indices, as done in Listing 8.8.

The subdivision algorithm then needs a dictonary from an Edge to the index of the position that was created when the edge was subdivided, such as the following:

```
Dictionary<Edge, int> edges = new Dictionary<Edge, int>();
```

Consider a triangle that needs to be subdivided along an edge with indices `triangle.I0` and `triangle.I1`, which correspond to positions `p0` and `p1`. Without edge caching, this code would look as follows:

```
subdividedPositions.Add((p0 + p1) * 0.5);
int i = subdividedPositions.Count - 1;

triangles.Enqueue(
    new TriangleIndicesUnsignedInt(triangle.I0, i, triangle.I2));
triangles.Enqueue(
    new TriangleIndicesUnsignedInt(i, triangle.I1, triangle.I2));
```

With edge caching, first the cache is checked for a subdivided edge with the same indices. If such an edge exists, the index of the subdivided point is retrieved from the cache. Otherwise, the edge is subdivided, the new position is added to `subdividedPositions`, and an entry is made in the cache:

```
Edge edge = new Edge(Math.Min(triangle.I0, triangle.I1),
                     Math.Max(triangle.I0, triangle.I1));
int i;
if (!edges.TryGetValue(edge, out i))
{
  subdividedPositions.Add((p0 + p1) * 0.5);
  i = subdividedPositions.Count - 1;
  edges.Add(edge, i);
}
// ...
```

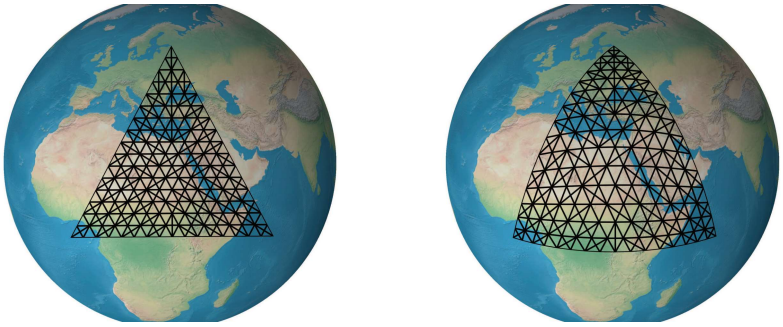
The edge is always looked up using the minimum index as the first argument and the maximum index as the second argument since edge (i_0, i_1) equals (i_1, i_0).

Try implementing an adaptive subdivision algorithm on the GPU using the programmable tessellation stages introduced in OpenGL 4.

◯◯◯◯ Try This

8.2.5 Setting the Height

The final stage of our pipeline is also the simplest stage. Each position from the subdivision stage is scaled to the surface by calling Ellipsoid.`ScaleToGeocentricSurface` (see Section 2.3.2). The wireframe result is shown in Figure 8.18.



(a) (b)

Figure 8.18. (a) The triangle from Figure 8.15(a) after subdivision with a 5° granularity. (b) The same wireframe after scaling all the positions to the surface.

8.2.6 Rendering

Once a triangle mesh is produced from the polygon pipeline, rendering is easy. Polygons are most commonly rendered with translucency, as shown in figures throughout this chapter. This is achieved by using a blend equation of add with a source factor of source_α and a destination factor of $1 - \text{source}_\alpha$.

For lighting, normals for the polygon can be computed procedurally in the fragment shader using the ellipsoid's geodetic surface normals (see Section 2.2.2). This takes advantage of the fact that the polygon is on the ellipsoid; we could not use the ellipsoid's normals for arbitrary polygons in space.

Even though back-face culling is usually only used for closed opaque objects, it can be used when rendering our translucent polygons if the polygon is rendered directly on the ellipsoid, that is, with a height of zero. This eliminates rasterizing fragments for triangles on the back face of the globe.

A complete implementation of the polygon pipeline and rendering is in `OpenGlobe.Scene.Polygon`.

8.2.7 Pipeline Modifications

The polygon pipeline is a subset of what we've used in STK and Insight3D for years. There are additional stages and worthwhile modifications worth considering. For starters, the triangulation stage can be modified to support polygons with holes. Typically, the convention is that the outer boundary has one winding order and boundaries for interior holes have the opposite winding order. The key to triangulating such polygons with ear

clipping is to first convert the polygon to a simple polygon by adding two edges connecting mutually visible points in both directions for every hole. Eberly describes such an algorithm [44].

There are several stages that can be added to improve rendering performance. Multiple LODs can be computed. One approach is to compute discrete LODs after the height-setting stage. For a fast and easy-to-code algorithm, see vertex clustering [106, 146]. For a good balance of speed and visual fidelity, consider quadric error metrics [55]. LODs can also be computed as part of the subdivision step by stopping subdivision early to create lower LODs. The drawback is that every LOD will include all triangles in the original triangulation.

Similar to polylines, polygons in virtual globes are typically rendered as part of a layer that may only be visible for certain viewer heights or view distances. This can make LODs for individual polygons less useful since a polygon's layer may be turned off before the polygon's lower LODs would even be rendered.

To improve rendering throughput, vertices and indices can be reordered into a GPU vertex cache-coherent layout [50, 148]. This stage can occur after subdivision or after LOD computation. If LODs are used, indices for each LOD should be reordered independently.

If the polygon is going to be culled, a bounding-sphere or other bounding-volume computation stage can be added to the pipeline. In many cases, just the boundary points are necessary to determine the sphere. However, one could conceive of polygons on really oblate ellipsoids where this is not true, requiring a robust solution to consider all points after subdivision.

Finally, it is worth observing that performance improvements can be made by combining pipeline stages. Fewer passes over the polygon result in less memory access and fewer cache misses. In particular, it is easy to combine the subdivision stage and the height-setting stage. The bounding-volume stage can also be combined with other stages. Although combining stages can result in a performance win, it comes with the maintenance cost of less clear code.

8.3 Polygons on Terrain

The polygon pipeline is an excellent approach to rendering polygons on an ellipsoidal globe. We've used it successfully in commercial products for many years. What happens, though, if the globe is not a simple ellipsoid but also includes terrain? The polygon's triangle mesh will render under the terrain, or above it in the case of undersea terrain. This is because the triangle mesh approximates the ellipsoid, not the terrain. Here, we introduce modifications to the polygon pipeline and a new way to render

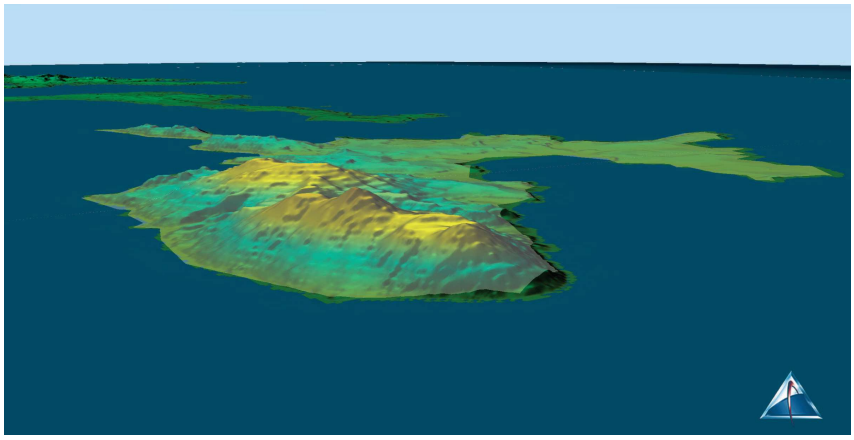


Figure 8.19. Polygon that conforms to terrain rendered using shadow volumes. (Image taken using STK.)

(a) (b)

Figure 8.20. (a) A polygon conforming to terrain using render to texture. Image (C) 2010 DigitalGlobe. (b) Aliasing occurs along the boundary when zoomed in close. Image USDA Farm Service Agency, Image NMRGIS, Image (C) 2010 DigitalGlobe. (Figures taken using Google Earth.)

the triangle mesh based on shadow volumes that make a polygon conform to terrain, as shown in Figures 8.19 and 8.20(a).

8.3.1 Rendering Approaches

Before describing our approach in detail, let's consider a few alternative approaches:

- *Terrain-approximating triangle mesh.* Perhaps the most obvious way to make a triangle mesh conform to terrain is to modify the set-height stage of the pipeline (see Section 8.2.5). That is, instead of raising each position of the mesh to the ellipsoid surface, raise each position

to the terrain's surface. Of course, this requires a function that can return the terrain's height given a geographic position.

Although this approach is simple, it does not produce satisfactory visual quality. Sometimes a polygon's triangles will be coplanar with terrain, and other times, they will be under the terrain creating obvious rendering artifacts. This cannot be resolved by a technique like averaged depth (see Section 7.2) because the triangle mesh needs to be depth tested against the true depth values of the terrain. Furthermore, most terrain is implemented using LOD; when the terrain LOD changes as the viewer zooms, different artifacts will appear.

- *Render to texture.* An alternative approach is to render a polygon to a texture, then render the texture on terrain using multitexturing. As mentioned in Section 8.1, a major drawback to this is that aliasing can occur if the texture is not of sufficient resolution, as shown in Figure 8.20(b). Also, creating the texture can be a slow process, making this approach less attractive for dynamic polygons.

A third approach is based on *shadow volumes* [36, 149]. A triangulated polygon is raised above terrain, duplicated, then lowered below terrain, forming the caps of a closed volume encompassing the terrain, as shown in Figure 8.23(b). Terrain intersecting the volume is shaded using shadow-volume rendering. This approach has several advantages:

- *Visual quality.* No aliasing occurs along the polygon boundary. The intersection of the shadow volume and terrain is accurate to the pixel. As the viewer zooms, the polygon's visual quality does not change.

- *Decoupled from terrain rendering.* Polygons are rendered separately and independently of terrain. It doesn't matter what terrain LOD algorithm is used, or if LOD is used at all. The only requirement is that the polygon's shadow volume needs to enclose the terrain and it needs to be rendered after terrain, when the terrain's depth buffer is available.

- *Low memory requirements.* For many polygons, this approach uses less memory than render to texture. Shadow-volume extrusion can be done in a geometry shader, further reducing the memory requirements.

- *Versatility.* This approach can be generalized to shade models or any geometry intersecting the shadow volume.

8.3.2 Shadow Volumes

Let's first look at shadow volumes in general before detailing how to render polygons with them. One approach to shadow rendering is to use shadow volumes and the stencil buffer to determine areas in shadow [68]. These areas are only shaded with ambient and emission components, while the rest of the scene is fully shaded, including diffuse and specular components.

Consider a point light source and a single triangle. Cast three rays, each from the point through a different vertex of the triangle. The truncated infinite pyramid formed on the side of the triangle opposite of the point light is a shadow volume, as shown in Figure 8.21(a).

Objects inside the shadow volume are in shadow. This is determined on a pixel-by-pixel basis. Imagine casting a ray from the eye through a pixel. A count is initialized to zero. When the ray intersects the front face of a shadow volume, the count is incremented. When the ray intersects the back face of a shadow volume, the count is decremented. If the count is nonzero when the ray finally intersects an object, the object is in shadow. Otherwise, the object is not in shadow (see Figure 8.21(b)). This works for nonconvex volumes and multiple overlapping volumes.

In practice, this is usually implemented with the stencil buffer. The algorithm for rendering the scene is as follows:

- Render the entire scene to the color and depth buffers with just ambient and emission components.
- Disable color buffer and depth buffer writes.
- Clear the stencil buffer and enable the stencil test.
- Render the front faces of all shadow volumes. Each object that casts a shadow will have a shadow volume. During this pass, increment the stencil value for fragments that pass the depth test.

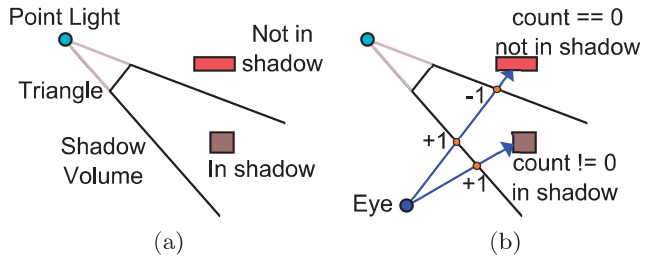


Figure 8.21. (a) A side view of a shadow volume formed from a point light and a triangle. Objects inside the volume are in shadow. (b) Conceptually, rays are cast from the eye through pixels to determine if an object is in shadow.

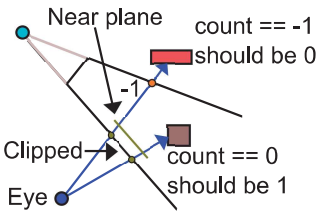


Figure 8.22. Z-pass shadow volumes lead to an incorrect count when shadow volumes are clipped by the near plane.

- Render shadow-volume back faces, decrementing the stencil value for fragments that pass the depth test.
- At this point, the stencil buffer is nonzero for pixels in shadow.
- Enable writing to the color buffer and set the depth test to less than or equal to.
- Render the entire scene again with additive diffuse and specular components and a stencil test that passes when the value is nonzero. Only fragments not in shadow will be shaded with diffuse and specular components.

Incrementing and decrementing the stencil value when rendering the shadow-volume front and back faces, respectively, implements casting a ray through the scene to determine areas in shadow. Although this is described using two passes, it can be implemented in a single pass using two-sided stencil, which allows different stencil operations for front- and back-facing triangles rendered in the same pass. This is a core feature since OpenGL 2.0.

The algorithm as described above is called z-pass because the stencil value is modified when a fragment passes the depth test. This algorithm fails when the viewer is inside the volume or the near plane intersects it. Since part of the volume is clipped by the near plane, the count becomes wrong, leading to an incorrect shadow test, as shown in Figure 8.22. The problem can be moved to the far plane, where it can be dealt with more easily by using z-fail shadow volumes [15, 89]. With z-fail, the front- and back-facing shadow-volume rendering passes are changed to:

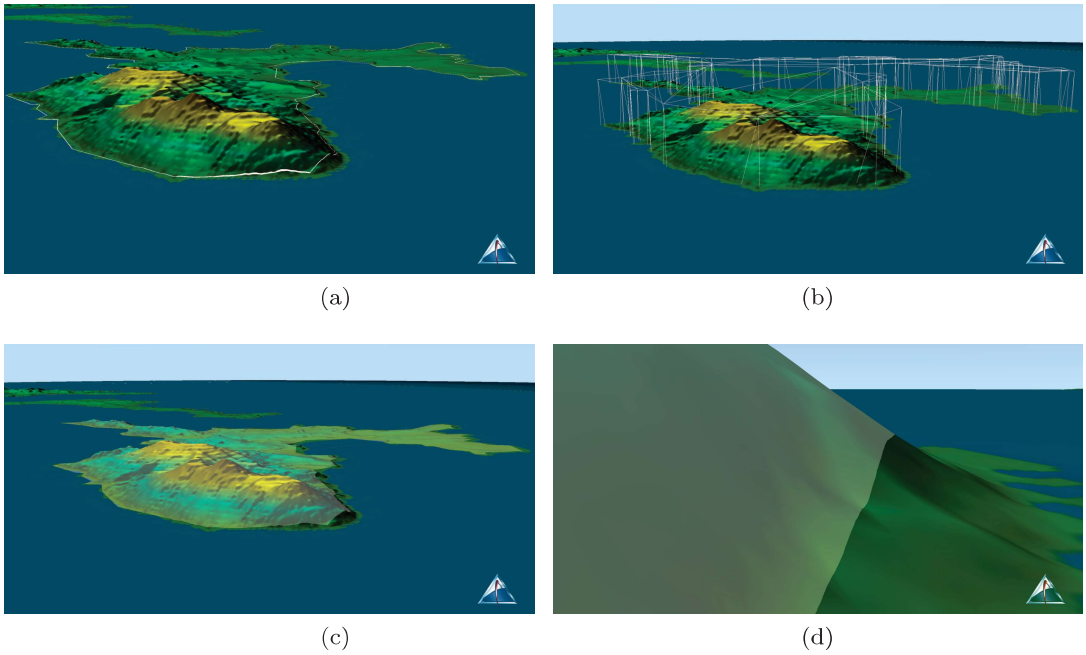
- render shadow-volume back faces, incrementing the stencil value if the fragment fails a less-than depth test;
- render shadow-volume front faces, decrementing the stencil value if the fragment fails a less-than depth test.

This also requires that shadow volumes be closed at their far ends. An incorrect count can still be computed if the shadow volume is clipped by the far plane. This can be solved in hardware using depth clamping, a core feature since OpenGL 3.2. Depth clamping allows geometry clipped by the near or far plane to be rasterized and writes the depth value clamped to the far depth range.

8.3.3 Rendering Polygons Using Shadow Volumes

Since shadow volumes are used for rendering shadows, how do we use them to render polygons that conform to terrain? There are two key insights:

- Shadow volumes are used to determine the area in shadow. A shadow volume can be constructed however we choose; in fact, it could be a volume encompassing a polygonal region of terrain, like that shown in Figure 8.23(b).
- The area in shadow is only shaded with ambient and emission components, but we can actually shade areas in shadow however we please.



(a) (b)

(c) (d)

Figure 8.23. (a) The outline of a polygon is shown in white. (b) The outline is triangulated and extruded to form a shadow volume intersecting terrain. (c) The polygon is rendered filled on terrain using a shadow volume. (d) Zooming in close on the polygon does not create aliasing artifacts. (Images taken using STK.)

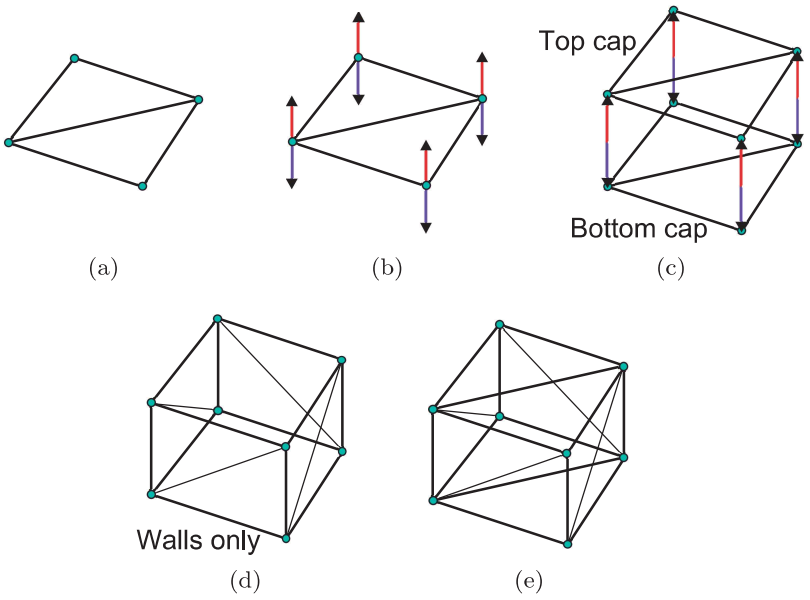


Figure 8.24. (a) A triangulated polygon before entering the shadow-volume creation stage. (b) Each vertex's normal and its opposite. (c) The top and bottom caps are formed by extruding each vertex. (d) A triangle strip forms a wall connecting the top and bottom cap. (e) The polygon's complete shadow volume with top and bottom caps and a wall.

> We can render the globe and terrain as usual, build shadow volumes around polygons that encompass the terrain, then shade the area "in shadow," that is, the terrain inside the volume, by modulating the color buffer with that of the polygon's color. The result is shown in Figure 8.23(c).

The algorithm starts with a polygon's boundary, like that shown in Figure 8.23(a). It then goes through every stage of our polygon pipeline except the last: input cleanup, triangulation, and subdivision. The final stage, set height, is replaced with a stage that builds the shadow volume.

In this stage, every position is raised along the ellipsoid's geodetic surface normal, forming a mesh above the terrain. This is the top cap of the shadow volume. Likewise, every vertex is lowered along the direction opposite the geodetic surface normal, forming the bottom cap below terrain. Finally, a wall is created along the polygon's boundary that connects the bottom cap and top cap, forming a closed shadow volume. This process is shown in Figure 8.24.

Once a shadow volume is created for a polygon, rendering the polygon is similar to the shadow-volume rendering algorithm previously described. The polygon can be rendered on a globe with terrain or just an ellipsoidal globe. All that is required is that the depth buffer be written before the polygons are rendered. The steps are as follows:

- Render the globe and terrain to the color and depth buffers.
- Disable color buffer and depth buffer writes (same as Section 8.3.2).
- Clear the stencil buffer and enable the stencil test (same as Section 8.3.2).
- Render the front and back faces of each polygon's shadow volume to the stencil buffer using either the z-pass or z-fail algorithm described previously (same as Section 8.3.2).
- At this point, the stencil buffer is nonzero for globe/terrain pixels covered by a polygon.
- Enable writing to the color buffer and set the depth test to less than or equal to (same as Section 8.3.2).
- Render each shadow volume again with a stencil test that passes when the value is nonzero. The most common way to shade the fragments is to use blending to modulate the polygon's color with the globe/terrain color, based on the polygon's alpha value.

Unlike using shadow volumes for shadows, using shadow volumes for polygons does not require rendering the entire scene twice. Instead, the final pass renders just polygon shadow volumes, not the full scene. If all polygons are the same color, performance can be improved by replacing the second pass with a screen-aligned quad. Doing so reduces the geometry load and can reduce fill rate because the depth complexity of the shadow volumes can be high, especially for horizon views, whereas the depth complexity of the quad is always one.

8.3.4 Optimizations

Rendering polygons with shadow volumes uses a great deal of fill rate. In particular, there is a large amount of wasted rasterization for fragments that do not pass the stencil test on the final pass. A ray cast through these pixels goes through the shadow volume(s) without intersecting the terrain or globe. When looking straight down at the globe, there is no wasted rasterization overhead, but horizon views can have a large amount of overhead, as shown in Figure 8.25.

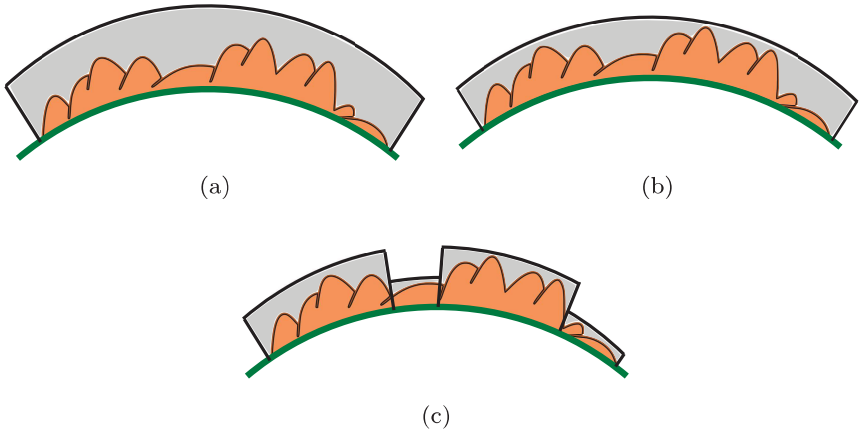


Figure 8.25. Shadow-volume rasterization overhead is shown in gray. (a) Using the globe's maximum terrain height for the shadow volume's top cap results in the most wasted rasterization. (b) Using the maximum terrain height of terrain encompassed by the polygon can decrease rasterization overhead. (c) Breaking the shadow volume into sections and using the maximum terrain height in each section can further reduce rasterization overhead.

One way to minimize the amount of rasterization overhead is to use tight-fitting shadow volumes. The simplest way to extrude the top and bottom caps is to move the top cap up along the maximum terrain height of the globe and to move the bottom cap down along the minimum terrain height. This can create a great deal of waste, creating tall, thin shadow volumes that yield very few shaded pixels for horizon views. A more sophisticated approach stores minimum and maximum terrain heights in a grid or quadtree covering the globe. The minimum and maximum heights for terrain approximately covered by a polygon is used to determine the height for the bottom and top caps. This can still lead to lots of wasted rasterization when terrain has high peaks and low valleys. To combat this, a shadow volume can be broken into sections, each with a different top cap and bottom cap height. Using multiple section trades increases vertex processing for decreased fill rate.

The amount of vertex memory required can be reduced by using a geometry shader. Instead of creating the shadow volume on the CPU, it can be created on the GPU using the original polygon triangulation. The geometry shader outputs the extruded top cap and bottom cap and the connecting wall. An additional benefit of this approach is that if terrain LOD requires the shadow volume to dynamically change its height, it can do so on the GPU.

Just like normal meshes, the shadow-volume mesh can be optimized for GPU vertex caches and have LOD applied. In addition, when the viewer is sufficiently far away, the shadow volume can be replaced with a simple mesh floating above terrain. At this view distance, the extra fill rate caused by shadow volumes usually isn't a problem, but the number of render passes is reduced by not using shadow volumes.

8.4 Resources

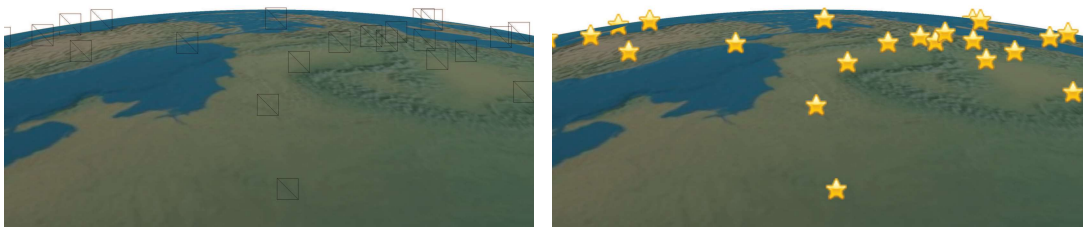
There are many discussions on triangulation on the web [44, 69–71, 127]. *Real-Time Rendering* contains an excellent survey of shadow volumes in general [3]. In recent years, using shadow volumes to render polylines and polygons on terrain has received attention in the research literature [36, 149].

9

Billboards

Screen-aligned *billboards* are textured quads rendered parallel to the viewport so that they are always facing the viewer. To support nonrectangular shapes, these quads are rendered with a texture containing an alpha channel, in combination with discard in the fragment shader (see Figure 9.1).[1]

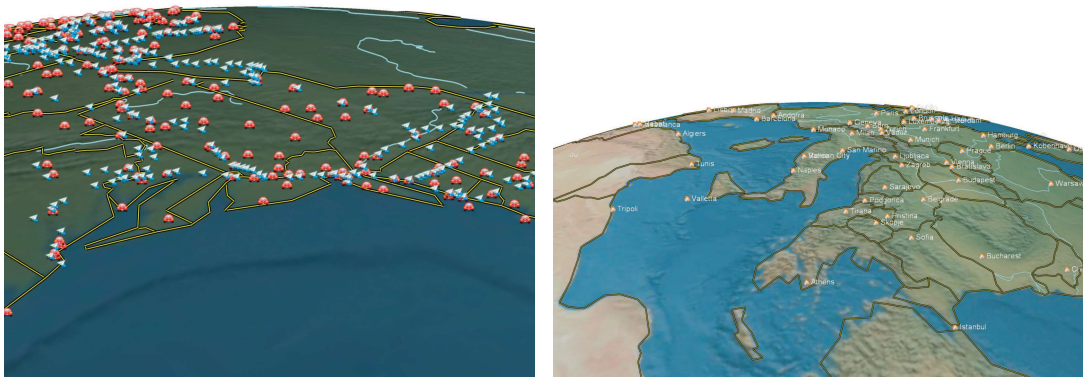
Billboards used in this fashion are also called *sprites* and serve many important purposes in virtual globe rendering, the most prominent of which is rendering icons for places of interest: cities, landmarks, libraries, hospitals, etc.—pretty much anything interesting whose position can be defined or approximated by a single geographic position (see Figure 9.2(a)). The position doesn't always need to be static; icons may be used for the current position of an airplane in flight. Most virtual globes even allow users to place icons at positions of their choosing (e.g., Google Earth's placemarks and ArcGIS Explorer's point notes).



(a) (b)

Figure 9.1. (a) Billboards are rendered using two screen-aligned triangles that form a quad. (b) A texture's alpha channel is used to discard transparent fragments.

[1] The alpha test could be used in place of discard, but it was deprecated in OpenGL 3.



(a) (b)

Figure 9.2. (a) Billboards used to render icons for airports and Amtrak stations in the United States. (b) Billboards used to render icons and labels for cities in Europe.

In addition to icons, billboards are also used to render text facing the viewer, as in Figure 9.2(b). In many cases, point features contain metadata, including a label that is displayed alongside its icon. Another type of text rendering seen in virtual globes is labels burned into imagery (e.g., the names of roads overlaid on satellite imagery). This type of text becomes hard for the user to read when the viewer is looking at an edge.

Billboards, in all their varieties, can serve many other uses: rendering vegetation, replacing 3D models with lightweight *imposters*, and rendering a wide range of image-based effects, including clouds, smoke, and fire. Our focus is on using billboards to render icons and text from GIS datasets.

9.1 Basic Rendering

Let's put aside textures for now and start our billboard-rendering algorithm by focusing on rendering screen-aligned quads of constant pixel size. The key observation is that since a billboard's position is defined in model (or world) coordinates and its size is defined in pixels, the size of a billboard in model space changes as the viewer zooms in and out. For example, a 32×32 pixel billboard for an airport is only a few meters large when zoomed in at street level. As the viewer zooms out, the size of the billboard in pixels does not change, but its size in meters increases. This means that when the viewer moves, our rendering algorithm needs to recompute the model-space size of every visible billboard in order to maintain its constant pixel size.

The basic approach is to transform the billboard's center point from model to window coordinates, then use simple 2D offsets, scaled by the bill-

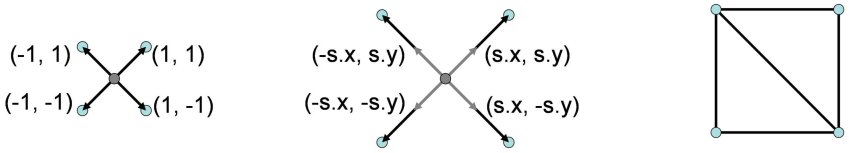


Figure 9.3. (a) Once transformed to window coordinates, a billboard's position is copied and moved towards each of the quad's corners. (b) The translation is scaled by one-half the texture's size, s, to compute the actual corner point. (c) The result is modeled with a two-triangle triangle strip.

board's texture size, to compute the corner points for a viewport-aligned quad. The quad, modeled with a two-triangle triangle strip, is then rendered using an orthographic projection (see Figure 9.3). Working in window coordinates makes it easy to align the billboard with the viewport and maintain its constant pixel size. The process is very similar to rendering lines of a constant pixel width discussed in Section 7.3.3. There are a variety of options for where exactly to do the transforms:

- *CPU.* Before programmable hardware, it was common to transform each billboard on the CPU, then render a quad either one at a time using OpenGL immediate mode or in an all-in-one call using client vertex arrays. On today's hardware this is horribly inefficient. In particular, it results in a lot of per-billboard CPU overhead, system bus traffic, and system-memory cache misses in the common case of rendering a large number of billboards. Furthermore, when the CPU has to touch each billboard and send it to the GPU, the CPU is not able to feed the GPU quickly enough to fully utilize it. The increasing gap between memory bandwidth and processing power will only make this approach worse over time. There must be a better way! In fact, since each billboard can be transformed independently of the others, a GPU implementation is ideal.

- *Point sprites.* In order to utilize the GPU for billboard rendering, both OpenGL and Direct3D exposed a fixed-function feature called *point sprites* [33]. Point sprites generalize point rendering to allow points to be rasterized as textured squares on the GPU, eliminating the need for per-billboard transforms on the CPU and significant system bus traffic, and only requiring one-fourth the amount of memory. Although point sprites can provide a significant performance improvement, they are less commonly used today in favor of the more flexible

approach of using shaders. In fact, fixed-function point sprites were removed in Direct3D 10.

- *Vertex shader.* Billboards are straightforward to implement in a vertex shader. Each billboard is represented by two triangles forming a quad defined by four vertices. Each vertex stores the billboard's model position and an offset vector of either $(-1, -1)$, $(1, -1)$, $(1, 1)$, or $(-1, 1)$ (see Figure 9.4). The triangles are degenerate until the vertex shader transforms the billboard's model position into window coordinates and uses the offset vector and texture size to translate the vertex to its corner of the quad. Similar to point sprites, this moves all the transforms to the GPU, but it uses four times the number of vertices as point sprites and requires larger vertices due to the offset vector. Fortunately, the geometry shader can be used to keep the flexibility of a shader-based approach without the need to duplicate vertices.

- *Geometry shader.* Billboard rendering is the quintessential geometry-shader example, which is why it is mentioned in both the NVIDIA and ATI programming guides (under the name point sprites) [123, 135]. Its implementation is basically fixed-function point sprites meet vertex-shader billboards. A point is rendered for each billboard, and the geometry shader transforms each point into window coordinates and outputs a viewport-aligned quad. This has all of the flexibility of the vertex-shader version, with only one-fourth the number of vertices, and no need for offset vectors. In a minimal implementation, each billboard only needs its position stored in the vertex buffer. In addition, the geometry shader reduces vertex setup costs, eliminates duplicate model-to-window transforms, and does not require any index data, making it the most memory-efficient, flexible way to render billboards.

Patrick Says ○○◯◯

I've had the pleasure of implementing billboard rendering using CPU, vertex-shader, and geometry-shader implementations in Insight3D. The geometry shader was the easiest to implement (with the exception that debugging the CPU version was the easiest) and is also the most efficient on today's hardware. Interestingly enough, the vertex-shader version outperformed the geometry-shader version on hardware with first-generation geometry-shader support. I resisted the urge to remove the geometry-shader version in hopes of it becoming faster, and sure enough, it did!

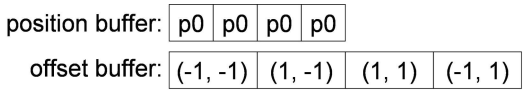


Figure 9.4. The vertex buffer layout for a single billboard rendered with the vertex-shader method. The billboard's position, $p0$, is duplicated across each vertex for a billboard.

Given the strength of the geometry-shader version, let's take a closer look at its implementation. We'll build up to the shaders used in `OpenGlobe.Scene.BillboardCollection`. The vertex data are simply one vertex per billboard containing the billboard's model-space position. A vertex shader is used to transform the position to window coordinates (see Listing 9.1). This transform could also be done in the geometry shader, but for best performance, per-vertex operations should occur in the vertex shader so shared vertices are not redundantly processed in the geometry shader. When rendering point primitives, as we are here, this rarely matters, but we do it regardless for illustration purposes.

The geometry shader, shown in Listing 9.2, is where the bulk of the work happens. Its purpose is to expand the incoming point to a quad. This takes advantage of the fact that the input primitive type does not need to be the same as the output primitive type. In this case, a point is expanded

```
in vec4 position;

uniform mat4 og_modelViewPerspectiveMatrix;
uniform mat4 og_viewportTransformationMatrix;

vec4 ModelToWindowCoordinates(
    vec4 v,
    mat4 modelViewPerspectiveMatrix,
    mat4 viewportTransformationMatrix)
{
  v = modelViewPerspectiveMatrix * v; // clip coordinates
  v.xyz /= v.w; // normalized device coordinates
  v.xyz = (viewportTransformationMatrix *
           vec4(v.xyz, 1.0)).xyz; // window coordinates
  return v;
}

void main()
{
  gl_Position =
      ModelToWindowCoordinates(
          position,
          og_modelViewPerspectiveMatrix,
          og_viewportTransformationMatrix);
}
```

Listing 9.1. Vertex shader for billboard rendering.

```
layout(points) in;
layout(triangle_strip, max_vertices = 4) out;

out vec2 fsTextureCoordinates;

uniform mat4 og_viewportOrthographicMatrix;
uniform sampler2D og_texture0;

void main()
{
  vec2 halfSize = 0.5 * vec2(textureSize(og_texture0, 0));

  vec4 center = gl_in[0].gl_Position;

  vec4 v0 = vec4(center.xy - halfSize, -center.z, 1.0);
  vec4 v1 = vec4(center.xy + vec2(halfSize.x, -halfSize.y),
                 -center.z, 1.0);
  vec4 v2 = vec4(center.xy + vec2(-halfSize.x, halfSize.y),
                 -center.z, 1.0);
  vec4 v3 = vec4(center.xy + halfSize, -center.z, 1.0);

  gl_Position = og_viewportOrthographicMatrix * v0;
  fsTextureCoordinates = vec2(-1.0, -1.0);
  EmitVertex();

  gl_Position = og_viewportOrthographicMatrix * v1;
  fsTextureCoordinates = vec2(1.0, -1.0);
  EmitVertex();

  gl_Position = og_viewportOrthographicMatrix * v2;
  fsTextureCoordinates = vec2(-1.0, 1.0);
  EmitVertex();

  gl_Position = og_viewportOrthographicMatrix * v3;
  fsTextureCoordinates = vec2(1.0, 1.0);
  EmitVertex();
}
```

Listing 9.2. Geometry shader for billboard rendering.

into a triangle strip. Thankfully, all triangle strips will be composed of four vertices, one for each corner point, so it is easy to explicitly set the output's upper bound (`max_vertices` = 4). If the number of vertices is variable, it is important to keep the number as low as possible, for best performance on NVIDIA hardware [123].[2]

In order to expand the point into a triangle strip, first the geometry shader determines the size of the billboard's texture, in pixels, using the `textureSize` function. This value is divided by two and used to create four new vertices, one for each corner of the quad. These vertices are then output as a triangle strip with clockwise winding order. The winding order is not particularly important here, however, because the triangle strip is

[2] ATI, on the other hand, states "the ATI Radeon HD 2000 series is largely insensitive to the declared maximum vertex count" (p. 9) and says it is OK to set this value to a safe upper bound when it is hard to determine at compile time [135].

always facing the viewer, so facet culling provides no benefit. Also, observe that the texture coordinates are easily generated on the fly when the vertex is output.

Since geometry shaders must preserve the order in which geometry was rendered, the GPU buffers geometry-shader output to allow several geometry-shader threads to run in parallel. If a geometry shader does a significant amount of amplification and, in turn, outputs a large number of vertices, the output can overflow from on-chip buffers to slow off-chip memory, which is why geometry-shader performance is commonly bound by the size of its output. Our geometry-shader implementation does not do too much amplification. Given it inputs one vertex and outputs four, its amplification rate is 1 : 4, which just so happens to be a rate that ATI hardware includes special support for [135]. See their programming guide for additional restrictions.

The output size of a geometry shader is commonly measured in number of scalar components. In our geometry shader, each vertex output is composed of a four-scalar position and a two-scalar texture coordinate, for a total of six scalars per vertex. Since we are outputting four vertices, the total number of scalars output is 24. The lower this number, the faster the geometry shader will run. On an NVIDIA GeForce 8800 GTX, a shader runs at maximum performance when outputting 20 or less scalars. Performance then degrades in a nonsmooth fashion as the number of scalars increases. For example, a shader runs at 50% performance when outputting 27 to 40 scalars. Therefore, it is important to minimize the number of scalars output in the geometry shader even if it calls for doing additional computation in the fragment shader.

```
in vec2 fsTextureCoordinates;
out vec4 fragmentColor;

uniform sampler2D og_texture0;
uniform vec4 u_color;

void main()
{
  vec4 color = u_color * texture(og_texture0,
                                 fsTextureCoordinates);

  if (color.a == 0.0)
  {
      discard;
  }
  fragmentColor = color;
}
```

Listing 9.3. Fragment shader for billboard rendering.

Let us turn our attention to the fragment shader in Listing 9.3. In most virtual globes, lighting does not influence billboard shading, which makes fragment shading quite simple. In the simplest case, the texel from the billboard's texture is assigned as the output color. Since we want to support nonrectangular shapes, our shader also discards fragments based on the texture's alpha channel. In addition, the texel isn't used directly. Instead, it is modulated with a user-defined color, shown here as a uniform, but it could also be defined per vertex and passed through to the fragment shader. In many cases, this will be white to let the texel come through as is. This color is most often used when rendering billboards with white textures that should be shaded a different color. This eliminates the need for creating different color versions of the same texture.

These vertex, geometry, and fragment shaders are all that are needed for a basic billboard implementation. However, there are lots of things that will make our billboards more useful. Let's start with efficiently rendering a large number of billboards with different textures.

9.2 Minimizing Texture Switches

For best performance, we want to render as many billboards as possible using the fewest number of draw calls: ideally, only one. Batching positions for several billboards into a single vertex buffer usually isn't a problem, but it is only half the batching battle. How do you assign a potentially different texture to each billboard? The last thing you want is code like in Listing 9.4.

Here, the need to bind a new texture reduces the number of billboards that can be rendered with a single draw call. This problem extends well beyond billboard rendering; NVIDIA conducted an internal survey of four Direct3D 9 games and found `SetTexture` to be the most frequent render-state change or "batch breaker" [121]. The code above shows a worst case scenario; only a single billboard is rendered per draw call. The fact that the first texture is rebound for the third draw call hints toward one technique for improving batch size: sorting by texture. Let's consider this and other techniques:

- *Sorting by texture.* Sorting by state, whether it be shader, vertex array, texture, etc., is a common technique for minimizing driver CPU overhead and exploiting the parallelism of the GPU (see Section 3.3.6). In our code example, sorting by texture would allow us to combine the first and third draw calls. If the vertex buffer isn't also rearranged, a call like `glMultiDrawElements` can be used to render

```
context.TextureUnits[0].Texture2D = Texture0;
context.Draw(PrimitiveType.Points, 0, 1, _drawState,
    sceneState);
context.TextureUnits[0].Texture2D = Texture1;
context.Draw(PrimitiveType.Points, 1, 1, _drawState,
    sceneState);
context.TextureUnits[0].Texture2D = Texture0;
context.Draw(PrimitiveType.Points, 2, 1, _drawState,
    sceneState);
// ...
```

Listing 9.4. Texture binds inhibiting batching of billboards.

multiple sets of indices in a single call. If there are only a handful of unique textures and lots of billboards, sorting by texture can result in large batch sizes.

Be wary of sorting every frame, as the added CPU overhead can drown out the performance gains of batching. Ideally, sorting occurs once at initialization. However, this makes dynamically changing a billboard's texture more difficult. Another drawback to sorting is if a large number of unique textures is used, an equally large number of draw calls is required.

- *Texture atlases.* An alternative to sorting by texture is to use a single large texture, called a *texture atlas*, containing all the images for a given batch, as shown in Figure 9.5. To avoid texture-filtering artifacts, each image is separated by a small border. Of course, texture

Figure 9.5. A texture atlas composed of seven images.

coordinates need to be computed and stored for each billboard, as they no longer simply range from $(0, 0)$ to $(1, 1)$. Besides providing better texture memory management and coherence than sorting by texture, texture atlases allow for very large batches. The texture atlas is bound once, then a single draw call is made. A downside is texture coordinates can no longer be procedurally generated in the geometry, as done in Listing 9.2. Instead, texture coordinates bounding a subrectangle of the texture atlas are stored per billboard in the vertex buffer. Also, the texture atlas itself should be organized such that it doesn't contain a large amount of empty space. A downside to texture atlases is that texture addressing modes that use coordinates outside $[0, 1]$ need to be emulated in a shader [121]. Fortunately, this is not a show-stopper for virtual globe billboards.

It is also important to keep mipmapping in mind when using a texture atlas. Generating a mipmap chain based on the entire atlas with a call like `glGenerateMipmap` will make images bleed together at higher levels of the mip chain. Instead, the mipmap chain should be generated individually for each image, then the texture atlas should be assembled. Other mipmapping considerations are covered in *Improve Batching Using Texture Atlases* [121].

- *Three-dimensional textures and texture arrays.* A third technique for minimizing texture switches is to use a 3D texture, where each 2D slice stores a unique image. Similar to texture atlases, this requires only a single texture bind and draw call. An advantage over texture atlases is that only a single texture-coordinate component needs to be stored per billboard, since this component can be used as the third texture coordinate to look up a 2D slice. For us, the most prominent downside to this technique is that all the 2D slices need to be the same dimensions.

  Another downside is this technique is incompatible with mipmapping. The entire 3D volume is mipmapped, as opposed to each individual 2D slice. Because of this, a feature called *array textures* was added to OpenGL [22,90]; a similar feature called texture arrays was added to Direct3D 10. This is basically a 3D texture where each 2D slice has its own mip chain, with no filtering between slices. So mipmapping works as expected for each individual slice. Array textures can also be 2D, storing a set of individually mipmapped 1D textures.

Since our billboards maintain a constant pixel size, mipmapping is not needed.[3] Therefore, the advantages of array textures over 3D textures are

[3] If billboards are sized dynamically, perhaps based on their distance to the viewer,

not important to us. Sorting by texture can lead to lots of small batches when many textures are used, and it is hard to come up with an efficient and flexible implementation. For us, the contest is between texture atlases and 3D textures. Since each 2D slice in a 3D texture needs to be the same size, either each billboard texture needs to be the same size or additional per-billboard texture coordinates need to be stored to access the subrectangle of a slice. If each slice is not packed with multiple images, using 3D textures can waste a lot of memory if the images vary significantly in size. For our needs, the most flexible approach is to use texture atlases.

9.2.1 Texture-Atlas Packing

Before changing our shaders to support texture atlases, we need to implement code to create a texture atlas. Texture-atlas creation can be done during initialization, at runtime in a separate resource-preparation thread, or as part of an offline tool-chain process, common in games. NVIDIA provides a tool for this purpose.[4] In virtual globes, it is not always known in advance what images will be used to create a texture atlas, so we'll focus on designing a class that can be used at runtime. Specifically, given a list of input images, we want to create a single atlas containing all the images and texture coordinates for each of the input images (see Listing 9.5).

Example input images and the output atlas and texture coordinates are shown in Figure 9.6.

A primary concern in texture-atlas creation is minimizing empty space in the new texture atlas. Naïve algorithms can result in a significant amount of wasted space. For example, consider the simple approach of making the atlas as high as the highest input image and as wide as the sum of each image's width. If one image is tall and narrow and every other image is short and wide, the majority of the atlas will be wasted space! Also, this approach will quickly overrun the OpenGL maximum texture-width limitation,[5] requiring more texture atlases be created.

Minimizing the amount of wasted space in a texture atlas is an application of the *2D rectangle bin-packing problem*, which turns out to be NP-hard [54]. Here, we will outline a simple heuristic-based greedy algorithm that is easy to implement; runs quickly; doesn't have input image size restrictions, such as requiring power of two dimensions or squares; and doesn't require a shader to later rotate texture coordinates. The resulting atlas's wasted space is far from minimal, but this algorithm will perform

mipmapping is useful. For example, as the viewer zooms out, the billboard may shrink, making mipmapping useful.

[4] http://developer.nvidia.com/legacy-texture-tools

[5] The OpenGL maximum texture width and height are usually at least 8,192 texels on today's hardware.

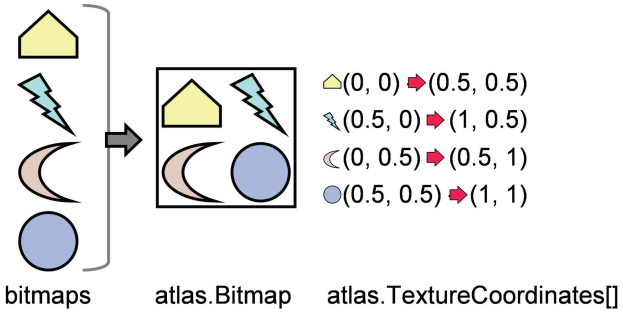


Figure 9.6. Four images (left) used to create a texture atlas (center) and texture coordinates for the original images (right). In the coordinates for the .NET Bitmap, `y` increases from the top, as opposed to OpenGL, where `y` increases from the bottom.

well on common virtual globe inputs, especially in the easy cases, for example, when all input images are 32×32. Full source code is in `OpenGlobe.Renderer.TextureAtlas`.

The first step of our packing algorithm is to determine the width for the texture atlas. We'd like the atlas to be reasonably square so we don't run into an OpenGL width or height limitation when the other dimension still has plenty of room. Our approach doesn't guarantee the atlas is square but provides sensible results for most realistic inputs. We simply take the square root of the sum of the areas of each input image, including a border (see Listing 9.6).

This function also tracks the maximum input image's width and is sure to create an atlas with at least this width. Next, we sort the input images in descending order by height. The result of sorting an example set of input

```
IList<Bitmap> bitmaps = new List<Bitmap>();
bitmaps.Add(new Bitmap("image0.png"));
bitmaps.Add(new Bitmap("image1.png"));
// ...

TextureAtlas atlas = new TextureAtlas(bitmaps);
Texture2D texture = Device.CreateTexture2D(atlas.Bitmap,
    TextureFormat.RedGreenBlueAlpha8, false);

// image0 has texture coordinate atlas.TextureCoordinates[0]
// image1 has texture coordinate atlas.TextureCoordinates[1]
// ...
atlas.Dispose();
```

Listing 9.5. Using the TextureAtlas class.

```
private static int ComputeAtlasWidth(IEnumerable<Bitmap> bitmaps,
                                     int borderWidthInPixels)
{
  int maxWidth = 0;
  int area = 0;
  foreach (Bitmap b in bitmaps)
  {
    area += (b.Width + borderWidthInPixels) *
            (b.Height + borderWidthInPixels);
    maxWidth = Math.Max(maxWidth, b.Width);
  }

  return Math.Max((int)Math.Sqrt((double)area),
                  maxWidth + borderWidthInPixels);
}
```

Listing 9.6. Computing the width for a texture atlas.

images is shown in Figure 9.7(b). At expected $\mathcal{O}(n \log n)$ complexity,[6] this is theoretically the most expensive part of our algorithm, given that every other pass is $\mathcal{O}(n)$ with respect to the number of input images. In practice, the bottleneck can be memory copies from the input images to the atlas, especially if the number of images is small and each image is large.

Now that the images are sorted by height, we make a pass over the images to determine the atlas's height and subrectangle bounds for each image. Starting at the top of the atlas and proceeding left to right, we iterate over the images in sorted order and temporarily save subrectangles bounds for each as if we're copying the image to the atlas without overlapping previous images (see Figures 9.7(c)–9.7(f)). When a row of images exceeds the atlas's width, a new row is created, and the process continues until all images have been accounted for. For a given row, the height is determined by the height of the first image since that will be the tallest image, given the images were sorted descending by height. The height of the atlas is the sum of the height of the rows.

Now that we know the width and height of the atlas, the atlas can be allocated. In a final pass, the images are copied to the atlas using the subrectangle bounds computed in the previous pass, and texture coordinates are computed using a simple linear mapping. The final atlas for the images in Figure 9.7(a) is shown in Figure 9.7(g). This implementation stores the atlas in a Bitmap so it is suitable for offline use (e.g., writing to disk) or runtime use (e.g., as the source for a texture). If atlas creation will be done exclusively at runtime, a Texture2D can be allocated instead of a Bitmap, and each subrectangle can be uploaded individually.

[6] In .NET, List<T>.Sort is implemented using Quicksort. For many inputs, it may be worth coding a radix sort, which runs in $\mathcal{O}(nk)$; in our case, n is the number of images and k is the number of digits required for their height. In most cases, k will only be two or three.

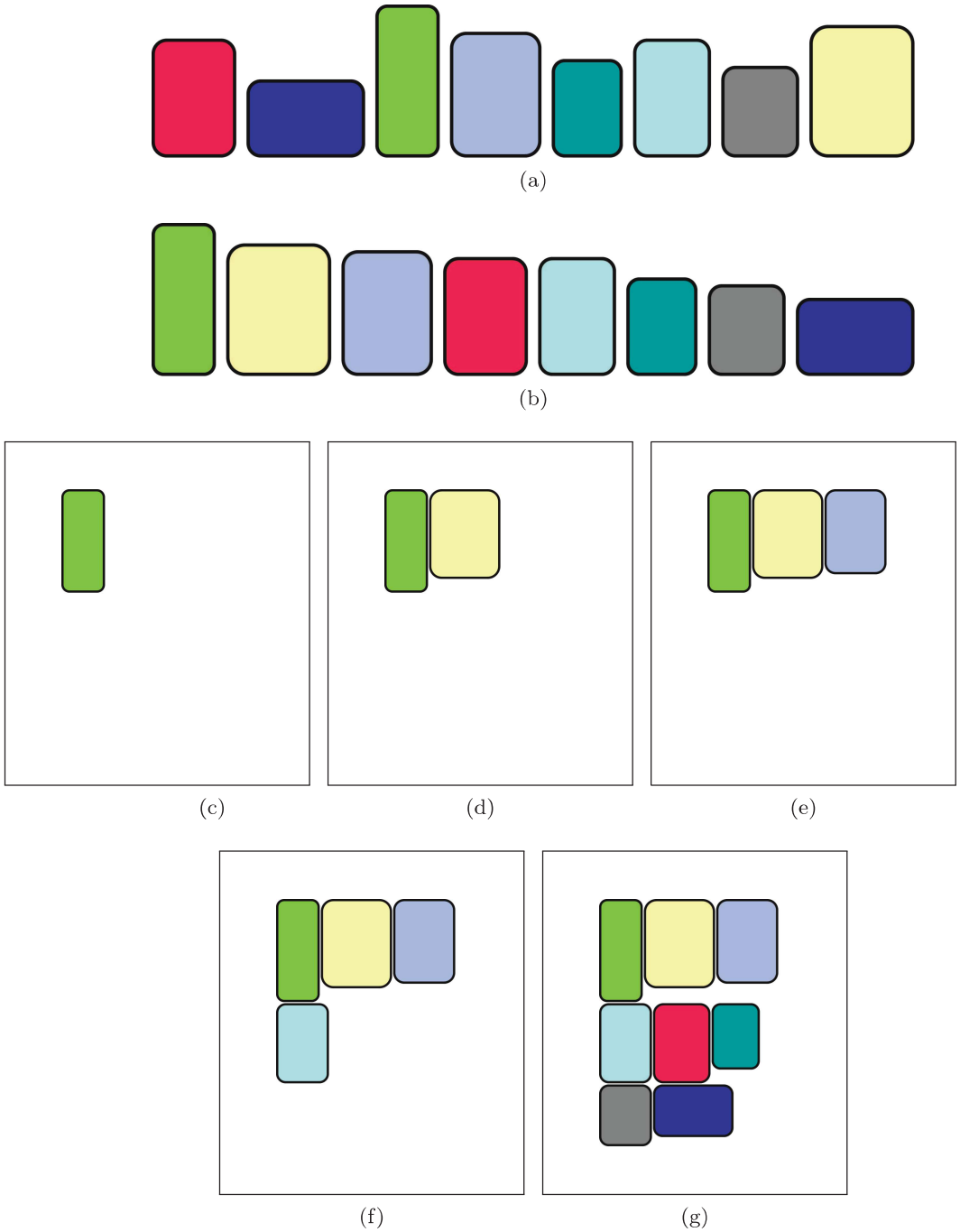


Figure 9.7. (a) A randomly ordered set of images. (b) The images sorted descending by height. (c)–(f) The first four steps of adding images to the atlas. (g) The final atlas.

Modify `OpenGlobe.Renderer`.TextureAtlas to have the option to write the atlas to a Bitmap or Texture2D object.

◯◯◯◯ Try This

An atlas with less wasted space can be created in a few ways, the simplest of which is to trim the width of the atlas so it is flush against the right most subrectangle. Our algorithm can also be extended to the approach presented by Igarashi and Cosgrove [75]. Before sorting the input images, images that are wider than they are tall are rotated 90° so all images are "stood up." The images are then sorted by height and placed in the atlas starting at the top row. The first row is packed left to right, the second row is packed right to left, and so on. As images are added, they are "pushed upward" in the atlas to fill any empty space between rows, like the empty space between the first two rows in Figure 9.7(g). When rendering with a texture atlas created in this fashion, the texture coordinates for images that were rotated also need to be rotated.

There are other alternatives for creating texture atlases. In some cases, texture atlases are created by hand, but that is rarely an option for virtual globes given their dynamic nature. Scott uses recursive spatial subdivision to pack light maps, which is essentially the same as creating texture atlases [151]. John Ratcliff provides C++ code for an approach that selects images with the largest area and largest edge first, filling the atlas starting at the bottom left using a free list and rotating images 90° if needed [143]. For comprehensive coverage of 2D rectangle bin-packing algorithms, see Jylänki's survey [78].

Modify `OpenGlobe.Renderer`.TextureAtlas to implement the packing algorithm described by Igarashi and Cosgrove [75] or another approach described above. Compare the amount of wasted space between your implementation and the original implementation for various inputs.

◯◯◯◯ Try This

9.2.2 Rendering with a Texture Atlas

Rendering billboards using a texture atlas is only slightly more involved than rendering with individual textures. The good news is we can just bind the texture atlas once, then issue a draw call for all our billboards.

After all, this was the ultimate goal—rendering large batches. In order for each billboard to know what subrectangle in the atlas is its texture, texture coordinates need to be stored per billboard. In our geometry-shader implementation, each vertex needs two 2D texture coordinates, one for the lower-left corner and one for the upper right of its subrectangle. These two texture coordinates can be stored in a single vec4. Since we are not using mirror or repeat texture addressing, the texture coordinates will always be in the range $[0, 1]$ so the VertexAttributeComponentType `.HalfFloat` vertex data type provides enough precision and only consumes 2 bytes per component.

To support texture atlases, the only addition needed to the billboard vertex shader in Listing 9.1 is to pass through the texture coordinates:

```
in vec4 position;
in vec4 textureCoordinates;

out vec4 gsTextureCoordinates;

uniform mat4 og_modelViewPerspectiveMatrix;
uniform mat4 og_viewportTransformationMatrix;

vec4 ModelToWindowCoordinates(/* ... */) { /* ... */ }

void main()
{
  gl_Position =
      ModelToWindowCoordinates(position,
                               og_modelViewPerspectiveMatrix,
                               og_viewportTransformationMatrix);
  gsTextureCoordinates = textureCoordinates;
}
```

Changes to the original geometry shader in Listing 9.2 are only slightly more involved. The pixel size of the billboard is no longer the size of the entire texture, it is the size of the texture divided by the size of the billboard's subrectangle, or, equivalently, scaled by the subrectangle's span in texture-coordinate space. Also, texture coordinates are no longer procedurally generated; instead, they are taken from the vertex input:

```
layout(points) in;
layout(triangle_strip, max_vertices = 4) out;

in vec4 gsTextureCoordinates[];
out vec2 fsTextureCoordinates;

uniform mat4 og_viewportOrthographicMatrix;
uniform sampler2D og_texture0;

void main()
```

```
{
  vec4 textureCoordinate = gsTextureCoordinates[0];
  vec2 atlasSize = vec2(textureSize(og_texture0, 0));
  vec2 subRectangleSize = vec2(
      atlasSize.x * (textureCoordinate.p - textureCoordinate.s),
      atlasSize.y * (textureCoordinate.q - textureCoordinate.t));
  vec2 halfSize = subRectangleSize * 0.5;

  vec4 center = gl_in[0].gl_Position;

  vec4 v0 = vec4(center.xy - halfSize, -center.z, 1.0);
  vec4 v1 = vec4(center.xy + vec2(halfSize.x, -halfSize.y),
                 -center.z, 1.0);
  vec4 v2 = vec4(center.xy + vec2(-halfSize.x, halfSize.y),
                 -center.z, 1.0);
  vec4 v3 = vec4(center.xy + halfSize, -center.z, 1.0);

  gl_Position = og_viewportOrthographicMatrix * v0;
  fsTextureCoordinates = textureCoordinate.st;
  EmitVertex();

  gl_Position = og_viewportOrthographicMatrix * v1;
  fsTextureCoordinates = textureCoordinate.pt;
  EmitVertex();

  gl_Position = og_viewportOrthographicMatrix * v2;
  fsTextureCoordinates = textureCoordinate.sq;
  EmitVertex();

  gl_Position = og_viewportOrthographicMatrix * v3;
  fsTextureCoordinates = textureCoordinate.pq;
  EmitVertex();
}
```

9.3 Origins and Offsets

At this point, our billboard implementation is pretty useful: it can efficiently render a large number of textured billboards. There are still several small features that can be implemented in the geometry shader to make it more useful, especially when billboards are lined up next to each other. For example, how would you render a label billboard to the right of an icon billboard, as done in Figure 9.8?

One solution is to treat the icon and label as one image. This can lead to a lot of duplication because unique images need to be made for each icon/label pair. Our geometry shader treats a billboard's 3D position as the center of the billboard. If an icon and label are treated as one billboard, its center point will be towards the middle of the label, as opposed to the more common center of the icon.

Our solution is to add per-billboard horizontal and vertical origins and a pixel offset. These are easily implemented in the geometry shader by translating the billboard's position in window coordinates.

Figure 9.8. Placing an icon and label billboard next to each other as done here requires left aligning and applying a pixel offset to the label billboard.

the origins determine how the billboard is positioned relative to the transformed position (see Figures 9.9 and 9.10). Our implementation thus far implicitly uses a center horizontal and vertical origin. In order to obtain a different origin, a vec2 is stored per vertex. Each component is either -1, 0, or 1. For the x-component, these correspond to left, center, and right horizontal origins, respectively. For the y-component, these correspond to bottom, center, and top vertical origins. The geometry shader simply translates the billboard's position in window coordinates by the origins multiplied by half the billboard's width and height (see Listing 9.7).

```
// ...
in vec2 gsOrigin[];

void main()
{
  // ...
  vec2 halfSize = /* ... */;

  vec4 center = gl_in[0].gl_Position;
  center.xy += (halfSize * gsOrigin[0]);
  // ...
}
```

Listing 9.7. Geometry-shader snippet for horizontal and vertical billboard origins.

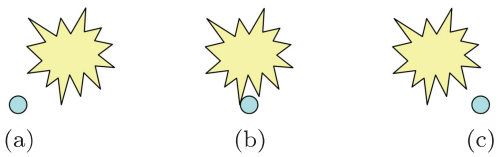


Figure 9.9. Billboard horizontal origins. (a) Left. (b) Center. (c) Right.

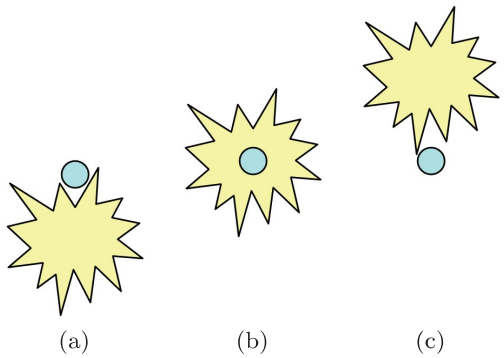


Figure 9.10. Billboard vertical origins. (a) Bottom. (b) Center. (c) Top.

In order the get the effect of Figure 9.8, the label billboard has the same position as the icon billboard and uses a left horizontal origin. The only additional step is to translate the label to the right in window coordinates so it is not on top of the icon. This can easily be achieved by storing a vec2 per vertex that is a pixel offset so the label can be offset to the right by half the width of the icon. The geometry-shader changes, shown in Listing 9.8, are trivial.

There are many other simple but useful operations: a rotation in window coordinates can be used, for example, to orientate a plane icon along the direction the plane is flying; scaling can be applied based on the distance to the billboard; or the billboard can be translated in eye coordinates, for example, to always keep it "on top" of a 3D model. There are a few things to keep in mind when adding features:

- Although these features generally do not increase the size of vertices output from the geometry shader, they tend to increase the

```
// ...
in vec2 gsPixelOffset[];

void main()
{
  // ...
  vec4 center = gl_in[0].gl_Position;
  center.xy += (halfSize * gsOrigin[0]);
  center.xy += gsPixelOffset;
  // ...
}
```

Listing 9.8. Geometry-shader snippet for pixel offsets.

size of vertices entering the vertex and geometry shader. Try to use the least amount of memory possible per vertex. For example, instead of storing horizontal and vertical components in a vec2, they can be stored in a single component and extracted in the geometry shader using bit manipulation. See the geometry shader for `OpenGlobe.Scene`.BillboardCollection. Although it is less flexible, you can also combine things like the resulting pixel offset from the origins and the user-defined pixel offset into one pixel offset, which results in less per-vertex data and fewer instructions in the geometry shader.

- Careful management of vertex buffers can reduce the amount of memory used and improve performance. In particular, if an attribute has the same value across all billboards (e.g., the pixel offset is $(0, 0)$), do not store it in the vertex buffer. Likewise, choose carefully between static, dynamic, and stream vertex buffers. For example, if billboards are used to show the positions for all the current flights in the United States, a static vertex buffer may be used for the texture coordinates since a billboard's texture is unlikely to change, and a stream vertex buffer may be used for the positions, which change frequently.

- If you cull billboards in the geometry shader, be careful to cull after all the offsets, rotations, etc., have been applied. Otherwise, a billboard may be culled when it should be visible. One reason OpenGL point sprites are less commonly used is because the entire sprite is culled based on its center point, which makes large sprites pop in and out along the edges of the viewport.

With respect to performance, it is also important to consider the fill rate. Depending on the texture, a large number of fragments will be dis-

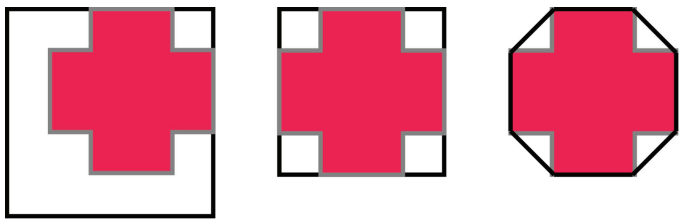


Figure 9.11. (a) A quad only requires four vertices but has a large amount of empty space that will result in discarded fragments. (b) Tightly fitting a bounding rectangle reduces the amount of empty space without increasing the number of vertices. (c) Using the convex hull decreases the amount of empty space even more but increases the vertex count to eight for this image.

carded, wasting rasterization resources. When billboards overlap, which is common in particle systems, overdraw makes the problem even worse. Persson presents a method and a tool for reducing the fill rate by using more vertices per billboard to define a tightly fit convex hull around an image's nonempty space [138] (see Figure 9.11). Of course, a balance needs to be struck between increasing vertex processing and decreasing fill rate.

9.4 Rendering Text

Given our billboard implementation, including the texture atlas and pixel-offset support, it is an easy task to render text for labels. In fact, no new rendering code needs to be written. Text can be implemented as a thin layer on top of billboards.

There are two common methods for rendering text. In both cases, we need a way to write a string to a Bitmap using a particular font. OpenGlobe contains a helper method, Device.`CreateBitmapFromText`, shown in Listing 9.9.

One method for rendering text is to store each word, phrase, or sentence in a texture atlas and render each using a billboard. For example, creating the texture atlas may look like Listing 9.10.

The advantages to this method are that it is easy to implement and uses very little vertex data. The downside is the texture atlas can have a lot of duplication. In the example, ", CA" is duplicated twice. If we consider a finer granularity, "o" is duplicated four times. One technique to minimize duplication is to cache and reuse words or phrases. In the above example, if we chose to cache words, "CA" would only be stored once in the texture atlas but six billboards would be required (assuming commas

```
public static Bitmap CreateBitmapFromText(string text, Font font)
{
  SizeF size = /* ... */

  Bitmap bitmap = new Bitmap(
      (int)Math.Ceiling(size.Width),
      (int)Math.Ceiling(size.Height),
      ImagingPixelFormat.Format32bppArgb);
  Graphics graphics = Graphics.FromImage(bitmap);
  Brush brush = new SolidBrush(Color.White);
  graphics.DrawString(text, font, brush, new PointF());

  return bitmap;
}
```

Listing 9.9. Writing a string to a Bitmap in OpenGlobe.

```
Font font = new Font("Arial", 24);
IList<Bitmap> bitmaps = new List<Bitmap>();
bitmaps.Add(Device.CreateBitmapFromText("Folsom, CA", font));
bitmaps.Add(Device.CreateBitmapFromText("Endicott, NY", font));
bitmaps.Add(Device.CreateBitmapFromText("San Jose, CA", font));

TextureAtlas atlas = new TextureAtlas(bitmaps);
Texture2D texture = Device.CreateTexture2D(atlas.Bitmap,
    TextureFormat.RedGreenBlueAlpha8, false);
```

Listing 9.10. Creating a texture atlas for text rendering.

and spaces are considered part of their preceding word). For larger input sets, caching can be even more effective.

If we take caching to the extreme, we arrive at the other text-rendering method: storing individual characters in the texture atlas (see Figure 9.12) and rendering each character with a single billboard. The amount of texture memory used is small, but the amount of vertex data can become large. For a given string, each billboard stores the same 3D position but a different pixel offset so the correct position for each character is reconstructed in window coordinates in the geometry shader. In addition to storing the texture coordinates for each character with the texture atlas, the pixel offset (i.e., the width and height) of each character also needs to be stored. A pitfall to watch for when implementing this method is computing the texture atlas for an entire character set before knowing what characters are actually used. For ANSI character sets, the atlas will not be very large,

Figure 9.12. A texture atlas storing individual characters for text rendering. The order of the characters was determined by the atlas-packing algorithm, not their position in the alphabet.

depending on the font used, but some Unicode character sets contain lots of characters, most of which will go unused! Regardless of if we are storing individual characters or entire strings, we should be mindful of the size of the created texture. In some cases, multiple textures will be required.

Storing individual characters increases the amount of work on the vertex and geometry shaders but decreases the amount of fill rate. Fewer fragments are shaded since differences among character heights, spacing, and multiline text are taken into account. For example, in the string "San Jose," the geometry around the lowercase characters does not have to be as tall as the geometry around the uppercase ones, and no fragments are shaded for the space between "San" and "Jose" since it can be accounted for with pixel offsets. Texture-caching opportunities are also better when storing individual characters since characters are frequently repeated. Storing individual characters also provides the flexibility to transform and animate individual characters.

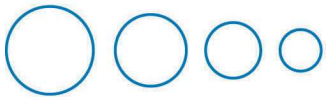
So, which method is better? Storing entire strings, individual characters, or a hybrid caching approach? Storing entire strings can be simpler to implement but wastes texture memory. A caching scheme can be efficient for some inputs but not for others; for example, imagine a simulation with objects named "satellite0," "satellite1," etc.—unless words are broken apart, caching will perform poorly. In Insight3D, we store individual characters in textures, as do most other text-rendering implementations we are aware of.

9.5 Resources

Considering that our implementations for polylines and billboards use geometry shaders, it is important to understand geometry-shader performance characteristics. A great place to start is the NVIDIA [123] and ATI [135] programming guides. Our polyline and billboard implementations require a solid understanding of transformations, which are nicely covered in "The Red Book" [155].

NVIDIA's white paper, *Improve Batching Using Texture Atlases*, is an excellent source of information [121]. In a *Gamasutra* article, Ivanov explains how they integrated texture atlases into a game's art pipeline and renderer [76]. For creating texture atlases, Jylänki provides a recent survey on 2D rectangle bin packing [78]. Jylänki's website includes example C++ code [79].

Matthias Wloka's slides go into great detail on using batching to reduce CPU overhead [183].



10 Exploiting Parallelism in Resource Preparation

The rendering of virtual globes requires constant reading from disk and network connections to stream in new terrain, imagery, and vector data. Before these new data are rendered, they usually go through CPU-intensive processing and are made into a renderer resource like a vertex buffer or texture. We call these tasks as a whole resource preparation. Doing resource preparation in the rendering thread dramatically reduces performance and responsiveness because it frequently blocks the thread on I/O or keeps it busy number crunching instead of efficiently feeding draw commands to the GPU.

This chapter explores ways to exploit the parallelism of modern CPUs via threading to offload resource preparation to other threads. We briefly review the parallelism of modern CPUs and GPUs, then consider various engine architectures for multithreaded resource preparation and look at the sometimes intimidating world of multithreading with OpenGL. Along the way, we learn how to use a message queue to communicate between threads.

If you are building a high-performance graphics engine, virtual globe or not, this material is for you!

10.1 Parallelism Everywhere

Modern CPUs and GPUs exploit a great deal of parallelism when executing our seemingly sequential code.

10.1.1 CPU Parallelism

Consider code written in a high-level programming language for a CPU. This code is compiled into lower-level instructions that the CPU executes. To boost performance, CPUs exploit instruction-level parallelism (ILP) to execute more than one instruction at the same time. Perhaps the most well-known form of ILP is pipelining, which allows new instructions to start executing before previous instructions finish. Suppose each instruction is executed in five stages: fetch, decode, execute, memory, and write back. Without pipelining, each instruction requires five cycles to execute. During those cycles, most of the hardware resources are not utilized. In an ideal pipelining scenario, a new instruction enters the fetch stage every clock cycle so multiple instructions are in flight at the same time. This is the same idea as having one load of laundry in the washer while having another load in the dryer. If we have several loads of laundry (instructions), we don't want to leave the washer idle until the dryer is finished. If there are no stalls in the pipeline, meaning no instructions are paused due to dependencies on previous instructions, an instruction can be completed every cycle, making the speedup directly proportional to the depth of the pipeline—five in our example.

Pipelining isn't the only form of ILP used to speed up our sequential code. Superscalar CPUs can execute more than one instruction at the same time. Combined with out-of-order execution, multiple instructions from the same instruction stream are executed in parallel using a scheduling algorithm that handles instruction dependencies and assigns stages of instructions to duplicated hardware units. Most CPUs are not more than four-way superscalar, meaning they can execute up to four instructions from the same instruction stream in parallel. Given branch-prediction accuracy and all the dependencies in real-world code, making processors any wider isn't currently effective [72].

In addition to ILP, many CPUs provide vector instructions to exploit data-level parallelism (DLP). These include SIMD operations that are widely used in vector math for computer graphics. For example, using a single instruction, each multiply in a dot product can be executed in parallel. It is called SIMD because there is a single instruction, multiply in our example, and multiple data since vectors, not scalars, are operated on. Superscalar is interinstruction parallelism, while vector instructions are intrainstruction parallelism.

What does all this computer architecture have to do with virtual globes? On the surface, it explains how CPUs use creative techniques to execute our single-threaded code efficiently. In addition, we can borrow ideas from a CPU's parallelism to help design and exploit parallelism in engine code. Before we do so, let's look at the parallelism we get for free with GPUs.

10.1.2 GPU Parallelism

The incredible triangle throughput and fill rates of today's GPUs are thanks to the amount of parallelism that can be exploited during rasterization and thanks to Moore's law providing more and more transistors to realize this parallelism. There is so much parallelism in rasterization that it is considered embarrassingly parallel. First, consider vertex and fragment shaders. Since transforming and shading a vertex does not depend on another vertex, a shader can execute on different vertices in parallel. The same is also true for fragments.

Similar to how CPUs use SIMD to exploit DLP, GPUs use single instruction, multiple threads (SIMT) to execute the same instruction on different data. Each vertex or fragment is shaded using a lightweight thread. When the same shader is used for several vertices or fragments, each instruction is executed at the same time (up to the number of lanes) on a different vertex or fragment using SIMT. For best performance, SIMT requires reasonable code-path coherency between threads so threads running on a single hardware unit follow the same code path. Divergent branches, for example, an if ... else statement where every other fragment takes a different part of the branch, can adversely affect performance.

A triangle does not need to go through all the stages of the pipeline and be written to the framebuffer before the next triangle enters the pipeline. Just like a pipelined CPU has multiple instructions in flight at the same time, multiple triangles are in flight on a GPU.

GPUs are designed to exploit the embarrassingly parallel nature of graphics. Today's GPUs have hundreds of cores that can host thousands of threads. When we issue `context.Draw`, we are invoking a massively parallel computer for rendering! Not only is the GPU itself parallel, but the CPU and GPU can work together in parallel. For example, while the GPU is executing a draw command, the CPU may be processing geometry that will be drawn in the future.

10.1.3 Parallelism via Multithreading

Given all the parallelism under the hood of a CPU and GPU, why do we need to write multithreaded code? After all, we are already writing massively parallel code by writing shaders—and this parallelism comes for free without the error-prone development and painful debugging stereotypically associated with multithreading. To add to the argument, we frequently hear claims that applications written using OpenGL cannot benefit from multithreading. So why embrace multithreading? Performance.

Given the trend of increasing CPU cores, software developers can no longer write sequential code and expect it to run faster every time a new

CPU is released. Although the increase in transistor density predicted by Moore's law has continued to hold true, this does not imply single-instruction stream performance continues to climb at the same pace. Due to a limited amount of ILP in real-world code and physical issues including heat, power, and leakage, these extra transistors are largely used to improve performance for multiple instruction streams.[1] To create multiple instruction streams, we need multiple threads. As Herb Sutter put it: "the free lunch is over" [161].

CPUs that exploit multiple instruction streams are said to exploit thread-level parallelism (TLP). CPUs can do this in many ways. Multithreaded CPUs support efficient switching between multiple instruction streams on a single CPU. This is particularly useful when a thread misses the L2 cache; instead of stalling the thread for a large number of cycles, a new thread is quickly swapped in. Although multithreaded CPUs do not improve the latency of any single instruction stream, they can increase the throughput of multiple instruction streams by tolerating cache misses.

Hyper-threaded CPUs turn TLP into ILP by executing instructions from different instruction streams at the same time, similar to how a superscalar processor executes instructions from the same instruction stream at the same time. The difference is the amount of parallelism in a single instruction stream is limited compared to the amount of parallelism in multiple instruction streams.

To utilize large transistor budgets, CPUs are exploiting multiple instruction streams by going multicore; loosely speaking, putting multiple CPUs on the same chip. This allows multiple instruction streams to truly execute at the same time, similar to a multiprocessor system with multiple CPUs. Considering that today's desktops can have six cores and even laptops can have four, we would be missing out on an awful lot of transistors if we did not write code to utilize multiple cores.

10.2 Task-Level Parallelism in Virtual Globes

If we write multithreaded code that runs on a multithreaded, hyper-threaded, multicore CPU, we can expect lots of fancy things to happen: threads execute at the same time on different cores, instructions from different threads execute at the same time on the same core, and threads are quickly switched on a core in the face of an L2 cache miss.

But how do we take advantage of this? Isn't most of our performance-critical code sequential: `bind vertex array`, `bind textures`, `set render`

[1] The most prominent exception is that larger caches can improve single instruction stream performance.

`state`, `issue draw call`, repeat? There is only one GPU, on most systems anyway. Well, yes, we spend a great deal of time issuing draw calls and multithreading has been shown to improve performance here [6, 80, 133], but multithreading can also be used to improve the responsiveness of our application. Considering that virtual globes are constantly loading and processing new data, multithreading is critical for smooth frame rates. Specifically, the following tasks are great candidates to move off the main rendering thread and onto one or more worker threads:

- *Reading from secondary storage.* Reading from disk or a network connection is very slow and can stall the calling thread.[2] Doing so in the rendering thread makes the application appear to freeze. By moving secondary storage access to a worker thread, the rendering thread can continue issuing draw calls while the worker thread is blocked on I/O. Doing so utilizes a multithreaded processor's ability to keep its core busy with unblocked threads.

- *CPU-intensive processing.* Virtual globes are not short on CPU-intensive processing, including triangulation (see Section 8.2.2), texture-atlas packing (see Section 9.2.1), LOD creation, computing vertex cache-coherent layouts, image decompression and even recompression, generating mipmaps, and other processing mainly used to prepare vertex buffers and textures. By moving these tasks to a worker thread, we are utilizing a hyper-threaded and multicore processor's ability to execute different threads at the same time.

- *Creating renderer resources.* Creating resources such as vertex buffers, textures, shaders, and state blocks can be moved to a separate thread to reduce the associated CPU overhead. Of course, data for vertex buffers and textures still need to travel over the same system bus, an operation that is generally considered asynchronous, even when executed from the main rendering thread. By moving these operations into a worker thread, we can simplify our application's architecture and avoid holding onto temporary copies of data prepared by the worker thread in system memory because renderer-resource creation isn't restricted to the rendering thread.

Delegating tasks like these to worker threads is an example of TLP, which is the focus of this chapter. Using this type of parallelism, tasks are the units of work and are executed concurrently on different CPU cores.

The goal is to reduce the amount of work done by the rendering thread so it can continue to issue draw commands. If the rendering thread spends

[2] With the exception of asynchronous I/O, which returns immediately and later issues an event when the I/O is complete.

too much time waiting on I/O or on CPU-intensive processing, it can starve the GPU. A primary goal of any 3D engine is to allow the CPU and GPU to work in parallel, fully utilizing both.

Resource preparation, mentioned in the introduction, is the pipeline of these three tasks: I/O, processing, and renderer-resource creation. Resource preparation is an excellent candidate for multithreading.

10.3 Architectures for Multithreading

In order to explore software architectures that move resource preparation to worker threads, let's build on Chapter07VectorData. At startup, the application loads several shapefiles on the main thread with code similar to Listing 10.1.

This involves going to disk, using CPU-intensive computations like triangulation, and creating renderer resources—all excellent candidates for multithreading! Chapter07VectorData takes a noticeable amount of time to start. By moving these tasks to a worker thread or threads, the application's responsiveness improves: the application starts quickly, the user can interact with the scene and spin the globe while shapefiles are loading, and a shapefile appears as soon as it is ready. In short, the main thread can focus on rendering while worker threads prepare shapefiles.

The usefulness of this goes far beyond application startup. The same responsiveness can be gained after the user selects files in a file open dialog. Better yet, shapefiles may be automatically loaded based on view parameters such as viewer altitude. If we combine automatic loading with a replacement policy to remove shapefiles, we are awfully close to an out-of-core rendering engine. This idea doesn't just apply to shapefiles, it is true for terrain and imagery as well.

```
_countries = new ShapefileRenderer(
    "110m_admin_0_countries.shp", /* ... */);
_states = new ShapefileRenderer(
    "110m_admin_1_states_provinces_lines_shp.shp", /* ... */);
_rivers = new ShapefileRenderer(
    "50m-rivers-lake-centerlines.shp", /* ... */);
_populatedPlaces = new ShapefileRenderer(
    "110m_populated_places_simple.shp", /* ... */);
_airports = new ShapefileRenderer("airprtx020.shp", /* ... */);
_amtrakStations = new ShapefileRenderer(
    "amtrakx020.shp", /* ... */);
```

Listing 10.1. Shapefile loading in the main thread in Chapter07VectorData.

10.3.1 Message Queues

When moving to multithreading, we need to consider how threads communicate with each other. The rendering thread needs to tell a worker thread to prepare a shapefile, and a worker thread needs to tell the rendering thread that a shapefile is ready for rendering. Depending on the division of labor, worker threads may even need to communicate with each other. In all cases, the communication is a request to do work or a notification that work is complete. These messages also contain data (e.g., the file name of the shapefile or the actual object for rendering).

An effective way for threads to communicate is via shared message queues. A message queue allows one thread to post messages (e.g., "load this shapefile" or "I finished loading a shapefile, here is the object for rendering") and another thread to process messages. Message queues do not require client code to lock; ownership of a message determines which thread can access it.

The code for our message queue is in `OpenGlobe.Core`.MessageQueue. Appendix A details its implementation. Here, we consider just its most important public members:

```
public class MessageQueueEventArgs : EventArgs
{
  public MessageQueueEventArgs(object message);
  public object Message { get; }
}

public class MessageQueue : IDisposable
{
  public event EventHandler<MessageQueueEventArgs>
      MessageReceived;
  public void StartInAnotherThread();
  public void Post(object message);
  public void TerminateAndWait();
  // ...
}
```

`Post` is used to add new messages to the queue. Messages are of type object,[3] so they can be simple intrinsic types, structs, or classes. `Post` is asynchronous; it adds a message to the queue, then returns immediately without waiting for it to be processed. For example, the rendering thread may post shapefile file names, then immediately continue rendering. Since `Post` locks under the hood, multiple threads can post to the same queue.

`MessageReceived` is the event that gets called to process a message in the queue. The message is received as MessageQueueEventArgs.`Message`. This

[3] If you were to implement this type of message queue in C or C++, you might use void * in place of object.

is where the work we want to move off the rendering thread is performed. The method `StartInAnotherThread` creates a new worker thread to process messages posted to the queue. This method only needs to be called once. Messages in the queue are automatically removed in the order they were added (i.e., first in, first out (FIFO)), and `MessageReceived` is invoked on the worker thread to process each. Finally, `TerminateAndWait` blocks the calling thread until the queue processes all of its messages. Our message queue contains other useful public members that we'll introduce as needed.

The following console application uses a message queue and a worker thread to square numbers:

```
class Program
{
  static void Main(string[] args)
  {
    MessageQueue queue = new MessageQueue();
    queue.StartInAnotherThread();

    queue.MessageReceived += SquareNumber;

    queue.Post(2.0);
    queue.Post(3.0);
    queue.Post(4.0);

    queue.TerminateAndWait();
  }

  private static void SquareNumber(object sender,
                                   MessageQueueEventArgs e)
  {
    double value = (double)e.Message;
    Console.WriteLine(value * value);
  }
}
```

It outputs 4.0, 9.0, and 16.0. First, it creates a message queue and immediately starts it in a worker thread by calling `StartInAnotherThread`. The thread will wait until it has messages to process. Each message posted to the queue is processed by the `SquareNumber` function in the worker thread. The main thread then posts three doubles to the queue. Since the main thread has no more work to do, it calls `TerminateAndWait` to wait for each double on the queue to be processed. This is a bit of a toy example because it doesn't have much, if any, parallelism between the main thread and the worker thread because the main thread doesn't really have anything else to do. In a 3D engine, the main thread continues rendering while worker threads prepare resources.

Given our understanding of message queues, let's explore architectures for moving shapefile preparation off the main thread.

10.3.2 Coarse-Grained Threads

Chapter07VectorData uses only a single thread. It first prepares shapefiles, then enters the draw loop. To make this application multithreaded, the simplest design is to add one worker thread responsible for preparing shapefiles, which allows the main thread to enter the draw loop immediately, instead of waiting for all shapefiles to be ready. The main thread spends its time issuing draw calls while the worker thread prepares shapefiles.

We call this a coarse-grained design because the worker thread is not divided into fine-grained tasks; it does everything that is required to prepare shapefiles for rendering: loading, CPU-intensive processing, and renderer-resource creation. Two message queues are used to communicate between the rendering and worker threads: one for requesting the worker thread to prepare a shapefile and another for notifying the rendering thread that a shapefile is ready (see Figure 10.1).

Chapter10Multithreading extends Chapter07VectorData to use coarse-grained threading with a single worker thread. It introduces a small class called ShapefileRequest to represent the "load shapefile" message that the rendering thread posts on the request queue for consumption by the worker thread. It contains the shapefile's file name and visual appearance, as seen in Listing 10.2.

```
internal class ShapefileRequest
{
    public ShapefileRequest(string filename,
                            ShapefileAppearance appearance);

    public string Filename { get; }
    public ShapefileAppearance Appearance { get; }
}
```

Listing 10.2. Shapefile request public interface.

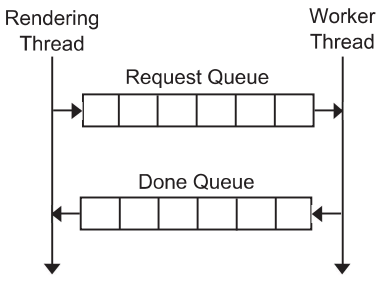


Figure 10.1. Coarse-grained threading with a single worker thread. The rendering thread and worker thread communicate using message queues.

The rendering thread has references to both the request and done queues. It addition, it keeps a list of shapefiles for rendering:

```
private readonly MessageQueue _requestQueue = new MessageQueue();
private readonly MessageQueue _doneQueue = new MessageQueue();
private readonly IList<IRenderable> _shapefiles =
    new List<IRenderable>();
```

As shapefiles come off the done queue, `_doneQueue`, they will be placed onto the shapefile list, `_shapefiles`, for rendering. When the application starts, instead of preparing the shapefiles immediately, as shown earlier in Listing 10.1, shapefile requests are added to the request queue, `_requestedFiles`, to be processed in the worker thread:

```
_requestQueue.Post(new ShapefileRequest(
    "110m_admin_0_countries.shp", /* ... */));
_requestQueue.Post(new ShapefileRequest(
    "110m_admin_1_states_provinces_lines_shp.shp", /* ... */));
_requestQueue.Post(new ShapefileRequest(
    "airprtx020.shp", /* ... */));
_requestQueue.Post(new ShapefileRequest(
    "amtrakx020.shp", /* ... */));
_requestQueue.Post(new ShapefileRequest(
    "110m_populated_places_simple.shp", /* ... */));
_requestQueue.StartInAnotherThread();
```

Compared to Listing 10.1 used in Chapter07VectorData, this code executes very quickly—it just adds messages to a queue. Shapefile preparation is moved to the worker thread using a small class:

```
internal class ShapefileWorker
{
  public ShapefileWorker(MessageQueue doneQueue)
  {
    _doneQueue = doneQueue;
  }

  public void Process(object sender, MessageQueueEventArgs e)
  {
    ShapefileRequest request = (ShapefileRequest)e.Message;
    _doneQueue.Post(
        new ShapefileRenderer(request.Filename, /* ... */));
  }

  private readonly MessageQueue _doneQueue;
}
```

The type ShapefileWorker represents the worker thread. It is constructed by the rendering thread, which passes it to the done queue:

```
_requestQueue.MessageReceived +=
    new ShapefileWorker(_doneQueue).Process;
```

Its trivial constructor executes in the rendering thread, but `Process` is invoked for each shapefile in the message queue on the worker thread. The `Process` call simply uses the message to construct the appropriate shapefile object for rendering. Once constructed, the object is placed on the done queue so it can be picked up by the rendering thread. The rendering thread uses a method called `ProcessNewShapefile` to respond to messages posted on the done queue:

```
_doneQueue.MessageReceived += ProcessNewShapefile;
```

It simply adds the shapefile to the list of shapefiles for rendering:

```
public void ProcessNewShapefile(object sender,
                                 MessageQueueEventArgs e)
{
  _shapefiles.Add((IRenderable)e.Message);
}
```

The rendering thread explicitly processes the done queue once per frame using the `ProcessQueue` message-queue method, not yet introduced. This method empties the queue and processes each message synchronously in the calling thread. In our case, this is a very quick operation since the messages are simply added to the shapefiles list. The entire render-scene function is as follows:

```
private void OnRenderFrame()
{
  _doneQueue.ProcessQueue();

  Context context = _window.Context;
  context.Clear(_clearState);
  _globe.Render(context, _sceneState);

  foreach (IRenderable shapefile in _shapefiles)
  {
    shapefile.Render(context, _sceneState);
  }
}
```

An alternative to using the done queue is to allow the worker thread to add a shapefile directly to the shapefile list. This is less desirable because it decreases the parallelism between threads. The shapefile list would need to be protected by a lock. The rendering thread would need to hold this lock while it is iterated over the list for rendering. During this time, which is likely to be the majority of the rendering thread's time, the worker thread will not be able to add a shapefile to it. Furthermore, providing the worker thread access to the shapefile list increases the coupling between threads.

The code in Chapter10Multithreading is slightly more involved than the code presented in this section. Since our renderer is implemented with OpenGL, additional considerations (coming up in Section 10.4) need to be taken into account when renderer resources are created on the worker thread.

Try This ○○◯◯

Change Chapter10Multithreading to load shapefiles from a file open dialog instead of at startup.

Try This ○○◯◯

After adding a file open dialog to Chapter10Multithreading, add the ability to cancel loading a shapefile. How does this affect synchronization?

Try This ○○◯◯

Change the synchronization granularity in Chapter10Multithreading to be more frequent than per shapefile. Have the rendering thread incrementally render a shapefile; for example, start once 25% of it is prepared, then update again at 50%, and so on. This will improve the responsiveness, especially for large shapefiles, since features start appearing on the globe sooner. How does this affect the overall latency of preparing a shapefile? How much complexity does this add to the design?

Multiple threads. Coarse-grained threading with a single worker thread does an excellent job of keeping resource preparation off the rendering thread. But does it fully utilize a multicore CPU? How responsive is it? Consider a dual-core CPU, with one core running the rendering thread and another running the worker thread. How often is the worker thread blocked on I/O?

How often is the worker thread doing CPU-intensive work? It depends, of course; the bottleneck for large polyline shapefiles preparation may be I/O, while the bottleneck for preparing polygon shapefiles may be CPU-intensive triangulation, LOD creation, and computing vertex cache-coherent layouts.

Using a single worker thread does not fully utilize the second core nor does it fully utilize the I/O bandwidth. A single worker also does not scale to more cores. Since our message queue processes messages in the order they were received, a single worker does not provide ideal responsiveness. If a large shapefile is placed on the queue, no other shapefiles will be rendered until the large one is prepared. This increases the latency for rendering other shapefiles that can be prepared quickly. The problem becomes more prominent if instead of going to disk, the I/O is going to an unreliable network. In this case, the worker thread will be blocked until the connection times out.

A simple solution to these scalability and responsiveness problems is to use multiple coarse-grained worker threads. Figure 10.2 shows an example with three workers. Compare this to Figure 10.1 with only a single worker. When using our message queue without modifications, each worker thread can post to the same done queue, but a different request queue is required per worker. The simplest request-queue-scheduling algorithm is round-robin: just rotate through the request queues.

The ideal number of workers depends on many factors, including the number of CPU cores, amount of I/O, and type of I/O. For example, if the I/O is always over a network, more workers will be useful. This is similar to how a web browser may download each image of a webpage in

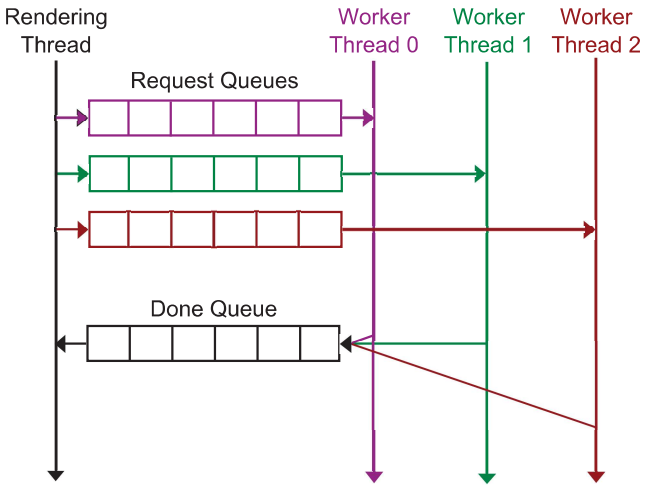


Figure 10.2. Coarse-grained threading with multiple worker threads.

a different thread. On the other hand, if the I/O is from a single optical disk, like a DVD, multiple threads can actually degrade performance due to excessive seeking. Likewise, multiple threads reading from a hard disk may hurt performance by polluting the operating system's file cache. More worker threads also means more memory, task switching, and synchronization overhead.

When a single worker thread is used, shapefiles are prepared one at a time in a FIFO order. With multiple workers, shapefiles are prepared concurrently, making the order that shapefiles are ready nondeterministic. Different shapefiles may complete in a different order from run to run.

Using multiple coarse-grained workers requires some attention to detail to avoid race conditions. In particular, it should be safe to create multiple independent instances of our shapefile types in separate threads. Fortunately, this is pretty straightforward since tasks like triangulation rarely have any threading problems; triangulating different polygons in different threads is completely independent. Shared data structures, such as a shader program cache, should be protected by a lock that is held for the least amount of time necessary.

Try This ○○◯◯

Modify Chapter10Multithreading to use multiple coarse-grained workers using round-robin scheduling. Each worker will need its own GraphicsWindow (see Section 10.4). What are the downsides to round robin? What are the alternatives?

10.3.3 Fine-Grained Threads

An alternative to multiple coarse-grained threads is a pipeline of more finely grained threads connected via message queues, like that shown in Figure 10.3. Here, two worker threads are used. The rendering thread sends requests to the load thread via the load queue. This is the same request used with coarse-grained threads. The load thread is just responsible for reading from disk or the network. The load thread then sends the data to the processing thread for CPU-intensive processing algorithms and renderer-resource creation. Finally, the processing thread posts ready-to-render objects to the done queue.

In essence, we've broken our coarse-grained thread into two more finely grained threads: one I/O bound and another CPU bound. The rationale is that breaking up the tasks in this manner leads to better hardware utilization. With multiple coarse-grained threads, every thread could be

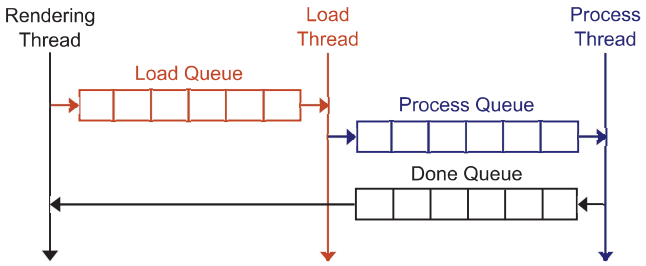


Figure 10.3. Fine-grained threading with separate workers for loading and processing (CPU intensive algorithms and renderer-resource creation).

waiting on I/O while CPU cores are idle, or vice versa. Similar to CPU pipelining, a pipeline of fine-grained threads results in longer latencies while trying to achieve better throughput. The latency is introduced by more message queues and the potential need to make intermediate copies. For example, in our pipeline, data from disk cannot be read directly into a mapped renderer buffer because the load thread and processing thread are independent.

It is generally worthwhile to use more than two worker threads. The number of processing threads can be equal to the number of available cores. As mentioned earlier, the number of load threads should be based on the type of I/O. If all I/O is from a local hard drive, too many load threads will contend for I/O. If the I/O is from different network servers, many load threads will help improve throughput.

Modify Chapter10Multithreading to use fine-grained threads. How many load and processing threads are ideal in your case?

○○○○ Try This

Texture-streaming pipeline. Van Waveren presents a pipeline for streaming massive texture databases using fine-grained threads [172,173]. It is similar to our fine-grained pipeline for shapefiles in that it has dedicated threads for I/O and processing, although only the rendering thread talks to the graphics driver.

A texture-streaming thread reads lossy compressed textures in a format similar to JPEG. The compressed texture is then sent to the transcoding thread for de-recompression, while the texture-streaming thread continues reading other data. The transcoding thread decompresses the texture using an SIMD-optimized decompresser, then recompresses the texture to DXT.

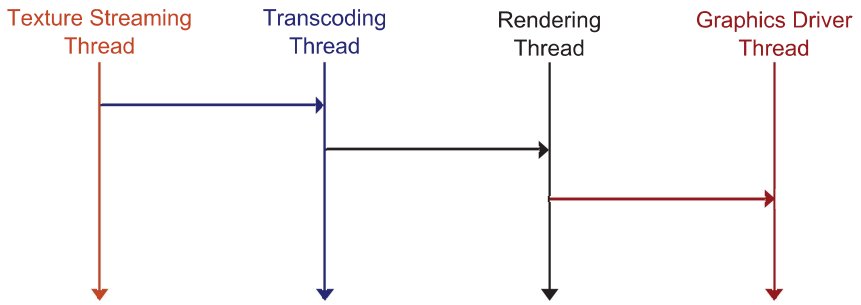


Figure 10.4. A pipeline for texture streaming that separates I/O and CPU processing, and assumes a threaded graphics driver (see Section 10.4.4).

DXT-compressed textures can be efficiently decoded in hardware so they result in using less graphics memory, generally faster rendering, and lower bandwidth. DXT is lossy so some quality may be lost, but when the same amount of memory is used for a DXT-compressed texture compared to a noncompressed texture, the DXT-compressed texture has better quality. Textures are not stored DXT compressed on disk because higher compression ratios can be achieved with other algorithms, reducing the amount of I/O. After DXT compression, the texture is passed to the rendering thread, which interacts with the graphics driver (see Figure 10.4).

Van Waveren observed that transcoding work was typically completely hidden by the time the streaming texture thread had read the data. Since the streaming thread spends most of its time waiting on I/O, a lot of CPU time is available for the transcoding thread.

10.3.4 Asynchronous I/O

We've presented fine-grained threads as a way to better utilize hardware by separating I/O- and CPU-bound tasks into separate threads. With fine-grained threads, I/O threads use blocking I/O system calls. Since these threads spend most of their time waiting on I/O, the CPU is free to execute the processing and rendering threads.

There is another way to achieve this hardware utilization without fine-grained threads: coarse-grained threads with asynchronous I/O. This is done by replacing the blocking I/O system calls with nonblocking, asynchronous I/O system calls that return immediately and trigger a callback when an I/O operation is complete. This lets the operating system manage pending I/O requests, instead of the application managing fine-grained I/O threads, resulting in less memory and CPU usage due to fewer threads and fewer potential data copies.

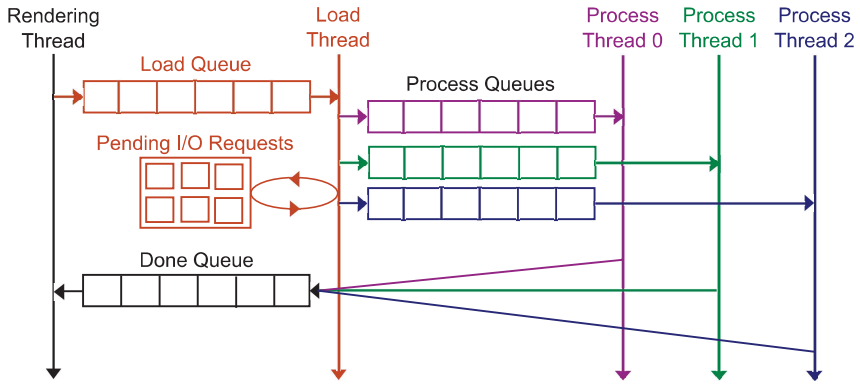


Figure 10.5. Fine-grained threads with a single load thread using asynchronous I/O, allowing the operating system to handle multiple pending I/O requests.

Asynchronous I/O can also be used with fine-grained threads. In this architecture, a single load thread uses asynchronous I/O, which then feeds multiple processing threads, as shown in Figure 10.5. Van Waveren suggests this method over coarse-grained threads with asynchronous I/O because a separate I/O thread allows sorting of multiple data requests to minimize seek times [173]. This is important for hard disks and DVDs but not for network I/O.

One drawback to threading architectures relying on asynchronous I/O is that they are not compatible with some third-party libraries.

10.3.5 Single-Threaded Test/Debug Mode

Regardless of the multithreading architecture used, it is worth including an option to run in a single thread. This is useful for debugging and performance comparisons. In fact, it is sometimes easier to implement a multithreaded architecture on a single thread and then, after verifying it works, add multithreading.

Chapter10Multithreading has a preprocessor variable, `SINGLE_THREADED`, that enables the application to run in a single thread. The major difference when running in a single thread is that the worker "thread" does not need a separate GraphicsWindow (see Section 10.4), and instead of creating a thread to process the request queue, it is processed immediately at startup:

```
#if SINGLE_THREADED
   _requestQueue.ProcessQueue();
#else
   _requestQueue.StartInAnotherThread();
#endif
```

Try This ○○◯◯

Run Chapter10Multithreading both with and without `SINGLE_THREADED` defined. Which one starts faster? Which one loads all the shapefiles faster? Why?

10.4 Multithreading with OpenGL

We've detailed using worker threads to offload I/O and CPU-intensive computations from the rendering thread, but up until this point, we've ignored how exactly the renderer resources are finally created. This section explains how to create renderer resources on a worker thread by taking a closer look at multithreading with OpenGL and its abstraction in `OpenGlobe.Renderer`. Similar multithreading techniques can also be implemented with Direct3D [114].

Let's consider three threading architectures for creating GL resources such as vertex buffers and textures.

10.4.1 One GL Thread, Multiple Worker Threads

In this architecture, worker threads do not make any GL calls. Instead, they prepare data that are used to create GL resources in the rendering thread, as shown in Figure 10.6. This is the simplest approach because all GL calls are made from the same thread. It does not require multiple contexts, locks around GL calls, or GL synchronization. On the downside,

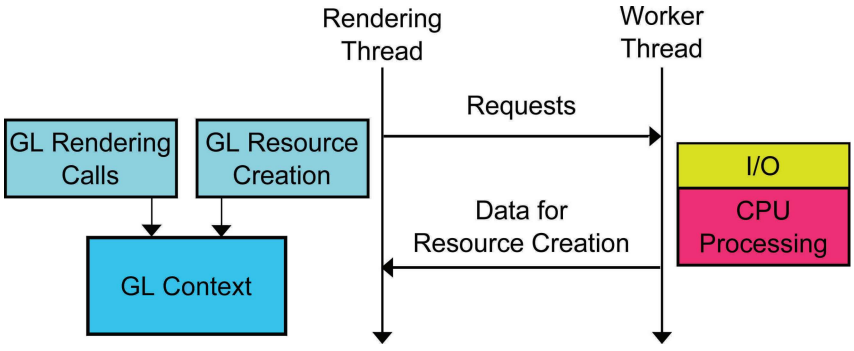


Figure 10.6. When only the rendering thread issues GL calls, worker threads prepare data in system memory or driver-controlled memory using buffer objects.

it may require some engine restructuring since GL resources cannot be created until the data are in the rendering thread.

A variation of this approach is to create vertex buffer objects (VBOs) or pixel buffer objects (PBOs) in the rendering thread, then hand off memory-mapped pointers to them to a worker thread. The worker thread can then write into driver-controlled memory using the mapped pointer, without issuing any GL calls. This variation also has the pitfall of requiring some restructuring because buffer objects need to be created in advance by the rendering thread.

10.4.2 Multiple Threads, One Context

Another approach to GL threading is to access a single context using multiple threads. This architecture allows worker threads to create GL resources, as shown in Figure 10.7. A coarse-grained lock around every GL call is required to ensure multiple threads don't call into GL at the same time. Furthermore, a context can only be current in one thread at a time. Each GL call needs to be wrapped, similar to Listing 10.3. It is possible to reduce the number of calls to `MakeCurrent` by using more coarse-grained synchronization.

This approach is not recommended because the significant amount of locking eliminates the potential parallelism gained by calling GL from multiple threads.

```
public class ThreadSafeContext
{
  public void GLCall()
  {
    lock (_context)
    {
      _context.MakeCurrent(_window.WindowInfo);
      // GL call
      _context.MakeCurrent(null);
    }
  }

  private NativeWindow _window;
  private GraphicsContext _context;
}
```

Listing 10.3. Accessing a context from different threads requires locking so only one thread has a current context.

10.4.3 Multiple Threads, Multiple Contexts

Our preferred architecture is to use multiple threads, each with its own context that is shared with the contexts of other threads, as shown in

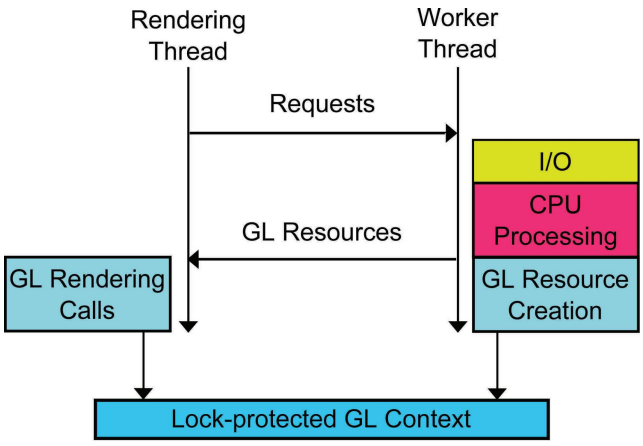


Figure 10.7. A single context accessed from multiple threads allows GL resource creation on a worker thread but requires a coarse-grained lock around GL calls and careful tracking of each thread's current context, as done in Listing 10.3.

Figure 10.8. This allows creating a GL resource in a worker thread, then using it in the rendering thread, without protecting each GL call with a lock. Each thread can make GL calls to its own context at the same time. GL synchronization is required so that the rendering thread's context knows when a GL resource created in a worker thread's context is completed. Since only a single context was used in the previous two approaches, GL synchronization was not required, making them easier to implement in this regard.

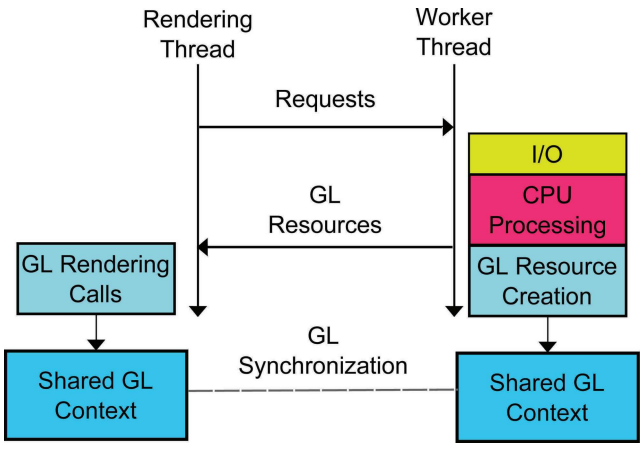


Figure 10.8. Multiple threads, each with a shared context. GL synchronization is required so the rendering thread knows when a GL resource created on a worker thread completed creation.

Once context sharing and synchronization are designed, this approach leads to a clean architecture because most GL calls can be done at a natural time. This can also be more efficient than making all GL calls in a single thread because driver CPU overhead can be moved off the rendering thread to a worker thread. Although CPU overhead is unlikely to be the bottleneck for generally bandwidth-limited tasks such as vertex buffer uploads, tasks such as compiling and linking shaders can have a significant amount of CPU overhead. Even texture uploads can have CPU overhead if the data need to be compressed or converted to an internal format. Be aware that the performance gain is potentially limited by the driver, which may hold coarse-grained locks. The main motivation for this multithreading architecture is that it lends itself to a natural design.

Sharing per-thread contexts. Our one-context-per-thread approach requires that each thread contain an OpenGL context that shares its resources with other OpenGL contexts. In Chapter10Multithreading, the context for the rendering thread and worker thread are created at startup by creating a window for each:

```
_workerWindow = Device.CreateWindow(1, 1);
_window = Device.CreateWindow(800, 600, "Multithreading");
```

Under the hood, Device.`CreateWindow` calls `wglCreateContextAttribs` and `wglMakeCurrent` on Windows. This creates a context shared with existing contexts, and makes it current in the calling thread.

The window for the worker thread is never shown. The worker-thread context is created first because creating a context makes the context current in the calling thread, and both contexts are created in the rendering thread. Alternatively, this could be coded as follows:

```
_window = Device.CreateWindow(800, 600, "Multithreading");
_workerWindow = Device.CreateWindow(1, 1);
_window.Context.MakeCurrent();
```

The worker-thread context can be passed to a worker thread like the one in Listing 10.4. The worker thread must then set the context current at least once before any GL calls are made.

Shared contexts share some but not all resources. Care needs to be taken so that a nonshared resource created in one context is not used in another context. The OpenGL 3.3 specification defines the following objects

```
internal class Worker
{
  public Worker(GraphicsWindow window)
  {
    _window = window; // Executed in rendering thread
  }

  public void Process(object sender, MessageQueueEventArgs e)
  {
    _window.Context.MakeCurrent(); // Executed in worker thread
    // GL calls ...
  }

  private readonly GraphicsWindow _window;
}
```

Listing 10.4. Setting a context current in a worker thread.

as not shared across contexts: framebuffer objects (FBOs), query objects, and vertex-array objects (VAOs) [152]. It makes sense not to share these objects because they are all lightweight; the overhead to make them shared may be significant relative to the object itself. FBOs and VAOs may sound heavyweight, but they are just containers for textures/render buffers and vertex/index buffers, respectively.

Shared objects in the OpenGL specification include PBOs, VBOs, shader objects, program objects, render buffer objects, sync objects, and texture objects.[4] Relative to nonshared objects, shared objects are usually much larger. For example, a VBO may contain a significant number of vertices, whereas a VAO is just a container whose size is independent of the size of the VBOs it contains.

The code in `OpenGlobe.Renderer` provides a natural separation between shared and nonshared objects. Nonshared objects, which can only be used in the context in which they are created, are created using factory methods on `OpenGlobe.Renderer`.Context:

```
public abstract class Context
{
  public abstract VertexArray CreateVertexArray();
  public abstract Framebuffer CreateFramebuffer();
  // ...
}
```

Shared objects are created using factory methods on `OpenGlobe.Renderer`.Device:

[4] The texture object named zero is not shared.

```
public static class Device
{
  public static ShaderProgram CreateShaderProgram();
  public static VertexBuffer CreateVertexBuffer(/* ... */);
  public static IndexBuffer CreateIndexBuffer(/* ... */);
  public static MeshBuffers CreateMeshBuffers(/* ... */);
  public static UniformBuffer CreateUniformBuffer(/* ... */);
  public static WritePixelBuffer CreateWritePixelBuffer(
      /* ... */);
  public static Texture2D CreateTexture2D(/* ... */);
  public static Fence CreateFence();
  // ...
}
```

This is a natural separation; anyone can access the static Device to create a shared object, but only those with access to a Context can create a nonshared object. The trouble point is when a worker thread creates a nonshared object using its context, then passes it back to the rendering thread, which cannot use the object. In our experience, we are rarely tempted to create FBOs on worker threads, but VAOs are more seductive.

Add a debugging aid that catches when a nonshared object is used in a context other than the one it was created in.

○○○○ Try This

Of course, the solution is to delay creating the VAO until the rendering thread can do it. This is similar to the first OpenGL multithreading architecture presented, in which all GL calls were done from the rendering thread, but now, only required GL calls are done on the rendering thread.

Using our abstractions in `OpenGlobe.Renderer` with only a single thread and single context, a class that renders a triangle mesh may use the code in Listing 10.5. The constructor creates a shader program and vertex array for a mesh. The call to `CreateVertexArray` creates vertex and index buffers for the mesh and attaches them to a newly created vertex array in one convenient call. The class's `Render` method is a simple one-liner that draws using the objects created in the constructor.

This class cannot be created in the same way on a worker thread. What context would be passed to the constructor? The worker thread cannot use the rendering thread's context without locking. It also cannot use its own context to create the vertex array. The class in Listing 10.6 shows the solution. The worker's context is used to create the shader program and vertex and index buffers, which are stored in an intermediate

```
public class RenderableTriangleMesh
{
  public RenderableTriangleMesh(Context context)
  {
    Mesh mesh = /* ... */;

    _drawState = new DrawState();
    _drawState.ShaderProgram =
        Device.CreateShaderProgram(/* ... */);
    _drawState.VertexArray = context.CreateVertexArray(
        mesh, _drawState.ShaderProgram.VertexAttributes,
        BufferHint.StaticDraw);
  }

  public void Render(Context context, SceneState sceneState)
  {
    context.Draw(PrimitiveType.Triangles,
                 _drawState, sceneState);
  }

  private readonly DrawState _drawState;
}
```

Listing 10.5. If an object is used from only a single context, all renderer resources can be created at the same time.

MeshBuffers object, which is a simple container—like an engine-level vertex array instead of a GL vertex array:

```
public class MeshBuffers : Disposable
{
  public VertexBufferAttributes Attributes { get; }
  public IndexBuffer IndexBuffer { get; set; }

  // ...
}
```

In the rendering thread, before the object is rendered, MeshBuffers is used to create the actual vertex array using an overload to `CreateVertexArray`. This is expected to be a quick operation since the vertex/index buffers were already created in the worker thread.

Question ○○○◯

Is it a good idea to create a vertex array immediately before rendering with it? Why or why not? What could you change to allow some time to pass between creating the vertex array and rendering?

```
public class RenderableTriangleMesh
{
  public RenderableTriangleMesh(Context context)
  {
    // Executes in worker thread

    Mesh mesh = /* ... */;

    _drawState = new DrawState();
    _drawState.ShaderProgram =
        Device.CreateShaderProgram(/* ... */);
    _meshBuffers = Device.CreateMeshBuffers(
        mesh, _drawState.ShaderProgram.VertexAttributes,
        BufferHint.StaticDraw);
  }

  public void Render(Context context, SceneState sceneState)
  {
    // Executes in rendering thread
    if (_meshBuffers != null)
    {
      _drawState.VertexArray =
          context.CreateVertexArray(_meshBuffers);
      _meshBuffers.Dispose();
      _meshBuffers = null;
    }

    context.Draw(PrimitiveType.Triangles,
                 _drawState, sceneState);
  }

  private readonly DrawState _drawState;
  private readonly MeshBuffers _meshBuffers;
}
```

Listing 10.6. If an object is created in one context and rendered in another, nonshared renderer resources need to be created in the rendering thread.

Synchronizing contexts in different threads. After sorting out how to create shared contexts and which objects are shared, the last consideration is synchronizing contexts. If there is only a single context, GL commands are processed in the order they are received. This means we can code naturally and write code that creates a resource, then renders with it, as shown in Figure 10.9. For example, if the resource is a texture, we can call Device.`CreateTexture2D`, which under the hood calls `glGenTexture`, `glBindTexture`, `glTexImage2D`, etc., then we can immediately issue a draw command with a shader using the texture. The updates to the texture are seen by the draw command.

When using multiple contexts, each context has its own command stream. It is still guaranteed that a change to an object at time t is completed when a command is issued at time $t+1$, as long as both commands are in the same context. If an object is changed at time t in one context and the CPU is synchronized (perhaps by posting a message to a

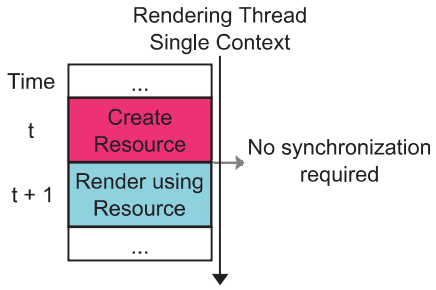


Figure 10.9. No CPU or GL synchronization is required when using a single context.

message queue), it is not guaranteed that a command issued in another context at time $t + 1$ sees the change (see Figure 10.10). For example, if a call to `glTexSubImage2D` is issued in a worker thread and then the thread posts a message to a message queue to notify the rendering thread, the rendering thread cannot issue a draw command with the guarantee that `glTexSubImage2D` is completed, even though the CPU was synchronized and `glTexSubImage2D` was called before the draw command.

Contexts can be synchronized in two ways. The simplest approach is to call `glFinish`, which is abstracted as Device.`Finish` in `OpenGlobe.Renderer`, after the resource is created, as shown in Figure 10.11. Although easy to implement, this synchronization approach can cause a significant performance hit because it requires a round-trip to the GL server. It blocks until the effects of all previous GL commands are realized in the GL client, server, and the framebuffer!

Fence sync objects, part of OpenGL 3.2 and also available through the ARB_sync extension [96], provide a more flexible way to synchronize contexts than `glFinish`. Instead of calling `glFinish`, create a fence object using `glFenceSync`. This inserts a fence into the context's command stream.

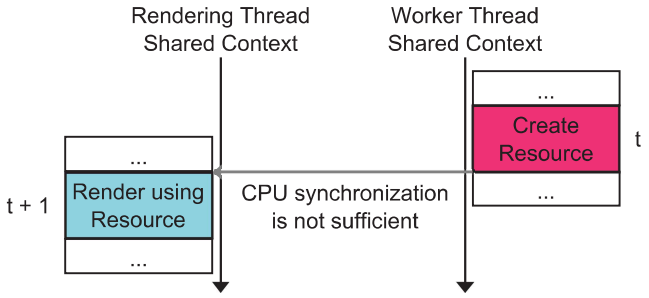


Figure 10.10. CPU synchronization is not sufficient to synchronize two contexts.

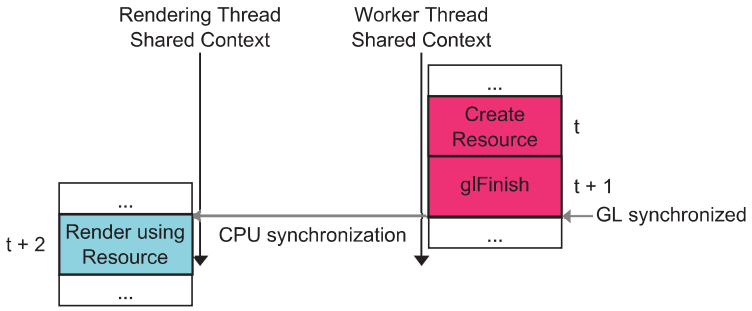


Figure 10.11. Synchronizing multiple contexts with `glFinish`.

Think of a fence as a barrier that separates commands before it and commands after it. To synchronize, call `glWaitSync` or `glClientWaitSync` to wait on the fence, that is, to block until all commands before the fence are completed.

> There are some reports of successfully using the much lighter `glFlush` instead of `glFinish` to synchronize contexts [160]. In fact, we've used it successfully in practice in Insight3D and STK. Unfortunately, this is not guaranteed to work. The OpenGL spec only mentions using `glFinish` or fences to determine command completion: "Completion of a command may be determined either by calling Finish, or by calling FenceSync and executing a WaitSync command on the associated sync object" (pp. 338–339) [152].
>
> ○○○○ Patrick Says

Two examples of synchronizing with fences are shown in Figure 10.12. In Figure 10.12(a), a fence is created in the worker thread after creating a resource. Before synchronizing the CPU with the rendering thread, `glFlush` is used to ensure the fence executes. Without flushing the command stream, it is possible that the fence would never execute, and a thread waiting on the fence would deadlock. After synchronizing the CPU, the rendering thread then waits on the fence to synchronize the GL. After waiting, a draw command can be issued with the assurance that resource creation in the other context completed.

A fence can be waited on in a few different ways. A call to `glWaitSync` can be used to block just the GL server until the fence is signaled. Blocking could occur on the CPU or GPU, an implementation detail; `glClientWaitSync` blocks the calling thread until the fence is signaled or a time-out

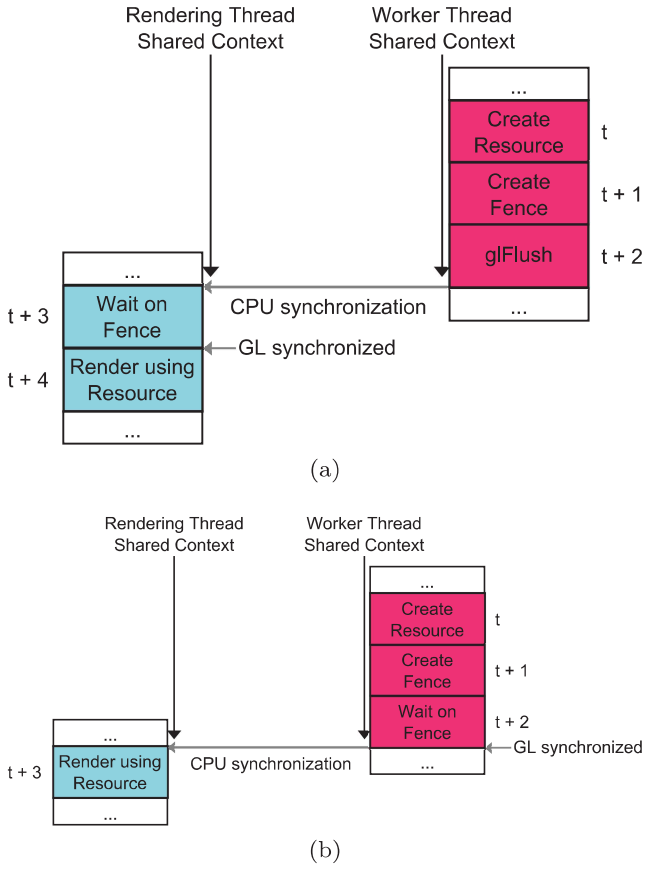


Figure 10.12. (a) GL synchronization by creating a fence in one context and waiting for it in another. (b) Creating a fence, then waiting on it for GL synchronization.

expires. When called with a time-out of zero, `glClientWaitSync` can be used to poll a fence. This is useful for the arrangement shown in Figure 10.12(b).

Here, a fence is created, then immediately waited on in the same context before synchronizing the CPU with the rendering thread. This is nearly identical to calling `glFinish`, as in Figure 10.11. If the client wait is done with a time-out of zero, the worker thread can continue to do other work, and occasionally poll the fence, until the resource creation completes.

Fences are more flexible than `glFinish` because `glFinish` waits for the entire command stream to complete; fences allow waiting on partial completion. The code in `OpenGlobe.Renderer` uses the abstract class Fence to represent fence sync objects. It has a straightforward implementation in `OpenGlobe.Renderer.GL3x`.FenceGL3x.

```
namespace OpenGlobe.Renderer
{
  public enum SynchronizationStatus
  {
    Unsignaled,
    Signaled
  }

  public enum ClientWaitResult
  {
    AlreadySignaled,
    Signaled,
    TimeoutExpired
  }

  public abstract class Fence : Disposable
  {
    public abstract void ServerWait();
    public abstract ClientWaitResult ClientWait();
    public abstract ClientWaitResult ClientWait(
        int timeoutInNanoseconds);
    public abstract SynchronizationStatus Status();
  }
}
```

10.4.4 Multithreaded Drivers

Some OpenGL drivers use multithreading internally to better utilize multicore CPUs [8]. OpenGL calls have some CPU overhead, including managing state and building commands for the GPU. Some calls even have significant CPU overhead, such as compiling and linking shaders. A multithreaded driver moves much of this CPU overhead from the application's rendering thread to a driver thread running on a separate core. When the application makes an OpenGL call, a command is put onto a command queue, which is later fully processed by the driver on a separate thread, and submitted to the GPU. Of course, some OpenGL calls, such as `glGet*`, require synchronization between the rendering and driver thread.

In the absence of a multithreaded driver, a similar multithreading approach can be taken by the application. The rendering thread can be split into a front-end thread that does culling and state sorting, and ultimately generates draw commands, and a back-end thread that executes these commands using OpenGL or Direct3D calls. This pipeline can increase throughput (i.e., frames per second), but it also increases latency as rendering will be a frame behind. The goal is the same as a multithreaded driver: to move the driver's CPU overhead to a separate core. By doing so with Direct3D 9, Pangerl saw improvements up to 200% in tests and 15% in an actual game [133].

Some care needs to be taken when splitting the rendering thread into front- and back-end threads. In particular, doing so can cause significant

memory overhead if data associated with draw commands (e.g., vertex buffers) are double buffered. The increased memory bandwidth and cache pollution can significantly degrade performance. Therefore, this design is not recommended on a single-core CPU. Front- and back-end threads were initially implemented in DOOM III but removed because the memory overhead came with a significant performance penalty on the single-core CPUs that were predominant at the time. Later, Quake 4 was released with two modes: one without a back-end thread and no double buffering for single-core CPUs and another with a back-end thread for multicore CPUs. This allowed a significant amount of work to be offloaded to a separate core on multicore systems without sacrificing any performance on single-core systems. When running on a single core, Kalra and van Waveren observed that about half the time in Quake 4 was spent in the OpenGL driver [80].

10.5 Resources

Herb Sutter's article "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software" does an excellent job of motivating the need to write parallel software [161]. The *CUDA Programming Guide* from NVIDIA discusses SIMT parallelism in their GPUs [122].

Van Waveren's work on texture streaming includes coverage of threaded pipelines, compression, and tactics for streaming from slow storage devices [172, 173].

Multithreading with OpenGL is a bit of a black art. Appendix D of the OpenGL 3.3 specification covers sharing objects among contexts [152]. It is valuable reading when using multiple threads with multiple shared contexts. The OpenGL spec also covers sync objects and fences, as does the ARB_sync extension [96]. The Hacks of Life blog contains several detailed posts on their experience with OpenGL multithreading in X-Plane [159]. MSDN covers multithreading with Direct3D 11, including resource creation and generating rendering commands from multiple threads using deferred contexts [114].

The Beyond Programmable Shading SIGGRAPH course notes are an excellent source of information on parallel programming architectures [97–99]. Next-generation game engines are moving to job-scheduling architectures to achieve scalability with an increasing number of cores [6, 100, 174]. Jobs are small, independent, stateless tasks that are organized into a graph based on their dependencies. An engine then schedules jobs using the graph. Think of this as taking fine-grained threading to the extreme.

Part IV

Terrain

11

Terrain Basics

Almost every graphics developer we know has written multiple terrain engines. Of course, there is good reason for this—terrain rendering is a fascinating area of computer graphics and the results are often stunning, beautiful landscapes with rolling hills and mountains. As shown in Figure 11.1, virtual globes have particularly impressive terrains given their use of real-world, high-resolution elevation data and satellite imagery.

Although fun and rewarding, developing a virtual globe-caliber terrain engine can be overwhelming. For this reason, we divide our discussion into four chapters. This first chapter covers the fundamentals: terrain representations, techniques for rendering height maps, computing normals, and a wide array of shading options. In order to understand terrain rendering from the bottom up, this chapter makes the simplifying assumptions that

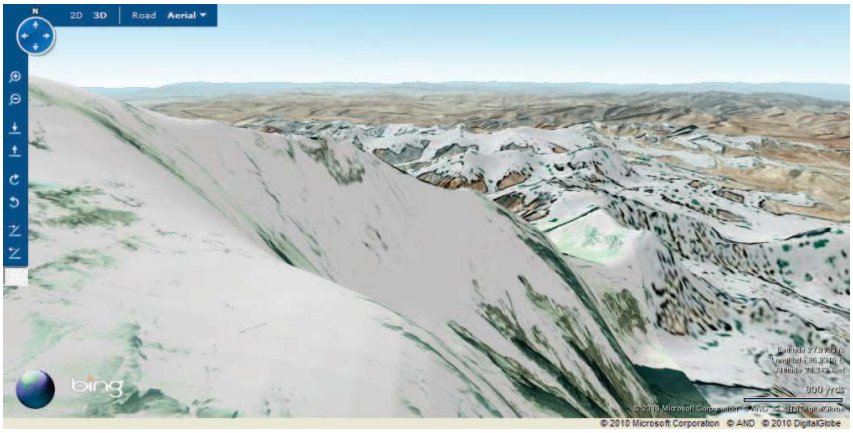


Figure 11.1. A screen shot of Bing Maps 3D showing real terrain and imagery for Mount Everest and surrounding peaks. © Microsoft Corporation © AND © 2010 DigitalGlobe.

the terrain is small enough to fit into memory and be rendered with brute force, and the terrain is extruded from a plane.

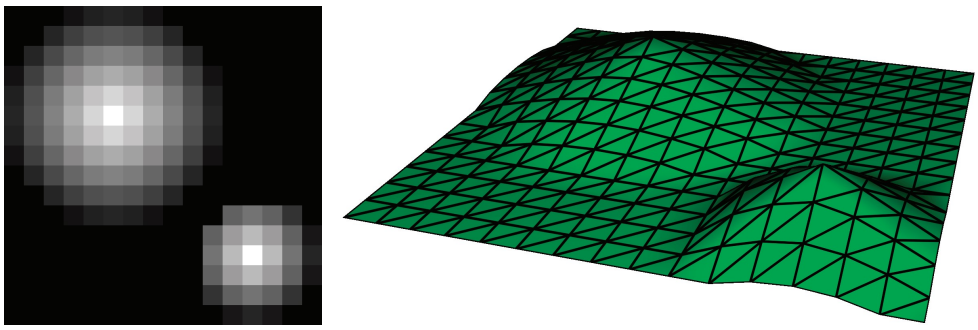
For some applications, these assumptions are perfectly valid. In particular, many games extrude terrain from a ground plane. Applications dealing with large datasets, however, cannot assume that the GPU is fast enough to render with brute force or that the terrain will fit into memory—or even on a single hard disk for that matter! Chapter 12 starts the discussion of massive terrain rendering, then Chapters 13 and 14 look at specific algorithms.

11.1 Terrain Representations

Terrain data come from a wide array of sources. Real-world terrain used in virtual globe and GIS applications is typically created using remote-sensing aircraft and satellites. Artificial terrains for games are commonly created by artists, or even procedurally generated in code. Depending on the source and ultimate use, terrain may be represented in different ways.

11.1.1 Height Maps

Height maps are the most widely used terrain representation. A height map, also called a *height field*, can be thought of as a grayscale image where the intensity of each pixel represents the height at that position. Typically, black indicates the minimum height and white indicates the maximum height.



(a) (b)

Figure 11.2. (a) A 16×16 grayscale image authored in a paint program. Height is indicated by intensity; black indicates minimum height and white indicates maximum height. (b) The image is used as a height map for terrain.

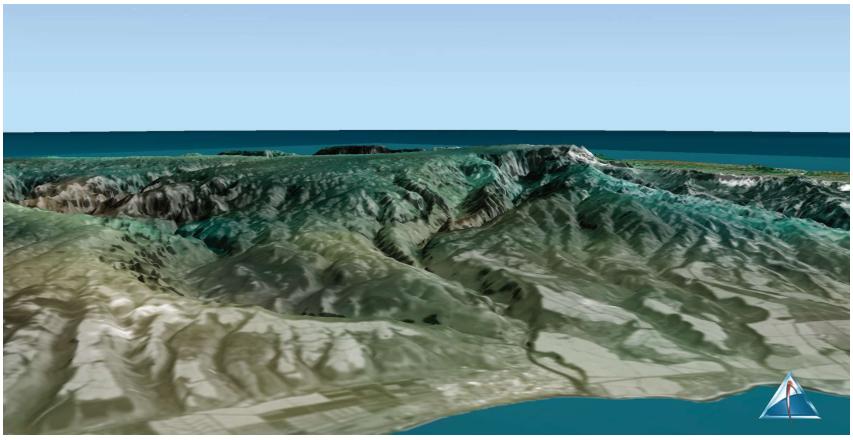


Figure 11.3. Kauai rendered using height-mapped terrain. (Image taken using STK.)

Imagine placing the height map in the ground plane, then extruding each pixel based on its height. The extruded pixels are called *posts*. In this chapter, the ground plane is the xy-plane, and height is extruded along z. It is also common to use the xz-plane with y being "up." Figure 11.2 shows a small 16×16 height map and terrain generated from it.

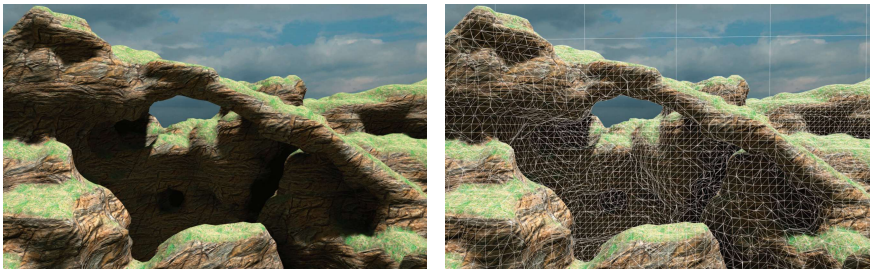
Height maps are popular because of their simplicity, the large amount of data available as height maps, and the wide array of tools for generating and modifying height maps. Even the gradient fill tool in free paint programs can be used to author simple terrains like the one in Figure 11.2(a).

Height maps are rendered in a wide variety of ways. This chapter will explore three approaches: creating a triangle mesh from a height map, using a height map as a displacement map in a vertex shader, and ray casting a height map in a fragment shader.

Height maps are sometimes called 2.5D. Since only one height per xy-position is stored, a single height map cannot represent terrain features such as vertical cliffs, overhangs, caves, tunnels, and arches. However, we can still produce stunning terrains with height maps (see Figures 11.1 and 11.3). Vertical cliffs can only be approximated by a high-resolution height map with a significant change in height between adjacent posts where the vertical cliff is located. Overhung features, including caves and tunnels, are usually handled using separate models.

11.1.2 Voxels

An alternative terrain representation to height maps that supports truly vertical and overhung features is *voxels*. Intuitively, a voxel is the 3D



(a) (b)

Figure 11.4. (a) A screen shot of the C4 Engine showing overhangs that can be achieved with voxels. (b) The triangle mesh generated from the voxel representation. (Images courtesy of Eric Lengyel, Terathon Software LLC.)

extension of a pixel. Just as a pixel is a picture element in two dimensions, a voxel is a volume element in three dimensions. Think of a volume as a 3D grid or bitmap, and each cube in the grid, or pixel in the bitmap, is a voxel. In the simplest case, a voxel is binary: a 1 represents solid and a 0 represents empty space. In many real-world cases, a voxel is more than binary, commonly storing density and material.

When terrain is represented using voxels, multiple heights per xy-position are possible. This easily allows vertical cliffs and overhung features, as shown in Figure 11.4.

A downside to voxels is their cubic memory requirements for uniform volumes. A $512 \times 512 \times 512$ volume, with each voxel being 1 byte, requires over 128 MB; whereas a 512×512 height map, with each pixel being one float, requires only 1 MB. Fortunately, several techniques can be used to decrease voxel memory requirements. Fewer bits per voxel can be used; in the extreme case, a single bit per voxel can suffice for simple solid/empty cases. Voxels can also be grouped into larger (e.g., 4×4 or 8×8) cells that can be marked completely solid or empty, therefore avoiding storage of individual voxels for nonboundary cases. Likewise, materials can be assigned per cell instead of per voxel. These two techniques were used for the destructible voxel terrain in Miner Wars [145]. Finally, voxels can be stored using hierarchical data structures, such as sparse octrees, such that branches of the hierarchy containing solid or empty space do not consume memory [129].

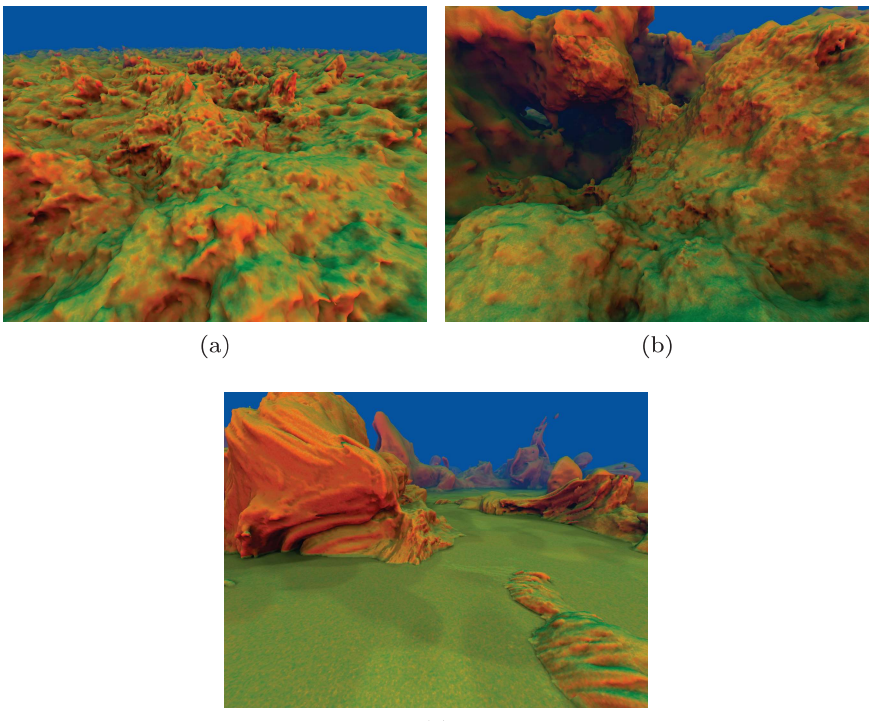
Once modeled, voxels may be rendered directly or converted to a triangle mesh for rendering. GPU ray casting is a recent technique for directly rendering voxels [35, 129]. Alternatively, an algorithm such as marching cubes may be used to generate a triangle mesh from a voxel representation [104].

Voxels are becoming an attractive terrain representation in games because of their artistic control. Some game engines are starting to use voxels for terrain editing. Unfortunately for virtual globes, we are unaware of any real terrain data available as voxels.

11.1.3 Implicit Surfaces

There is a whole area of terrain rendering, called *procedural terrain*, that does not rely on real-world or even artist-created terrain. In fact, very little about the terrain is stored. Instead, the terrain surface is described by an implicit function used to procedurally create the terrain at runtime.

For example, Geiss uses a density function for generating procedural terrain on the GPU [59]. Given a point, (x, y, z), the function returns a positive value for solid and a negative value for empty space. The boundary between positive and negative values describes the terrain's surface. For example, to make the terrain surface the plane $z = 0$, the density function is



(a) (b)

(c)

Figure 11.5. Terrain procedurally generated on the GPU using a density function based on noise. (c) A floor is added by increasing the density for positions below a certain z-coordinate. (Images courtesy of NVIDIA Corporation and Ryan Geiss.)

density $= -z$. By combining multiple octaves of noise, each with different frequencies and amplitudes, a wide array of interesting terrains can be generated (see Figure 11.5). Many customizations are possible, such as adjusting the density based on z to create a floor, shelves, or terraces. Similar to voxels, marching cubes can be used to create a triangle mesh for rendering an implicit function.

Procedural generation can produce very stunning terrain with complex overhanging features using a minimal amount of memory. Given the trends in GPU architecture, procedural generation will likely have a bright future in games and simulations. Procedural generation for real terrains is quite challenging to say the least—we are unaware of any endeavors.

11.1.4 Triangulated Irregular Networks

A *triangulated irregular network (TIN)* is basically a triangle mesh. TINs are formed by triangulating a point cloud to create a watertight mesh. Due to the high-resolution TINs available for some geographic regions, TINs are becoming popular in GIS applications.

Point clouds for TINs are commonly retrieved by aircraft using *light detection and ranging (LiDAR)*. LiDAR is a remote-sensing method that samples Earth's surface using timed pulses of laser light. The time between emitting light and receiving its reflection is converted to distance and then to a point in the point cloud based on parameters such as the aircraft and sensor orientation. Since the reflected pulses may include buildings,

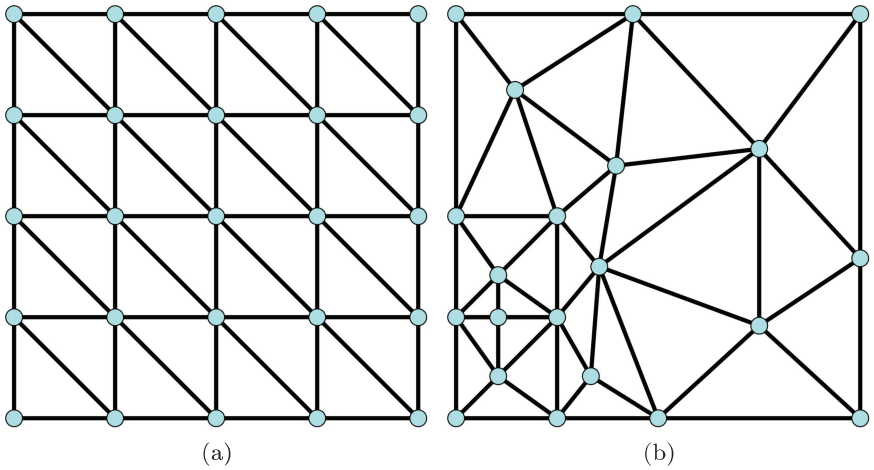


Figure 11.6. (a) A standard triangulation of a height map results in a uniform grid. (b) TINs allow for nonuniform sampling. The lower-left part of the triangulation has the highest concentration of triangles.

vegetation, etc., filtering algorithms and manual post-processing are used to create a bare-Earth terrain model.

As shown in Figure 11.6, compared to the uniform structure of height maps, TINs allow for nonuniform sampling; large triangles can cover flat regions and small triangles can represent fine features. Unlike other terrain representations, a TIN is generally ready to render without any additional processing, provided it fits into memory. Given its triangle-mesh nature, a TIN representation allows for vertical and overhung features.

11.1.5 Summary of Terrain Representations

Table 11.1 summarizes the discussed terrain representations. Different representations may be used together or even converted to the same representation for rendering. For example, a height map may be used for the base terrain and voxels may be used when only overhangs are required. A height map may resemble a TIN after it is triangulated and simplified. Likewise, triangle meshes may be derived from voxels or implicit surfaces.

Given their widespread use, we focus on using height maps for terrain rendering.

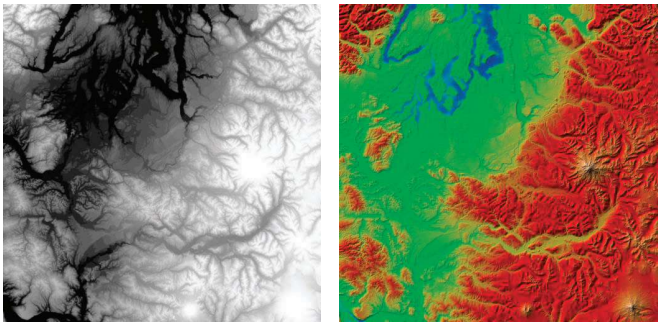
Height map	Most widely used. Simple to edit, simple to render. A large number of tools and datasets are readily available. Cannot represent vertical cliffs and overhung features.
Voxels	Excellent artistic control, including support for vertical cliffs and overhung features. Not as straightforward to render as height maps or TINs.
Implicit surface	Use very little memory. Used for procedural terrain to generate complex terrains on the fly. Not as straightforward to render as height maps or TINs.
TIN	High-resolution data are becoming available. Its nonuniform grid can make better use of memory than a height map. Easy to render.

Table 11.1. Summary of terrain representations.

11.2 Rendering Height Maps

We will cover three approaches to rendering height-map-based terrain. We'll start with the most obvious: rendering a triangle mesh created from a height map on the CPU. The next two approaches more fully utilize the GPU. One approach uses a height map as a displacement map in a vertex shader and the other casts rays through a height map in a fragment shader.

Throughout this chapter, we use a small 512×512 height map of Puget Sound in Washington state, shown in Figure 11.7.



(a) (b)

Figure 11.7. (a) A low-resolution height map and (b) *color map* (texture) of Puget Sound from the Large Geometric Models Archive at the Georgia Institute of Technology.

In addition to data for heights, height maps typically have associated metadata used to determine how to position the terrain in world space. This chapter uses a class with the following public interface to represent a height map and its metadata:

```
public class TerrainTile
{
  // Constructors ...
  public RectangleD Extent { get; }
  public Vector2I Resolution { get; }
  public float[] Heights { get; }
  public float MinimumHeight { get; }
  public float MaximumHeight { get; }
}
```

The property `Extent` is the height map's world-space boundary in the xy-plane. The property `Resolution` is the x- and y-resolution of the height map, similar to an image's resolution. The property `Heights` includes the actual height-map values in bottom to top row major order. Figure 11.8 shows this layout for a 5×5 height map. This class is named TerrainTile because, as we shall see in Chapter 12, terrain is typically stored in multiple sections, called *tiles*, for efficient culling, level of detail, and paging.

11.2.1 Creating a Triangle Mesh

Given an instance of a TerrainTile, creating a uniform triangle mesh is straightforward. As shown in Figure 11.9, imagine the process by looking straight down at the height map. Create a vertex at each pixel location

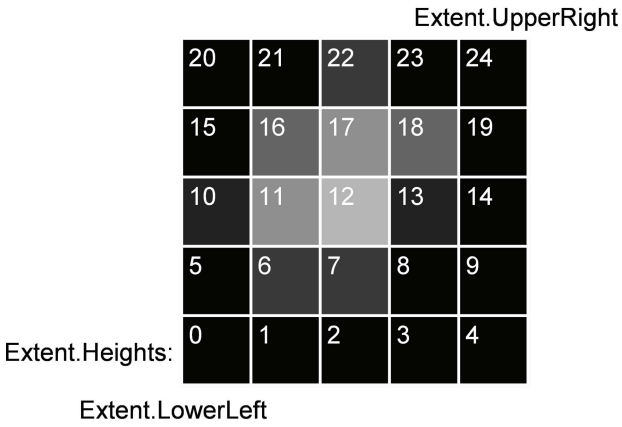


Figure 11.8. Extent.`Heights` stores heights row by row, bottom to top.

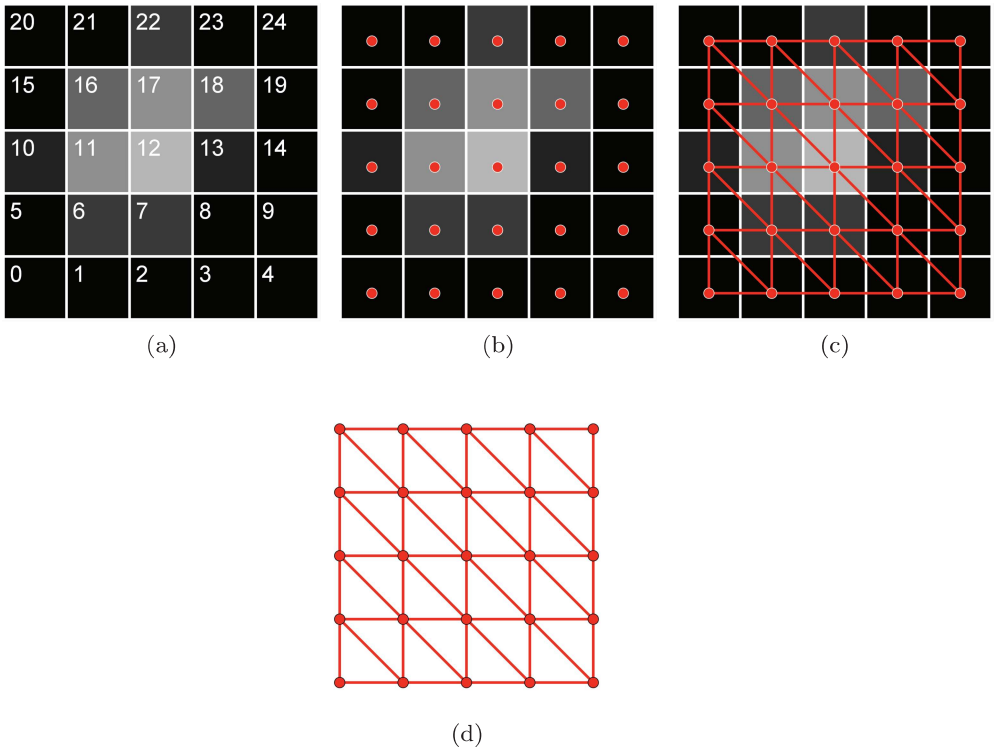


Figure 11.9. (a) A 5×5 height map. (b) Top-down view of vertices. (c) Triangles connecting vertices. (d) Triangle mesh only.

```
Mesh mesh = new Mesh();
mesh.PrimitiveType = PrimitiveType.Triangles;
mesh.FrontFaceWindingOrder = WindingOrder.Counterclockwise;

int numberOfPositions = tile.Resolution.X * tile.Resolution.Y;
VertexAttributeDoubleVector3 positionsAttribute =
    new VertexAttributeDoubleVector3("position",
                                     numberOfPositions);
IList<Vector3D> positions = positionsAttribute.Values;
mesh.Attributes.Add(positionsAttribute);

int numberOfPartitionsX = tile.Resolution.X - 1;
int numberOfPartitionsY = tile.Resolution.Y - 1;
int numberOfIndices =
    (numberOfPartitionsX * numberOfPartitionsY) * 6;
IndicesUnsignedInt indices =
    new IndicesUnsignedInt(numberOfIndices);
mesh.Indices = indices;
```

Listing 11.1. Initializing a mesh for a height map.

using the height of the pixel, then connect surrounding vertices with triangle edges. Once the mesh is created, the height map is no longer required for rendering, but it can be useful for other tasks such as collision detection or reporting the height under the mouse pointer.

An xy-height map yields $x \times y$ vertices and $2\,(x-1)\,(y-1)$ triangles. Given a `tile`, a mesh can be initialized with enough memory for these using the code in Listing 11.1.

The next step is to create one vertex per height-map pixel. A vertex's xy-position is computed by offsetting the tile's lower left position based on the pixel's location in the height map. The vertex's z-coordinate is taken directly from the height map. A simple nested for loop suffices, as shown in Listing 11.2.

```
Vector2D lowerLeft = tile.Extent.LowerLeft;
Vector2D toUpperRight = tile.Extent.UpperRight - lowerLeft;
int heightIndex = 0;
for (int y = 0; y <= numberOfPartitionsY; ++y)
{
  double deltaY = y / (double)numberOfPartitionsY;
  double currentY = lowerLeft.Y + (deltaY * toUpperRight.Y);

  for (int x = 0; x <= numberOfPartitionsX; ++x)
  {
    double deltaX = x / (double)numberOfPartitionsX;
    double currentX = lowerLeft.X + (deltaX * toUpperRight.X);
    positions.Add(new Vector3D(
        currentX, currentY, tile.Heights[heightIndex++]));
  }
}
```

Listing 11.2. Computing vertex positions from a height map.

```
int rowDelta = numberOfPartitionsX + 1;
int i = 0;
for (int y = 0; y < numberOfPartitionsY; ++y)
{
  for (int x = 0; x < numberOfPartitionsX; ++x)
  {
    indices.AddTriangle(new TriangleIndicesUnsignedInt(
        i, i + 1, rowDelta + (i + 1)));
    indices.AddTriangle(new TriangleIndicesUnsignedInt(
        i, rowDelta + (i + 1), rowDelta + i));
    i += 1;
  }
  i += 1;
}
```

Listing 11.3. Computing triangle indices from a height map.

Vertices are stored sequentially row by row, bottom to top, with the same indices as `tile.Heights` (see Figure 11.9(a)). The final step is to create indices for the actual triangles. For each pixel, except pixels in the top row and right column, create two triangles forming a quad between the pixel, the pixel to the right, the pixel to the upper right, and the pixel above, as done in Listing 11.3.

A complete implementation of creating a triangle mesh from a height map is provided in `OpenGlobe.Scene`.TriangleMeshTerrainTile. Once the triangle mesh is created, rendering is as simple as issuing a draw call. The original height map can be forgotten about if it is not needed for other tasks.

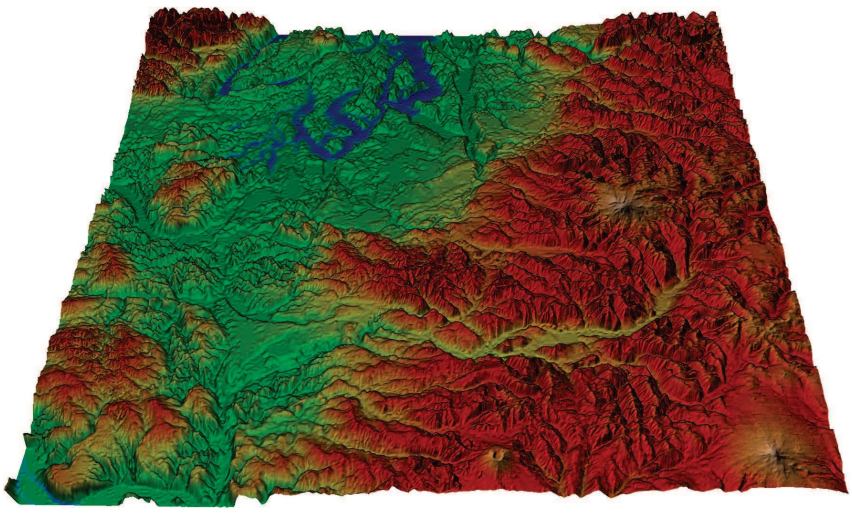
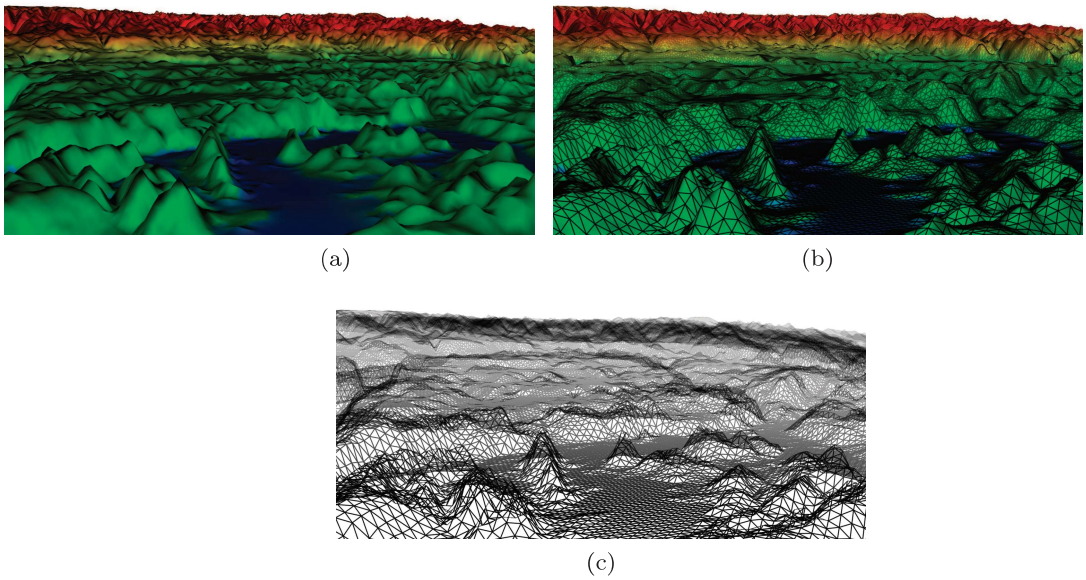


Figure 11.10. The height and color maps of Figure 11.7 rendered with a triangle mesh.



(a) (b)

(c)

Figure 11.11. (a) Zoomed-in view of terrain. (b) Wireframe overlay. (c) Wireframe only.

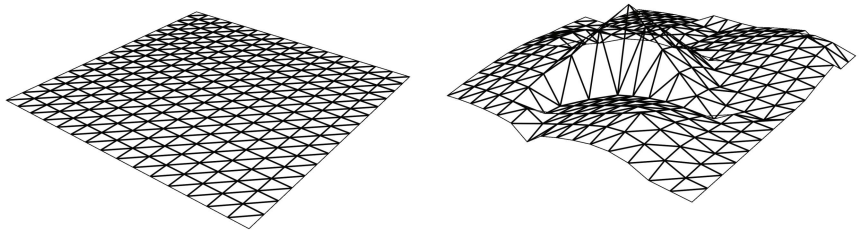
An advantage of this uniform layout is, if multiple tiles are rendered, memory can be reduced since each tile can use the same index buffer. To improve rendering performance, indices can be reordered into a cache-coherent layout [50, 148].

Compare Figure 11.10 to the original height map and color map in Figure 11.7. Dark height-map pixels correspond to low heights and bright pixels correspond to higher heights.

Figure 11.11 shows a zoomed-in view of the same terrain, including a wireframe version. The wireframe shows two important points regarding efficiency. First, the mountains in the distance are drawn with an awful lot of triangles. In fact, far fewer triangles could be drawn and the scene would look almost identical. This is not specific to rendering height maps with triangle meshes; the other brute-force height-map-rendering techniques in this chapter have similar problems. Second, the wireframe reveals that flat regions, like the blue area in the foreground, have many unnecessary triangles. This is the result of creating a uniform grid from a height map. In Chapter 12, we discuss level of detail, which can minimize both of these issues.

11.2.2 Vertex-Shader Displacement Mapping

For many years, terrain was rendered using static triangle meshes. When the ability to read from textures was added to vertex shaders, a new GPU



(a) (b)

Figure 11.12. (a) Uniform grid in the xy-plane. (b) Terrain is created by displacing grid vertices in the vertex shader based on the height-map texture.

approach became popular [177]. In this approach, a uniform triangle mesh with vertices corresponding to height-map pixels is created as previously described, the difference being that the z-coordinate is not stored in the mesh; only the position in the xy-plane is stored. The height map is written to a one-component floating-point texture, which is sampled in the vertex shader based on the vertex's xy-position. The height map is treated as a *displacement map* applied to the uniform tessellation of the xy-plane, as in Figure 11.12.

At first glance, the advantages of this approach may not be obvious, but there are many:

- *Simplicity.* The implementation is easy, requiring only a one-line vertex shader to do the height map lookup:

```
in vec2 position;
uniform mat4 og_modelViewPerspectiveMatrix;
uniform sampler2DRect u_heightMap;

void main()
{
  gl_Position = og_modelViewPerspectiveMatrix *
      vec4(position, texture(u_heightMap, position).r, 1.0);
}
```

- *Flexibility.* It is easy to implement destructible terrain by modifying a subset of the height-map texture or adding an additional destruction texture. Although not common in virtual globes, destructible terrain is used in games in response to explosions.

- *Low memory usage.* When only a single tile is rendered, displacement mapping uses a similar amount of memory as a uniform static triangle mesh. What about when several tiles are rendered? A height map for

each tile is required, but the triangle mesh can be reused by modifying the model matrix to translate and scale it to overlap the tile's world-space extent. This means that additional tiles can be rendered with only the memory costs of a one-component floating-point texture, instead of a three-component vertex buffer.

Of course there are a few downsides to using vertex-shader displacement mapping:

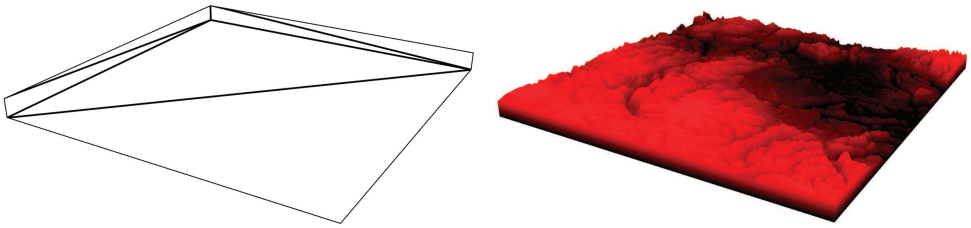
- *Extra texture reads.* An extra texture read is required per vertex. On modern GPUs, these are quite efficient. Even an old NVIDIA GeForce 6800 is capable of 33 million vertex texture reads per second compared to its peak processing of 600 million vertices per second [60].
- *Simplification limitations.* The uniform nature of the height map and triangle mesh allows reusing the same mesh for each tile. But the uniform nature also prevents simplifying flat regions that do not require fine detail.

For many applications, the benefits of vertex-shader displacement mapping far outweigh its downsides, which explains its popularity in game engines such as Frostbite [4].

11.2.3 GPU Ray Casting

The final approach to rendering height-map-based terrain we will consider is GPU ray casting. This algorithm is similar to ray casting an ellipsoid for globe rendering, discussed in Section 4.3. The height map is stored in a one-component floating-point texture. Terrain is rendered by rendering a tile's axis-aligned bounding box (AABB) to invoke a fragment shader that casts rays from the eye through the fragment, shading fragments with rays that intersect the height map and discarding those that miss. The advantages of this approach are the following:

- *Reduced memory usage.* Rendering a tile only requires the tile's height map and AABB. Therefore, more tiles can fit into GPU memory and there is less system bus traffic. This is an important advantage given the widening performance gap between processing power and memory access. Using less memory directly attacks the memory bandwidth bottleneck.
- *Reduced vertex processing.* Since only an AABB is rendered per tile, there is very little vertex shading and triangle setup overhead. On today's GPUs with unified shader architectures, this frees up more GPU resources to process fragments.



(a) (b)

Figure 11.13. (a) The back faces of a tile's AABB are rendered to invoke a ray-casting fragment shader. (b) Rays step through the height map to find the visible terrain surface.

The downsides include the following:

- *Complexity.* Ray casting a height map is not as easy to implement as rendering a static mesh or displacement mapping. It is also not as simple as the ellipsoid ray casting we've already seen, which has an analytic solution; ray casting a height map requires stepping the ray through the height map. Fortunately, the complexity is isolated in the fragment shader, making the CPU code straightforward.
- *Increased fragment processing.* Since the ray-casting fragment shader is complex and may require many texture reads, it can require a significant amount of processing.

The question is, does the reduced memory usage and vertex processing offset the complex fragment processing? We'll need to take a closer look at the algorithm to answer this.

Our implementation closely follows the work of Dick et al. [37]. To begin, a triangle mesh for a tile's AABB is created. This AABB is rendered with front-face culling, as in Figure 11.13(a). If back-face culling were used, the terrain would disappear when the viewer enters the AABB because no fragments would be rasterized.

Only the simple vertex shader of Listing 11.4 is required. The world-space position, `position.xyz`, is passed to the fragment shader as `boxExit`. After interpolation, this is the point where a ray cast from the eye through the fragment will exit the AABB. Since only back faces are rendered, we are sure this is the exit point, not the entry point. Computing the ray's entry point is the first task of the fragment shader. Then the fragment shader does the heavy lifting:

- The ray is stepped across height-map pixels, called *texels* here since the height map is stored in a texture, stopping at the first intersection.

```
in vec4 position;
out vec3 boxExit;
uniform mat4 og_modelViewPerspectiveMatrix;

void main()
{
    gl_Position = og_modelViewPerspectiveMatrix * position;
    boxExit = position.xyz;
}
```

Listing 11.4. Vertex shader for GPU ray casting.

```
in vec3 boxExit;
uniform sampler2DRect u_heightMap;
uniform vec3 u_aabbLowerLeft;
uniform vec3 u_aabbUpperRight;
uniform vec3 og_cameraEye;
```

Listing 11.5. GPU ray casting fragment shader inputs.

- If an intersection is found, the intersection point's depth is computed and the fragment is shaded; otherwise, it is discarded.

Except for the ray's exit point, `boxExit`, the fragment shader takes all of its inputs from uniform variables, as shown in Listing 11.5.

The height map is stored in a rectangle texture, which allows for unnormalized texture coordinates. This simplifies the shader since world space xy-positions more easily line up with texel st-coordinates. The AABB corners are `u_aabbLowerLeft` and `u_aabbUpperRight`. Given the ray's exit point and eye position, the ray's direction is simply `boxExit` − `og_cameraEye`.

Instead of stepping along the ray starting at the eye, it is more efficient to analytically compute the ray's intersection with the front of the AABB and start stepping the ray from there. Of course, if the viewer is inside the

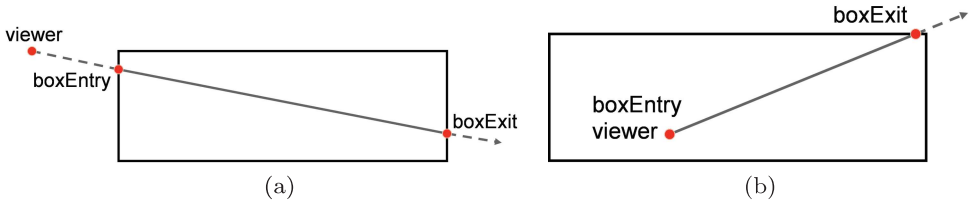


Figure 11.14. Side views of a ray and a tile's AABB. Only the segment of the ray that intersects the AABB needs to be considered. (a) Viewer outside the AABB. (b) Viewer inside the AABB.

```
struct Intersection
{
  bool Intersects;
  vec3 IntersectionPoint;
};

bool PointInsideAxisAlignedBoundingBox(vec3 point,
    vec3 lowerLeft, vec3 upperRight)
{ // ... }

Intersection RayIntersectsAABB(vec3 origin, vec3 direction,
    vec3 aabbLowerLeft, vec3 aabbUpperRight)
{ // ... }

void main()
{
  vec3 direction = boxExit - og_cameraEye;

  vec3 boxEntry;
  if (PointInsideAxisAlignedBoundingBox(og_cameraEye,
      u_aabbLowerLeft, u_aabbUpperRight))
  {
    boxEntry = og_cameraEye;
  }
  else
  {
    Intersection i = RayIntersectsAABB(og_cameraEye, direction,
        u_aabbLowerLeft, u_aabbUpperRight);
    boxEntry = i.IntersectionPoint;
  }

  // ...
}
```

Listing 11.6. Computing the ray's entry point.

AABB, then starting at the eye is appropriate. These two cases are shown in Figure 11.14. Since the fragments were rasterized using the AABB, the ray is guaranteed to intersect the AABB, but it is not necessarily true that the ray will intersect the height map. See Listing 11.6 for the portion of the fragment shader used to compute the ray's entry point, `boxEntry`.

Next, the ray is stepped through the height map in texel space. For simplicity, we assume texel space maps directly to the xy-plane in world space—hence the use of a rectangle texture for the height map. This also assumes the lower-left xy-world position of the tile is $(0, 0)$.

Ray stepping starts at `boxEntry` and ends when an intersection is found or the end of the height map is reached. To simplify the ray-stepping logic, mirroring is used so the direction is monotonically increasing in x and y as the ray steps over texels. If `direction.x` and `direction.y` are nonnegative, nothing is mirrored (see Figure 11.15). Otherwise, if either are negative, the component is negated (e.g., `direction.x` $= -$`direction.x`). To compensate, the corresponding component(s) of the entry point and texture coordinates are mirrored. The relevant fragment-shader snippet

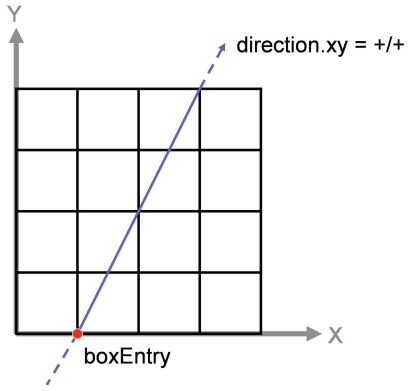


Figure 11.15. When `direction.x` and `direction.y` are nonnegative, no mirroring is required.

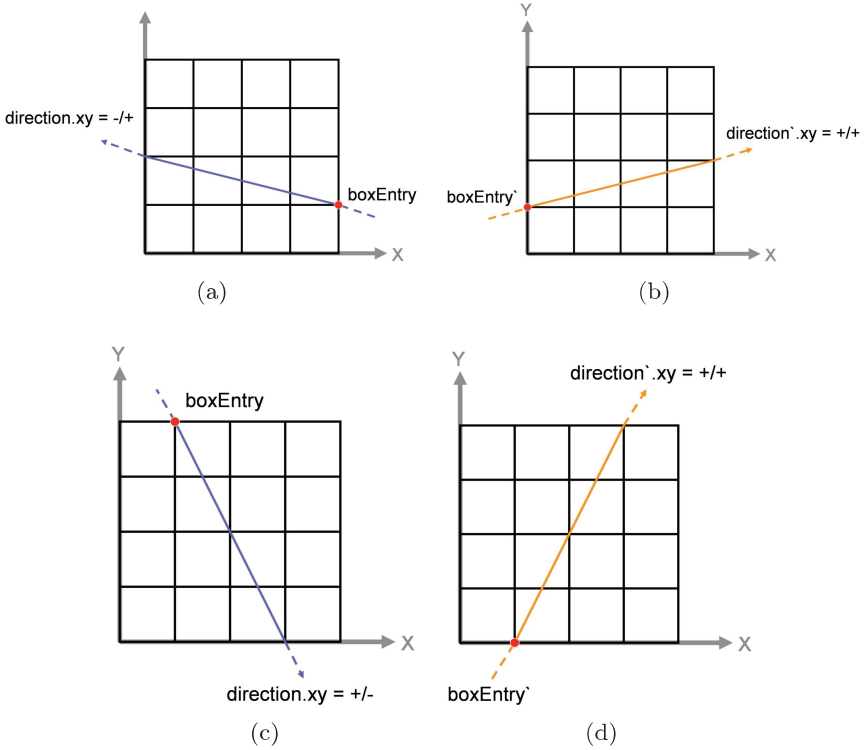


Figure 11.16. In (a), `direction.x` is negative. This is compensated for by mirroring in (b). The same scenario is shown for `direction.y` in (c) and (d).

is shown in Listing 11.7. The adjustments to `direction` and `boxEntry` are shown graphically in Figure 11.16.

After mirroring adjustments, we are ready to step from texel to texel checking for intersection with the height map. The process is similar to line rasterization. Two new variables are used: `texEntry`, the entry point of the ray with the current texel, and `texExit`, the exit point of the ray with the current texel. Initially, `texEntry` is set to the AABB's entry point. As the ray steps from texel to texel, the previous texel exit point becomes the current entry point, as shown in Figure 11.17.

Since mirroring was performed, a ray always enters a texel through the left or bottom edge and exits through the right or top edge. The texel exit point is found using the code in Listing 11.8.

The lower-left corner of the texel is `floorTexEntry`, which makes `floorTexEntry + vec2(1.0)` the upper-right corner. This is used to compute `delta`, the distances along the ray to the right and top edges. The smaller

```
void main()
{
  // ...

  vec2 heightMapResolution = vec2(textureSize(u_heightMap));
  bvec2 mirror = lessThan(direction.xy, vec2(0.0));
  vec2 mirrorTextureCoordinates = vec2(0.0);

  if (mirror.x)
  {
    direction.x = -direction.x;
    boxEntry.x = heightMapResolution.x - boxEntry.x;
    mirrorTextureCoordinates.x = heightMapResolution.x - 1.0;
  }

  if (mirror.y)
  {
    direction.y = -direction.y;
    boxEntry.y = heightMapResolution.y - boxEntry.y;
    mirrorTextureCoordinates.y = heightMapResolution.y - 1.0;
  }

  // ...
}
```

Listing 11.7. Mirroring the ray direction.

```
vec2 floorTexEntry = floor(texEntry.xy);
vec2 delta = ((floorTexEntry + vec2(1.0)) - texEntry.xy) /
             direction;
vec3 texExit = texEntry + (min(delta.x, delta.y) * direction);
```

Listing 11.8. Computing `texExit`.

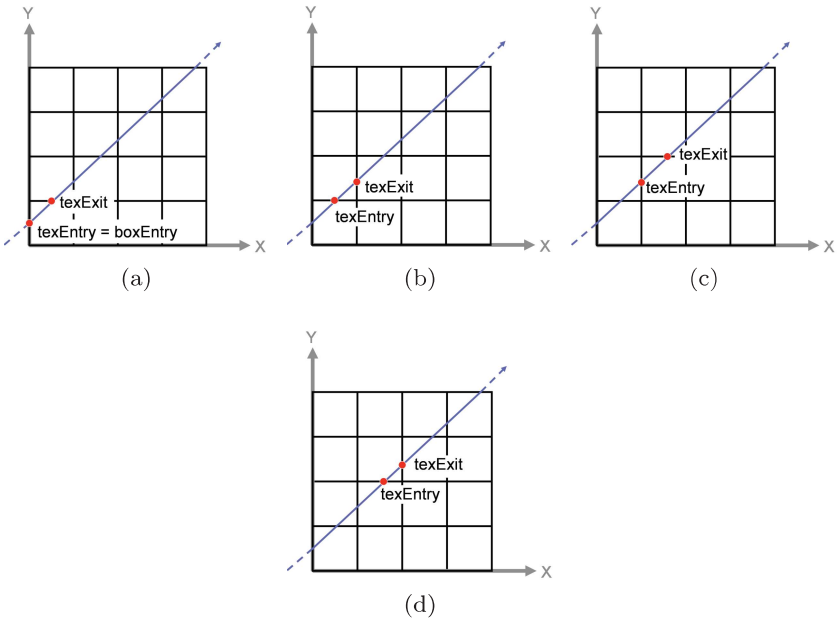


Figure 11.17. `texEntry` and `texExit` for the first four ray steps.

of `delta.x` and `delta.y` is used to offset along the ray, selecting the closer of the two potential exit points, as shown in Figure 11.18.

Now that we know how to step from texel to texel, computing the entry and exit points, the next step is to check for intersection with the height map. If no mirroring was used, the height is looked up using `floorTexEntry` as the texture coordinate. Otherwise, mirrored repeat is emulated as done in Listing 11.9.

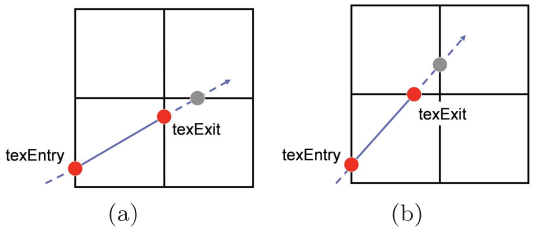


Figure 11.18. The shorter of the two distances along the ray from `texEntry` to the texel's right and top edges is used to find `texExit`. In (a), the right edge is closer. In (b), the top edge is.

```
vec2 MirrorRepeat(vec2 textureCoordinate,
                  vec2 mirrorTextureCoordinates)
{
  return vec2(mirrorTextureCoordinates.x == 0.0
        ? textureCoordinate.x
        : mirrorTextureCoordinates.x - textureCoordinate.x,
      mirrorTextureCoordinates.y == 0.0 ? textureCoordinate.y
        : mirrorTextureCoordinates.y - textureCoordinate.y);
}

// ...

vec2 floorTexEntry = floor(texEntry.xy);
float height = texture(u_heightMap, MirrorRepeat(floorTexEntry,
    mirrorTextureCoordinates)).r;
```

Listing 11.9. Looking up a texel's height.

Given the texel's height, an intersection occurs if the ray goes under the texel. Two intersection cases need to be considered. If the ray is heading upwards (`direction.z` >= 0), the ray intersects the texel if `texEntry.z` < `height`, as shown in Figure 11.19(a). In this case, the intersection point is `texEntry`. A more common view for many virtual globe users is when the ray is heading downwards (`direction.z` <= 0). Here, an intersection occurs if `texExit.z` <= `height`, as in Figure 11.19(b), with an intersection point of `texEntry` + (`max`)(((`height` − `texEntry.z`)/ `direction.z`, 0.0) ∗ `direction`).

Listing 11.10 brings together the ray-stepping algorithm and intersection test to build a loop that steps across texels until an intersection is found.

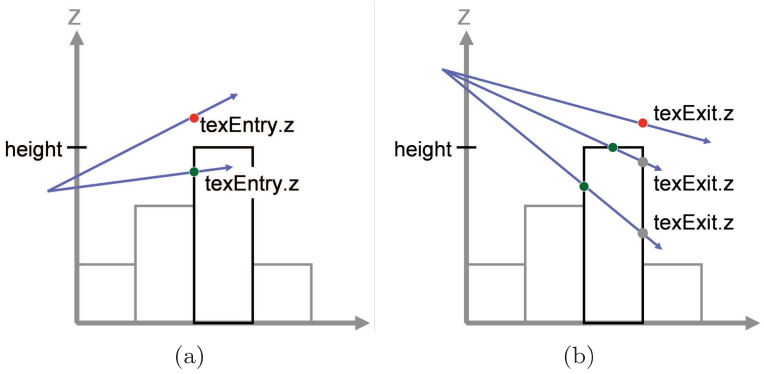


Figure 11.19. (a) Upward-heading ray intersecting a texel. (b) Downward-heading ray. Intersects are green and misses are red.

```
void main()
{
  // ...

  vec3 texEntry = boxEntry;
  vec3 intersectionPoint;
  bool foundIntersection = false;

  while (!foundIntersection &&
         all(lessThan(texEntry.xy, heightMapResolution)))
  {
    foundIntersection = StepRay(direction,
        mirrorTextureCoordinates, texEntry, intersectionPoint);
  }

  // ...
}
```

Listing 11.10. Ray-stepping loop.

As stated previously, `texEntry` is initialized with the intersection point of the ray and the AABB. The ray-stepping loop continues until an intersection is found or the end of the tile is reached. The bulk of the work is done in the `StepRay` function in Listing 11.11.

This code should seem familiar. The texel's height is read. The exit point, `texExit`, is computed. Then, the actual intersection test occurs. Note how `texEntry` is set to `texExit` for the next iteration of the loop. Also observe that during the loop, the sign of `direction.z` never changes because the ray does not change direction. Therefore, the (`direction.z` >= 0.0) check can be moved outside of the loop. For that matter, the conditionals in `MirrorRepeat` can be moved as well.

```
bool StepRay(vec3 direction, vec2 mirrorTextureCoordinates,
             inout vec3 texEntry, out vec3 intersectionPoint)
{
  vec2 floorTexEntry = floor(texEntry.xy);
  float height = texture(u_heightMap, MirrorRepeat(
      floorTexEntry, mirrorTextureCoordinates)).r;

  vec2 delta = ((floorTexEntry + vec2(1.0)) - texEntry.xy) /
               direction.xy;
  vec3 texExit = texEntry + (min(delta.x, delta.y) * direction);

  if (delta.x < delta.y)
  {
    texExit.x = floorTexEntry.x + 1.0;
  }
  else
  {
    texExit.y = floorTexEntry.y + 1.0;
  }

  bool foundIntersection = false;
```

```
  if (direction.z >= 0.0)
  {
    if (texEntry.z <= height)
    {
      foundIntersection = true;
      intersectionPoint = texEntry;
    }
  }
  else
  {
    if (texExit.z <= height)
    {
      foundIntersection = true;
      intersectionPoint = texEntry +
          max((height - texEntry.z) / direction.z, 0.0) *
          direction;
    }
  }

  texEntry = texExit;
  return foundIntersection;
}
```

Listing 11.11. `StepRay` function.

Since the ray is always going to exit a texel at either the right or top edge, one of the `texExit` components will be exactly one greater than the corresponding `floorTexEntry` component. Although the math for the initial assignment to `texExit` works out this way, Dick et al. warn that roundoff errors can occur, leading to infinite loops. After locking up my computer a few times, I added the explicit assignment suggested by Dick et al. [37].

Patrick Says

If an intersection is found, the fragment is shaded and a depth value is computed using the same technique as for ray casting an ellipsoid in Section 4.3. This allows normal rasterized geometry to interact correctly with the ray-casted terrain. The intersection point needs to be adjusted if mirroring was used. If the ray-stepping loop exits without finding an intersection, the fragment is discarded. The end of the fragment shader is shown in Listing 11.12.

```
void main()
{
  // ...

  if (foundIntersection)
  {
    if (mirror.x)
```

```
    {
      intersectionPoint.x =
          heightMapResolution.x - intersectionPoint.x;
    }

    if (mirror.y)
    {
      intersectionPoint.y =
          heightMapResolution.y - intersectionPoint.y;
    }

    fragmentColor = // shade however you like
    gl_FragDepth = ComputeWorldPositionDepth(intersectionPoint);
  }
  else
  {
    discard;
  }
}
```

Listing 11.12. Shading or discarding based on ray intersection.

See `OpenGlobe.Scene.`RayCastedTerrainTile for the completed fragment shader. A zoomed-in screen shot with shading by height is shown in Figure 11.20. Why does the terrain appear blocky?

Our ray-stepping approach is using nearest-neighbor interpolation, so the height is considered constant over an entire texel. In the zoomed view, a projected texel covers several pixels. This produces blockiness because the height map is not of a high enough resolution. Nearest-neighbor interpolation is very efficient, though. For many views, LOD can be used to keep texels less than a pixel.

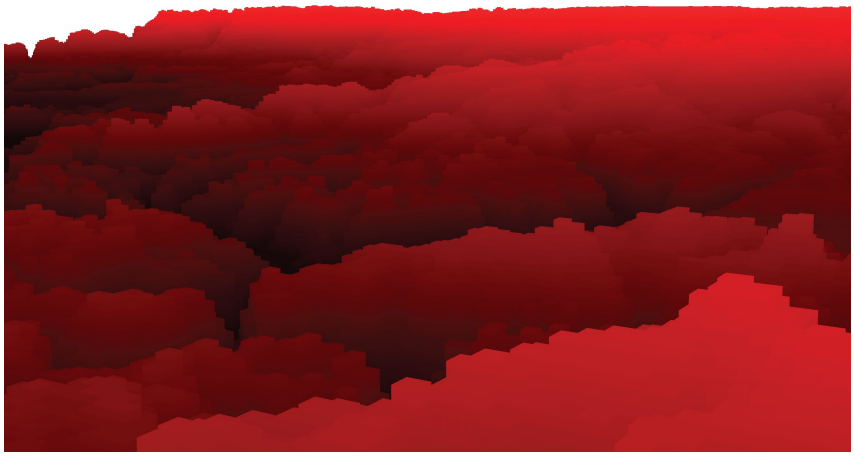


Figure 11.20. Nearest-neighbor interpolation results in blocky terrain if the height map doesn't have sufficient detail for the view.

(a) (b)

Figure 11.21. (a) For downward-facing views, the ray-stepping loop exits quickly. (b) Horizon views require an increasing number of iterations as pixels blend from orange to blue.

Performance. Our brute-force implementation of GPU ray casting has some interesting performance characteristics. Performance is highly dependent on the number of iterations in the ray-stepping loop. More iterations means more texture reads and more instructions executed. The number of iterations is dependent on the view and terrain features. It is instructive to visualize which pixels are the most expensive by shading the terrain based on the number of iterations, as in the following:

```
fragmentColor = mix(vec3(1.0, 0.5, 0.0), vec3(0.0, 0.0, 1.0),
    float(numberOfIterations) / (heightMapResolution.x +
    heightMapResolution.y));
```

With this shading, the color blends between orange, for areas where rays quickly find terrain, and blue, for areas requiring a large number of steps. The two images in Figure 11.21 are rendered with this shading. For nearly top-down views, terrain is quickly found, as shown by the orange in Figure 11.21(a). A more challenging view is looking along the horizon, as in Figure 11.21(b); intersections with peaks near the viewer are found quickly, but reaching terrain in the distance requires a large number of steps, as shown by the pixels blending to blue. A close look at this figure reveals that the top of the distant center peak did not require many steps. For this view, rays intersecting the top of this peak enter the tile's AABB further along the ray than do rays intersecting lower on the peak; therefore, the higher rays step over fewer texels before finding the intersection point. Some of the most expensive pixels aren't even shaded; consider a ray that steps all the way across the height map only to realize that no intersection occurred.

Fortunately, the ray-stepping loop can be optimized quite a bit. It can be reduced to nearly logarithmic time by using a hierarchical data

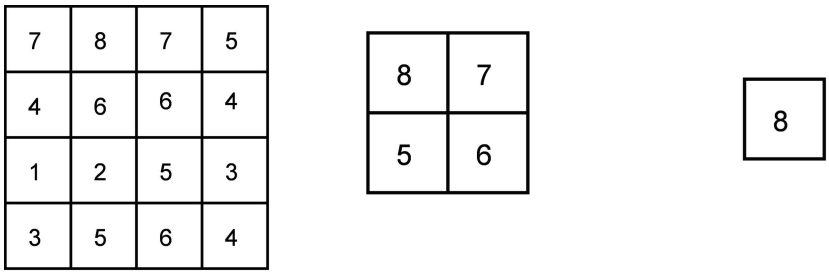


Figure 11.22. A maximum mipmap pyramid. (a) Level zero. (b) Level one. (c) Level two.

structure called a maximum mipmap pyramid [166]. This is a fully subdivided quadtree where the original height map is the finest level, level zero, and texels in coarser levels are the maximum height of the 2×2 corresponding heights in the next-finer level, as shown in Figure 11.22. This data structure can be computed very quickly on the GPU and requires only an additional one-third of the memory of the original height map.

A maximum mipmap pyramid can be used to implement hierarchical ray stepping. Instead of stepping texel by texel along the original height map, the pyramid is used to accelerate stepping by stepping over large parts of the height map in one loop iteration. Since the ray always intersects the top 1×1 level, ray stepping starts at the next-highest level (2×2). If the ray does not intersect any texels at this level, it will not intersect any texels in the original height map, so the fragment is quickly discarded. If the ray intersects a texel, the ray is stepped forward to the intersection point and the pyramid is descended to the next-finer level. Ray stepping continues at this intersection point. If the ray misses the texel, the ray is stepped to the texel's exit point. If the ray also left a 2×2 region in this pyramid level, the pyramid is ascended to the next-coarsest level. Recursion continues until the end of the height map is reached or an intersection is found in the finest level.

An alternative ray-stepping acceleration technique is cone step mapping [42]. In a preprocessing step, a cone is computed for each texel. The cone's apex is centered on top of the texel, with the cone opening upward. The cone's angle is computed to be as large as possible without intersecting other texels. This gives the guarantee that is used to accelerate ray stepping: if a ray intersects a cone, it can be stepped all the way across the cone without intersecting any texels. The downside to cone stepping is the expensive preprocessing step and memory requirements.

(a) (b)

Figure 11.23. High-resolution terrain and imagery of Vorarlberg, Austria, rendered using GPU ray casting. (Images courtesy of Christian Dick, Computer Graphics and Visualization Group, Technische Universität München, Germany. Geo data provided by the Landesvermessungsamt Feldkirch, Austria.)

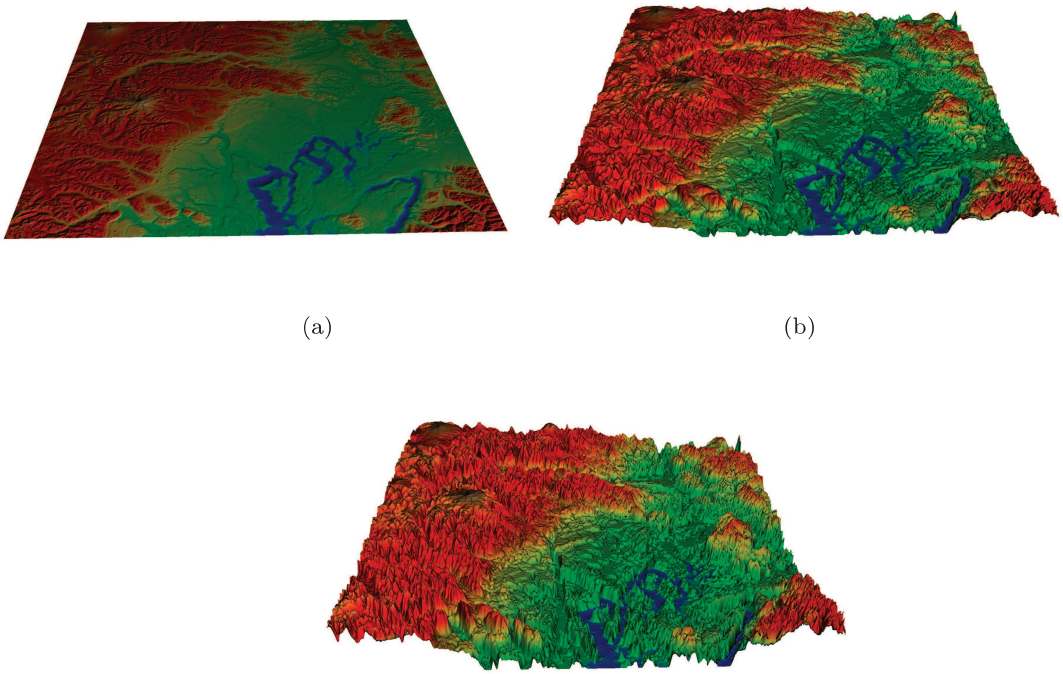
Try This

Run Chapter11TerrainRayCasting, enable shading by number of ray steps, and experiment with different views. Implement a maximum mipmap pyramid or cone step mapping and visually observe the drastic reduction in ray steps.

Implementing optimized GPU ray casting of height maps involves a fair amount of work, even when we are only considering a single tile. Is it worth it? Does the reduction in memory bandwidth and vertex processing outweigh the complex fragment shading? Dick et al. compared GPU ray casting using hierarchical ray stepping with a maximum mipmap pyramid and other occlusion optimizations to their optimized rasterization-based terrain engine [37]. Their results showed that for very high-resolution datasets (see Figure 11.23), ray casting both had a higher frame rate and used less memory than rasterization did; using hierarchical ray stepping resulted in a speedup of five over brute-force ray stepping.

11.2.4 Height Exaggeration

Exaggerating terrain heights is a visual aid that makes subtle height differences more noticeable or flattens areas with significant height differences. Figure 11.24 shows how height exaggeration highlights height differences in a nearly flat terrain. Height exaggeration is a common feature in virtual globe applications and goes by many names; it is called elevation exaggeration in Google Earth and vertical exaggeration in ArcGIS Explorer.



(a) (b)

(c)

Figure 11.24. (a) A terrain tile spanning 512 units in the xy-plane with height in the range $[0, 1]$ appears flat. (b) Applying a height exaggeration of 30 makes height differences more noticeable. (c) A height exaggeration of 60 overemphasizes height differences.

Implementing height exaggeration is straightforward. The height of a height-map pixel is scaled by the exaggeration value. Values greater than one make height differences more noticeable and values less than one flatten the terrain. For the case of terrain extruded from the xy-plane, each position's z-component is multiplied by the exaggeration value. If normals are procedurally generated, they should be computed using the exaggerated heights.

If the exaggeration is static and will not change, as is the case for terrain formats that include a scale factor, height exaggeration can occur when a tile's triangle mesh is created, or its height map is created in the case of displacement mapping or ray casting. This maps the units for height in the terrain format to your model-space units for height.

```
in vec3 position;

uniform mat4 og_modelViewPerspectiveMatrix;
uniform float u_heightExaggeration;

void main()
{
  vec4 exaggeratedPosition =
      vec4(position.xy, position.z * u_heightExaggeration, 1.0);
  gl_Position =
      og_modelViewPerspectiveMatrix * exaggeratedPosition;
}
```

Listing 11.13. Terrain height exaggeration in a vertex shader.

If the exaggeration is dynamic, perhaps because it can be adjusted by the user via a slider, height exaggeration can occur in a vertex shader, as done in Listing 11.13.

Instead of explicitly multiplying the z-component, height exaggeration can be built into the model matrix by multiplying it by a z-scaling matrix.

Static and dynamic height exaggeration are not mutually exclusive; a terrain format may require scaling the stored heights to determine the true heights, then the user may wish to exaggerate the true heights at runtime.

11.3 Computing Normals

Given the various methods for rendering height maps we've seen thus far, we've solved half of the terrain-rendering battle: finding the visible surface. The other half is shading. The first step we'll take in shading is computing terrain normals. Normals have an obvious use in lighting equations, like the ones presented in Section 4.2.1, and less obvious uses such as shading terrain based on steepness, which we will see in Section 11.4.3.

Traditionally, per-vertex normals for arbitrary triangle meshes are computed by using the cross product to find normals for each triangle sharing a vertex, then averaging these normals. The computed normals are stored with other per-vertex data, such as position. The same algorithm will work for terrain normals. Since we are focusing on terrain derived from height maps, we will also focus on deriving normals from height maps. When normals are stored in a 2D image, similar to a height map, the image is called a *normal map*.

Depending on an application's requirements, terrain normals may be computed at different times:

- *Preprocess.* If an application doesn't have access to programmable GPUs or can't afford to compute normals in a shader, a normal map

can be computed as a preprocess step. This may happen offline (e.g., in a game's content-creation pipeline). Major hardware vendors provide normal-map generation tools. NVIDIA has an Adobe Photoshop plug-in to generate normal maps from height maps,[1] and a standalone tool, Melody, to generate normal maps for arbitrary simplified models. AMD provides a standalone tool, Normal Mapper, with C++ source, to generate normals maps from height maps.[2]

Normal maps can also be computed online, after the height map is loaded from disk. This is attractive because it saves on disk space and I/O bandwidth. Rendering, disk I/O, and normal computation can occur on different threads to keep the CPU cores saturated (see Section 12.3). To save memory, a normal map can store two of the three components and compute the third at runtime (e.g., store x and y and compute z), taking advantage of the fact that the normal is of unit length.

- *Procedurally in a shader.* When a shader has access to the height map, which is the case when doing GPU displacement mapping or ray casting, a normal map does not need to be stored. Instead, normals can be procedurally generated based on neighboring heights. Procedural generation saves memory, reduces memory bandwidth, and generally results in less code. In the case of terrain, procedurally generated normals have the added benefit of easily allowing destructible terrain. For example, if a bomb destroys part of the terrain in a game, only the height-map texture needs to be modified; the adjusted normals will be automatically computed.

- *Hybrid.* A hybrid approach can be used to support destructible terrain with normal maps. When the height map is first loaded and every time it is updated, a normal-map texture is generated by a fragment shader that uses the height map to write to an offscreen buffer representing the normal map.

An advantage of using a normal map is that a high-resolution normal map can be used to shade a lower-resolution height map. This improves visual quality without increasing the number of triangles. When a normal map is used in this way, it is called *bump mapping*. The simplicity and memory savings of procedurally generated normals, however, have made it an attractive technique used by many 3D engines.

[1] http://developer.nvidia.com/object/photoshop_dds_plugins.html
[2] http://developer.amd.com/gpu/normalmapper/

11.3.1 Forward Difference

Regardless of whether a normal map or procedurally generated normals are used, the same algorithms for computing normals can be applied. They all work by approximating a position's normal using neighboring heights. In the simplest case, the *forward difference* can be used. This is a fancy term for $f(x+h)-f(x)$. Here, $f(x)$ is the function for the height map, and x is the position of the vertex in the xy-plane. If we let $h=(1,0)$, the forward difference is an approximation of the partial derivative in the x direction. Likewise, if $h=(0,1)$, it is approximate to the partial derivative in the y direction. As shown in Figure 11.25, taking the cross product of these yields an approximate unnormalized normal. Example code is shown in Listing 11.14.

Given the assumption that terrain is extruded from the xy-plane, Listing 11.14 can be simplified further. Since the forward difference uses samples to the right and above, it also has problems along the top and rightmost rows of the height map because adjacent heights are not available. The *backward difference*, $f(x)-f(x-h)$, can be used in these cases. However, the LOD algorithm usually impacts the solution to this, so we will

```
vec3 ComputeNormalForwardDifference(vec3 position,
                                    sampler2DRect heightMap)
{
  vec3 right = vec3(position.xy + vec2(1.0, 0.0),
      texture(heightMap, position.xy + vec2(1.0, 0.0)).r);
  vec3 top = vec3(position.xy + vec2(0.0, 1.0),
      texture(heightMap, position.xy + vec2(0.0, 1.0)).r);
  return cross(right - position, top - position);
}
```

Listing 11.14. Computing normals for terrain using the forward difference.

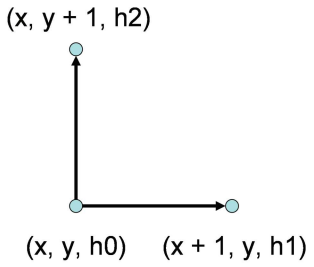
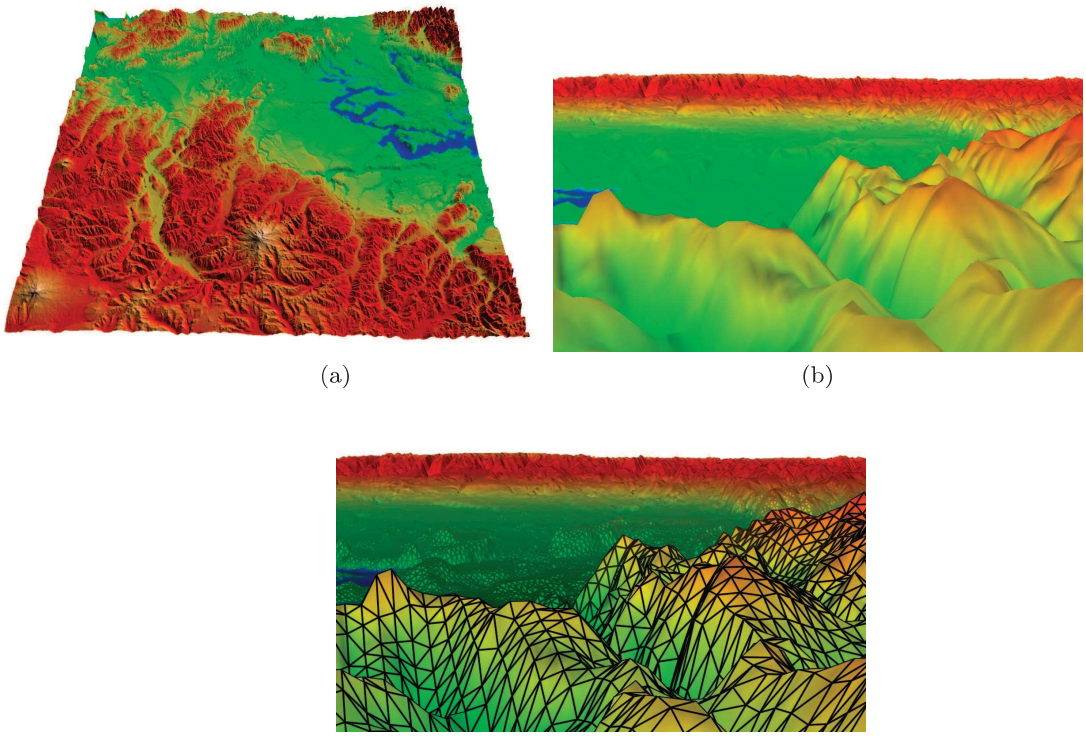


Figure 11.25. The forward difference. The approximate normal for a position can be computed by taking the cross product of two vectors created from three height-map samples.



(a) (b) (c)

Figure 11.26. Terrain shaded with a color map and no lighting. In (c), the wireframe overlay shows the location of vertices.

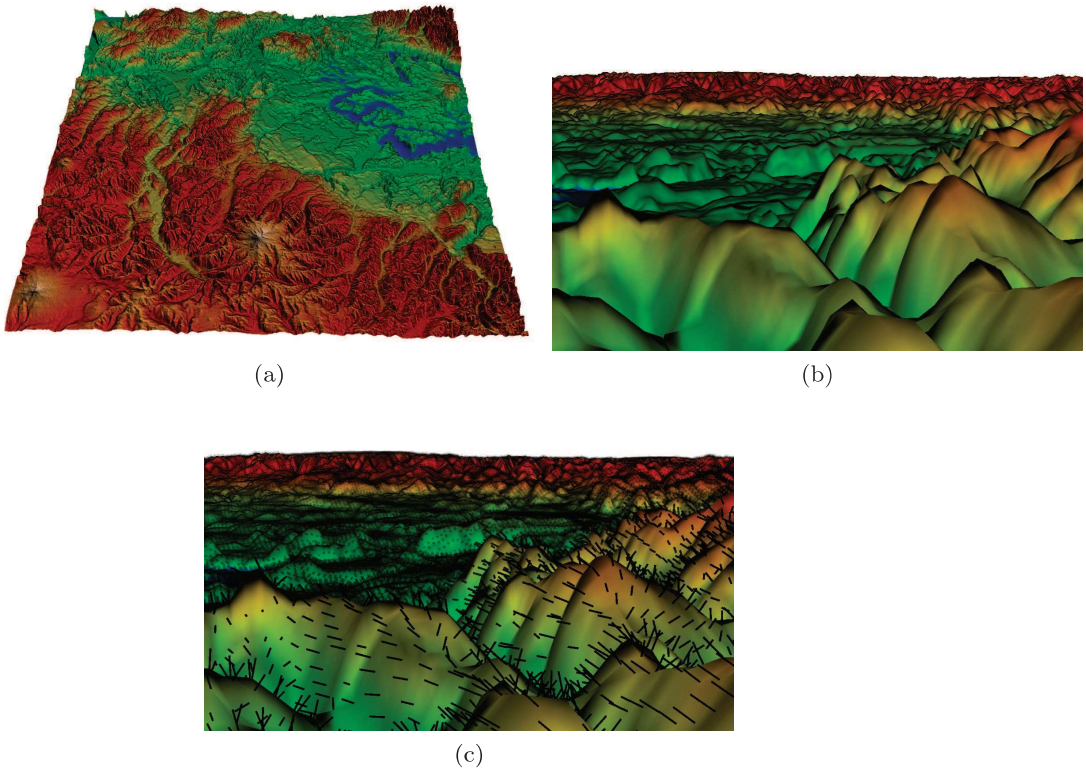
defer the full discussion until later chapters. In this chapter, we will not concern ourselves with artifacts along the borders of a terrain tile.

Figure 11.27 shows shaded terrain with normals computed using the forward difference. For comparison, Figure 11.26 shows the same scene without lighting.

Try This ○○◯◯

The best way to see the visual differences between the different ways of computing normals is to run Chapter11TerrainShading and rotate through each algorithm. There are options to overlay the terrain's wireframe and normals that will help you notice differences.

Using forward or backward difference has the benefit of being very efficient; it requires only three height-map samples. If this is done in a



(a) (b) (c)

Figure 11.27. Terrain shaded with normals computed using the forward difference. The positional light is attached to the camera. This technique is fast, requiring the fewest texture reads and least amount of computation, but produces less accurate results.

displacement-mapping vertex shader, one of the samples is required anyway, so only two extra texture reads and a small amount of computation is required to derive a normal. The downside is that the normal isn't very accurate since it only depends on two additional heights. This is tolerable for low frequency terrains but can be very inaccurate for high-frequency terrains with steep features.

11.3.2 Central Difference

The *central difference*, $f(x + \frac{1}{2}h) - f(x - \frac{1}{2}h)$, can also be used to compute normals. Using the four adjacent heights to a position to compute its normal provides a nice balance between performance and accuracy. This is a good choice for many applications that procedurally generate normals in a shader. The principle is the same as using the forward difference:

```
vec3 ComputeNormalCentralDifference(vec3 position,
                                    sampler2DRect heightMap)
{
  vec3 left = vec3(position - vec2(1.0, 0.0),
      texture(heightMap, position - vec2(1.0, 0.0)).r);
  vec3 right = vec3(position + vec2(1.0, 0.0),
      texture(heightMap, position + vec2(1.0, 0.0)).r);
  vec3 bottom = vec3(position - vec2(0.0, 1.0),
      texture(heightMap, position - vec2(0.0, 1.0)).r);
  vec3 top = vec3(position + vec2(0.0, 1.0),
      texture(heightMap, position + vec2(0.0, 1.0)).r);
  return cross(right - left, top - bottom);
}
```

Listing 11.15. Computing terrain normals using the central difference.

```
vec3 ComputeNormalCentralDifference(vec3 position,
                                    sampler2DRect heightMap)
{
  float leftHeight =
      texture(heightMap, position.xy - vec2(1.0, 0.0)).r;
  float rightHeight =
      texture(heightMap, position.xy + vec2(1.0, 0.0)).r;
  float bottomHeight =
      texture(heightMap, position.xy - vec2(0.0, 1.0)).r;
  float topHeight =
      texture(heightMap, position.xy + vec2(0.0, 1.0)).r;
  return vec3(leftHeight - rightHeight,
              bottomHeight - topHeight, 2.0);
}
```

Listing 11.16. An optimized version of computing terrain normals using the central difference.

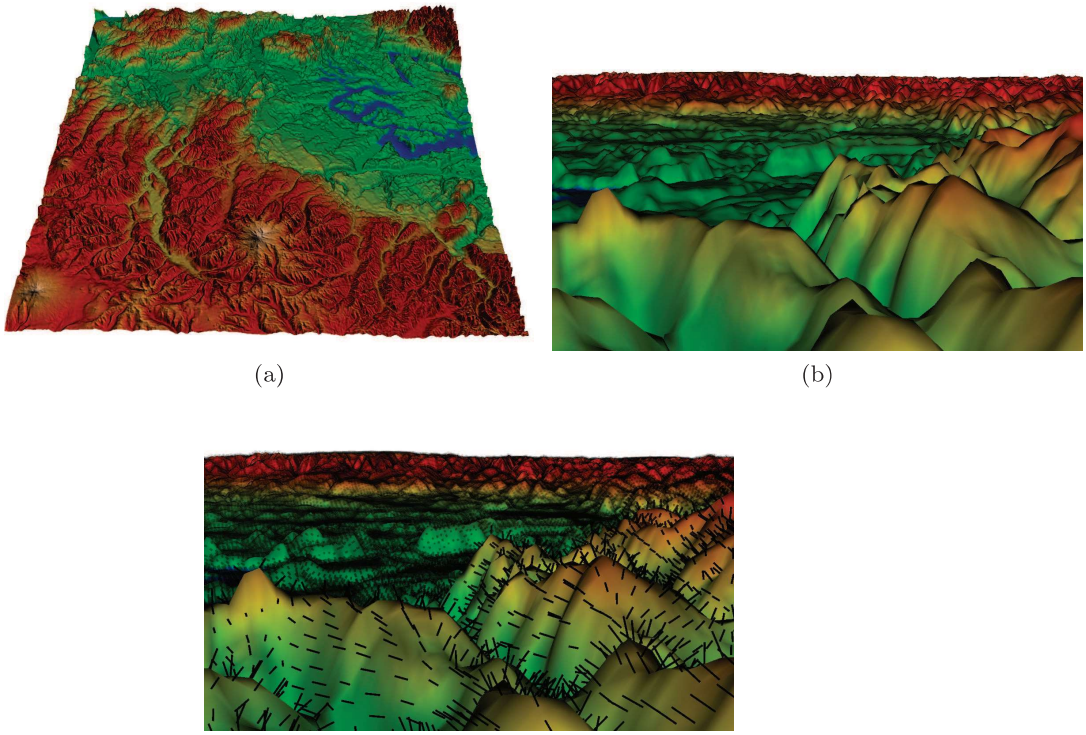
take the cross product of two vectors representing partial derivatives. The vector in the x direction is based on the left and right heights, and the vector in the y direction is based on the top and bottom heights. Using four samples in this way is sometimes called a *star* or *cross filter*.

A GLSL function for computing unnormalized normals using the central difference is shown in Listing 11.15. Compare this to the optimized version of the same function in Listing 11.16 [154]. The optimized version assumes terrain is extruded from the xy-plane and posts are one unit apart. Figure 11.28 shows the visual results.

11.3.3 Sobel Filter

A popular technique for deriving normals from a height map is to use a *Sobel filter* [156]. This is used by AMD's NormalMapper tool; AMD's RenderMonkey provides GLSL and HLSL code examples.[3] The Sobel filter

[3] http://developer.amd.com/gpu/rendermonkey/



(a) (b)

(c)

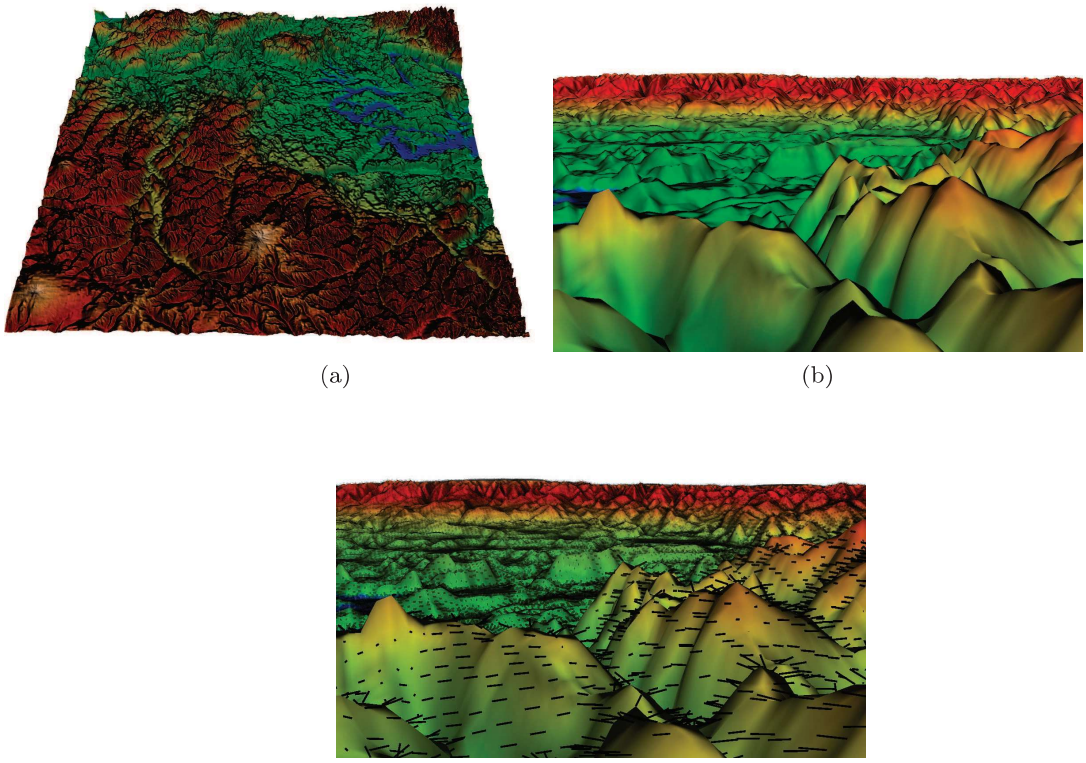
Figure 11.28. Terrain shaded with normals computed from four samples provides a reasonable trade-off between performance and accuracy.

is an edge-detection filter used in image processing with separate kernels to detect horizontal and vertical edges. The horizontal kernel is shown in Equation (11.1) and the vertical kernel is shown in Equation (11.2):

$$\begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}, \tag{11.1}$$

$$\begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}. \tag{11.2}$$

In image processing, a horizontal or vertical edge is detected by "centering" the corresponding kernel over a pixel and summing the product of the kernel with the pixel and its eight neighbors component-wise. If the sum



(a) (b)

(c)

Figure 11.29. Terrain shaded with normals computed using a Sobel filter.

```
vec3 ComputeNormalSobelFilter(vec3 position,
                              sampler2DRect heightMap)
{
  float upperLeft =
      texture(heightMap, position.xy + vec2(-1.0, 1.0)).r;
  float upperCenter =
      texture(heightMap, position.xy + vec2(0.0, 1.0)).r;
  float upperRight =
      texture(heightMap, position.xy + vec2(1.0, 1.0)).r;
  float left =
      texture(heightMap, position.xy + vec2(-1.0, 0.0)).r;
  float right =
      texture(heightMap, position.xy + vec2(1.0, 0.0)).r;
  float lowerLeft =
      texture(heightMap, position.xy + vec2(-1.0, -1.0)).r;
  float lowerCenter =
      texture(heightMap, position.xy + vec2(0.0, -1.0)).r;
  float lowerRight =
      texture(heightMap, position.xy + vec2(1.0, -1.0)).r;

  float x = upperRight + (2.0 * right) + lowerRight -
            upperLeft - (2.0 * left) - lowerLeft;
  float y = lowerLeft + (2.0 * lowerCenter) + lowerRight -
```

```
            upperLeft - (2.0 * upperCenter) - upperRight;

  return vec3(-x, y, 1.0);
}
```

Listing 11.17. Computing terrain normals from a height map using a Sobel filter.

is above a threshold, an edge is detected. Since the kernels approximate the partial derivatives in x and y of the image's intensity, a Sobel filter can also be used to compute terrain normals, as shown in Listing 11.17. Like all functions shown thus far, the returned normal is not normalized. The constant 1.0 is used for the normal's z-component; this can be adjusted to smooth out or sharpen the shading.

Figure 11.29 shows terrain shaded with normals computed with a Sobel filter. The edge-detection nature of the Sobel filter is apparent in the zoomed-out view. Increasing the normal's z-component would smooth out the shading and make edges less apparent.

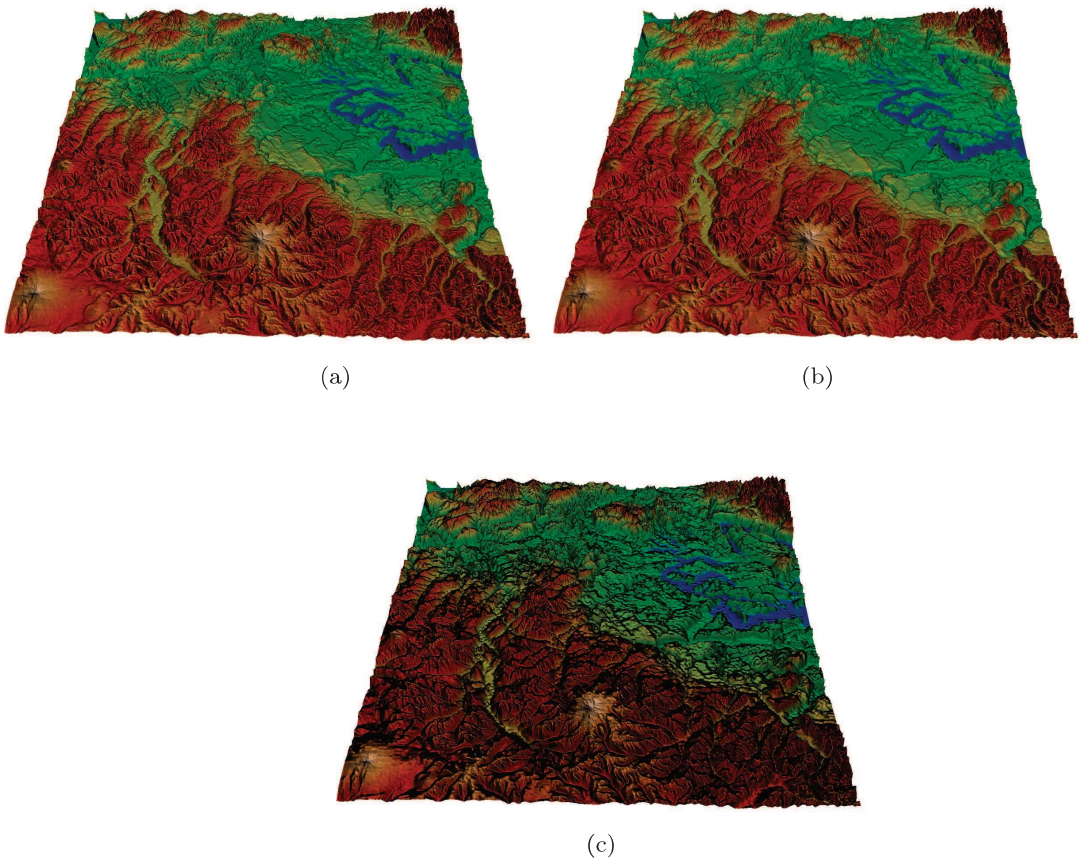
11.3.4 Summary of Normal Computations

For many applications, normals are used primarily for lighting. In this case, the accuracy of the normal is not that important as long as the result looks right. In other cases, the normal may be used for analytic visualization, such as shading based on steepness or compass direction. In these cases, the accuracy of the normal may be more important than its speed of computation. When a normal map is computed offline, the speed of computation is typically even less important.

Figures 11.30, 11.31, and 11.32 show side-by-side comparisons of the three normal computations described in this chapter. In all cases a single positional light is located at the camera.

11.4 Shading

A wide array of creative terrain-shading algorithms are used in virtual globes, GIS applications, and games. Virtual globes tend to focus on shading with high-resolution satellite imagery. GIS applications commonly shade terrain to highlight its features, such as valleys or steep areas. Perhaps games have the ultimate challenge as their terrain shading needs to portray a realistic image of an artificial, potentially destructible landscape and enable artists to easily author such scenes.



(a) (b)

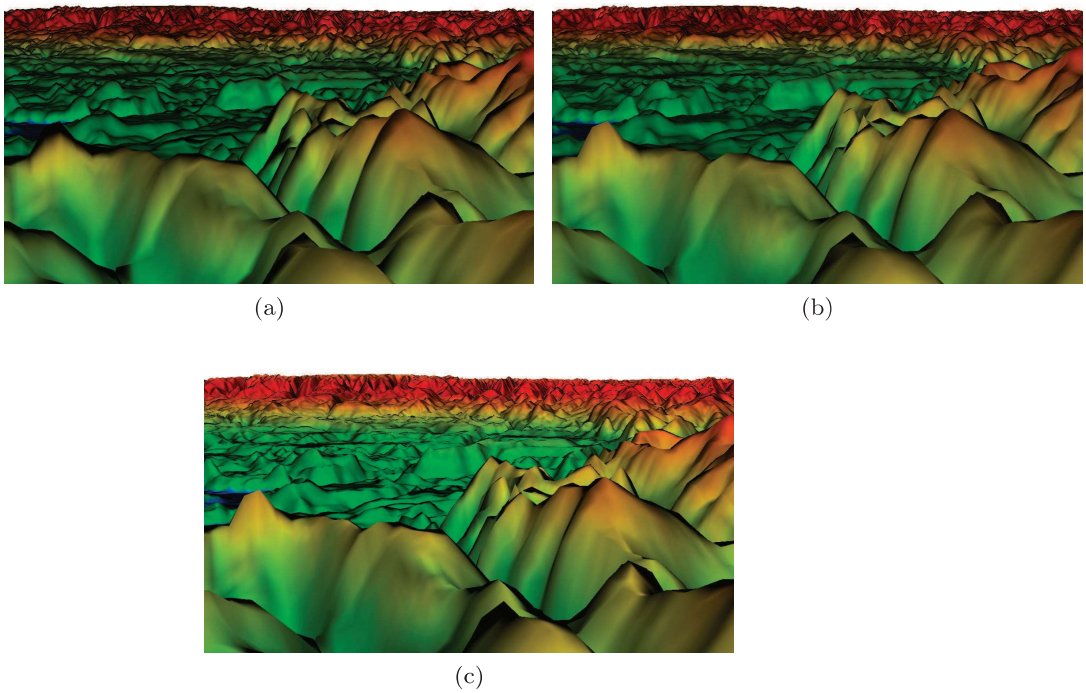
(c)

Figure 11.30. In a zoomed-out view, the (a) forward difference and (b) central difference are similar. The central difference produces slightly smoother shading. The (c) Sobel filter results in the sharpest shading, highlighting features that are less apparent in (a) and (b).

11.4.1 Color Maps and Texture Coordinates

A common way to shade terrain is to use a color map like the one shown earlier in Figure 11.7(b). A color map is a texture containing colors used for shading. It is exactly what we think of when we think of a texture map. Since we may store other things in textures when rendering terrain (e.g., normals for lighting or alpha values for blending multiple textures), we explicitly state we are storing colors by calling the texture a color map. A color map could be real satellite imagery or artist-created grass, dirt, stone, etc.

When a single color map is draped over a terrain tile, the fragment shader can be as simple as a single texture read, usually modulated with



(a) (b) (c)

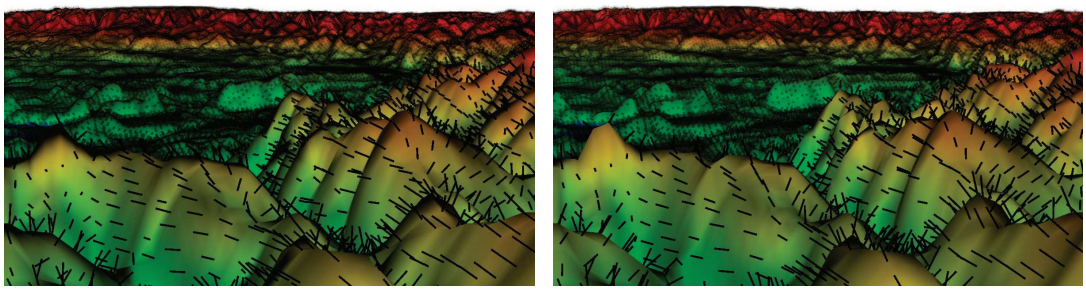
Figure 11.31. In a zoomed-in view, the (a) forward difference and (b) central difference are again pretty similar. The differences are most noticeable on the left side in the close peaks. In this view, the (c) Sobel filter produces smoother shading (see Figure 11.32).

lighting intensity. If a color map is only modulated with the diffuse-lighting component, it is called a *diffuse map*. Likewise, it is called a *specular map* if it is only modulated with the specular component.

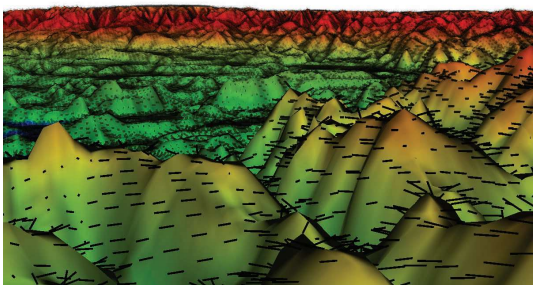
Texture coordinates used to read a color map can be procedurally generated in the vertex shader. A texture coordinate can be computed from the vertex's xy-world-space coordinates using a simple linear mapping, as done in Listing 11.18.

When `u_textureCoordinateScale` is set to $\left(\frac{1}{x \text{ tile resolution}}, \frac{1}{y \text{ tile resolution}}\right)$, the texture coordinates span from $(0, 0)$ in the bottom left corner of the tile to $(1, 1)$ in the upper-right corner.

When color maps have repeating patterns (e.g., grass or dirt), it is useful to use repeat or mirrored-repeat texture filtering and set `u_texture CoordinateScale` to $\left(\frac{n}{x \text{ tile resolution}}, \frac{m}{y \text{ tile resolution}}\right)$ to repeat the color map



(a) (b)



(c)

Figure 11.32. The same zoomed-in view as in Figure 11.31 with normals drawn at vertices. The (c) Sobel filter's normals do not point in the z direction as much as the other methods and, therefore, produce smoother, lighter shading because of the light source's position at the camera.

```
in vec2 position;
out vec2 textureCoordinate;

uniform vec2 u_textureCoordinateScale;
uniform vec3 u_aabbLowerLeft;

void main()
{
  // ...
  textureCoordinate =
      (position - u_aabbLowerLeft.xy) * u_textureCoordinateScale;
}
```

Listing 11.18. Generating texture coordinates in a vertex shader.

along the tile n times in the x direction and m times in the y direction. This can improve visual quality by creating a more even texel to pixel ratio without increasing the size of the actual color map. To make the repetition less noticeable, the repeat can be made view dependent by blending between two sets of texture coordinates based on the distance to the camera [145].

It is also common to generate more than one set of texture coordinates to look up different textures. For example, the texture coordinate used to read a base color map may span from $(0, 0)$ to $(1, 1)$ while a color map with fine detail may be repeated across a tile using texture coordinates spanning from $(0, 0)$ to $(4, 4)$. In fact, this technique is common enough that the fine-detail color map is called a *detail map*.

11.4.2 Detail Maps

Due to limited texture resolution, terrain near the viewer can appear blurry. For example, in virtual globes, sparsely populated areas tend to have lower-resolution satellite imagery than densely populated areas. When the user is zoomed in close to the low-resolution imagery, one texel maps to many pixels, making the terrain look blurry. For applications that can tolerate synthesizing detail, a detail map can be used to reduce blurring. A detail map is a texture containing high-frequency details, including dips and bumps, commonly authored using a noise function. To avoid blurring near the viewer, a color map is blended with a tiled detail map to increase the appearance of detail.

11.4.3 Procedural Shading

Procedural-shading techniques compute a fragment's color at runtime using properties of the terrain and typically very little memory. Procedural techniques create visually rich shading with little memory or shading that highlights terrain features such as steepness. Either way, we are amazed at how so little code can generate such visually pleasing and useful shading!

The figures in this section provide a good sense of procedural shading, but to really get acquainted, run Chapter11TerrainShading. In addition to rotating through the algorithms, enable and disable lighting to get a sense of how each shading technique highlights the terrain's features.

Try This

Height-based shading. Many terrain-shading algorithms are based on the terrain's height. A vertex shader typically passes the height, which is interpolated during rasterization, to the fragment shader. Perhaps the simplest approach is to map the minimum height to one color and the maximum height to another color and linearly interpolate between the two based on the fragment's height. The fragment shader in Listing 11.19 does this by assigning black to the minimum height and green to the maximum height.

The result is shown in Figure 11.33(c). Compared to no shading at all (see Figure 11.33(a)), shading by height provides some sense of terrain features. Compared to just lighting (see Figure 11.33(b)), shading by height provides a better sense of height for a particular location; it is easy to determine that dark regions are low and bright regions are high—such observations are not as easy to make with just lighting.

Figure 11.33(d) shows lighting combined with shading by height, achieving the best of both worlds. These are combined by multiplying the height-based color with the light intensity. Combining the two also avoids a weak-

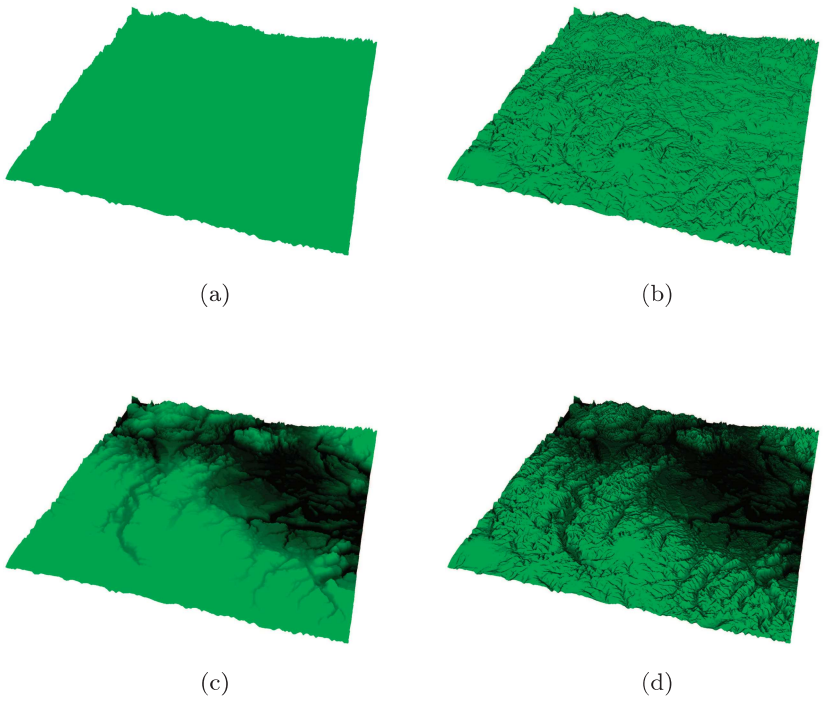
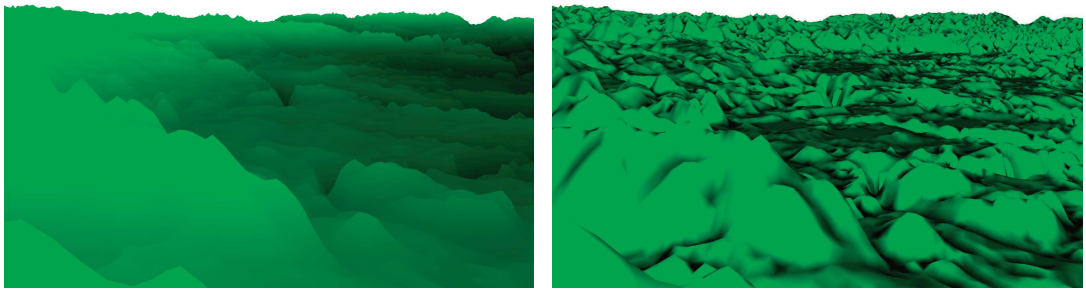


Figure 11.33. (a) Without shading, terrain features are unrecognizable. (b) With lighting, terrain features become apparent. (c) Shading by height provides a sense of terrain features even without lighting. (d) Combining lighting and shading by height conveys both terrain features and height.



(a) (b)

Figure 11.34. (a) For horizon views, shading by height alone does not give a good sense of terrain features. (b) Lighting brings out terrain features for all views, especially horizon views.

ness of shading by height alone—horizon views. As shown in Figure 11.34, just shading by height makes it hard to distinguish peaks in the background and foreground that are at similar heights.

Allowing the user to determine the approximate terrain height at a quick glance is useful for scientific visualization and GIS applications. Shading by height is one way to achieve this. Another approach, which can be combined with shading by height, is to render contour lines of constant height, as shown in Figure 11.35.

Height contours can be procedurally generated in a fragment shader using the same technique used to render a latitude-longitude grid on a globe, as described in Section 4.2.4. Listing 11.20 shows a fragment-shader snippet that shades contours red and terrain green.

The interval between contours is used to determine the distance from the fragment to the closest contour. If the distance is small enough, the

```
in float height;
out vec3 fragmentColor;

uniform float u_minimumHeight;
uniform float u_maximumHeight;

void main()
{
  fragmentColor = vec3((height - u_minimumHeight) /
                       (u_maximumHeight - u_minimumHeight),
                       0.0, 0.0);
}
```

Listing 11.19. Shading by height in a fragment shader.

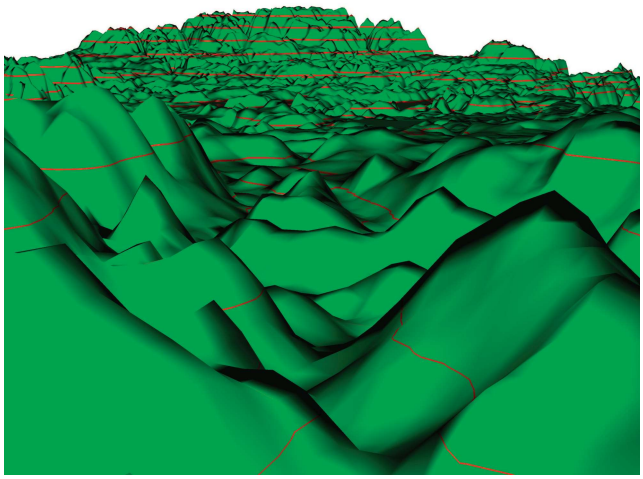


Figure 11.35. Contour lines procedurally generated in a fragment shader allow the user to quickly gauge approximate terrain height and curvature.

fragment is shaded appropriately. Screen-space partial derivative functions, dFdx and dFdy, are used to maintain a constant pixel width.

Try This ○○◯◯

Similar to latitude-longitude grids, a variety of shading options exist for height contours. Make contours a different color or width based on height.

In addition to providing an approximate sense of height, the curvature of height contours can highlight terrain features, as shown in Figure 11.36.

Thus far, our shading techniques have used just height to determine a fragment's color. A wide array of creative techniques become possible when additional data in the form of textures are also used. One such technique

```
float distanceToContour = mod(height, u_contourInterval);
float dx = abs(dFdx(height));
float dy = abs(dFdy(height));
float dF = max(dx, dy) * u_lineWidth;
fragmentColor = mix(vec3(0.0, intensity, 0.0),
                    vec3(intensity, 0.0, 0.0),
                    (distanceToContour < dF));
```

Listing 11.20. Creating height contours.

(a) (b)

Figure 11.36. (a) As seen by the curvature in the foreground, height contours provide some sense of terrain features even without shading. (b) Without shading, terrain features are unrecognizable.

uses a fragment's height to look into a 1D texture, or a one-texel-wide 2D texture, called a *color ramp*, to determine the fragment's color. A color ramp, such as the one in Figure 11.37, can easily be authored in a paint program. Figure 11.38 shows terrain shaded with this color ramp, which ramps from water to grass to dirt to snow.

Using a color-ramp texture is more general and easier to implement than trying to blend between several colors in a fragment shader. It requires only a single line of code, as demonstrated in Listing 11.21. It is very efficient

Figure 11.37. Color ramp used for height-based shading.

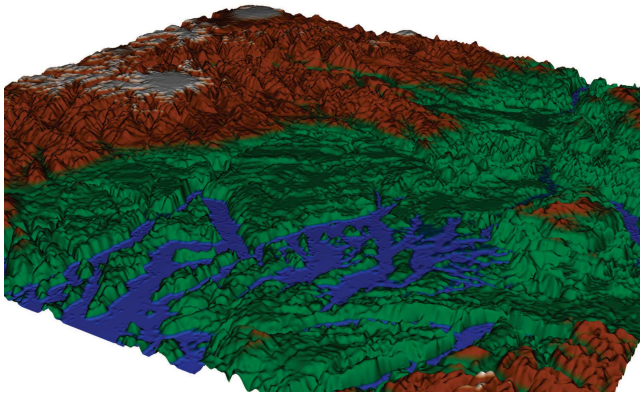


Figure 11.38. Terrain shaded with the color ramp in Figure 11.37.

in terms of speed and memory, costing only a single read from a usually small texture.

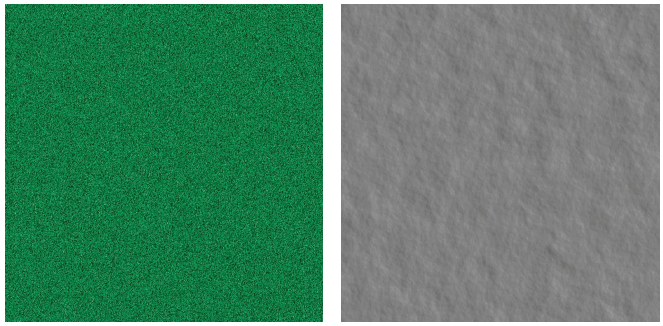
Color ramps are useful in virtual globes when satisfactory satellite imagery is not available or as a memory-efficient alternative to satellite imagery for certain areas. Height-based color ramps do not need to represent materials such as grass and dirt. They can represent height intervals, giving the user another method of approximating height in addition to height contours and height-based shading. This can be useful in aircraft simulation and in water level what-if scenarios (e.g., if the water level rose two meters, what areas would flood?).

Using a color ramp can add significant visual richness with very little performance and memory costs. Its major limitation is that the same color is used for all fragments at a given height. This means that materials like grass and stones have to be described by a single color. In many cases, it is desirable to use color maps, like the ones in Figure 11.39, for materials.

A simple way to do this is to extend the color-ramp approach. Instead of storing colors in the ramp, store an alpha value that is used to blend two color maps. An example blend ramp is shown in Figure 11.40. Linearly interpolating between the two color maps based on the alpha in the blend ramp is usually sufficient (i.e., color $=$ $((1 - \text{alpha}) * \text{grassColorMap}) + (\text{alpha} * \text{stoneColorMap})$).

```
vec2 coord = vec2(0.5, (height - u_minimumHeight) /
                       (u_maximumHeight - u_minimumHeight));
fragmentColor = intensity * texture(u_colorRamp, coord).rgb;
```

Listing 11.21. Using a height-based color ramp for terrain shading.



(a) (b)

Figure 11.39. (a) Grass and (b) stone color maps that are blended together based on the blend ramp in Figure 11.40. When the blend ramp is black, grass is used. When the blend ramp is white, stone is used.

Listing 11.22 shows how this is implemented. A texel is read from both the grass and stone color maps. The texture coordinates for the color maps are not dependent on the blend ramp or each other; therefore, color maps can be repeated inside terrain tiles independent of each other. Next, the fragment's height is used to look up the alpha value in the blend ramp. This alpha value is used to linearly interpolate between the two texels with

```
float normalizedHeight = (height - u_minimumHeight) /
                         (u_maximumHeight - u_minimumHeight);
fragmentColor = intensity * mix(
    texture(u_grass, repeatTextureCoordinate).rgb,
    texture(u_stone, repeatTextureCoordinate).rgb,
    texture(u_blendRamp, vec2(0.5, normalizedHeight)).r);
```

Listing 11.22. Blending two color maps by height.

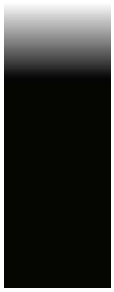


Figure 11.40. Blend ramp used for height-based shading.

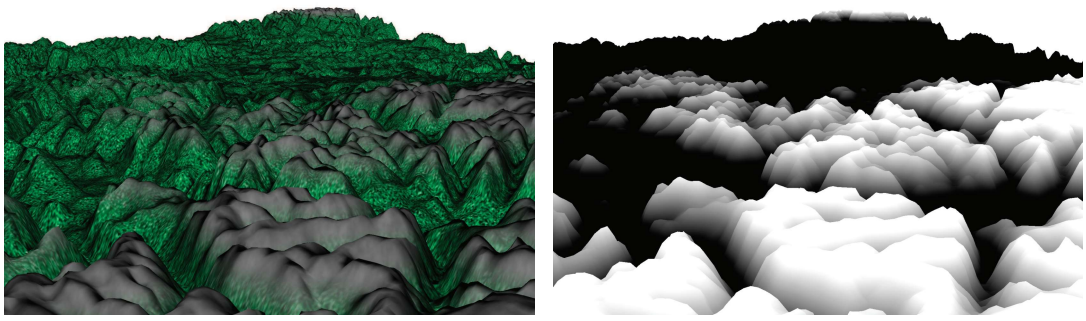
mix. To make the blend between color maps more realistic, a noise function can be used.

Try This ○○◯◯

A blend ramp is likely to have a lot of 0s and 1s, making one of the color map texture reads in Listing 11.22 unnecessary. Try optimizing the shader in Chapter11TerrainShading using dynamic branches to avoid the unnecessary texture read, similar to the night-lights fragment shader in Listing 4.15 in Section 4.2.5. Does it improve performance? Why or why not?

Figure 11.41 shows the result of using a blend ramp to blend between grass and stone based on height. A blend ramp allows flexible nonlinear control of blending, which is evident as most heights are grass and only the highest locations have stone.

More than two color maps can be used with a blend ramp between each. Consider a scene with water, sand, and grass color maps and a blend ramp, *ramp*0, between water and sand, and a blend ramp, *ramp*1, between sand and grass. First the water and sand are blended: color0 = ((1 − ramp0.alpha) ∗ waterColorMap) + (ramp0.alpha ∗ sandColorMap). This color is then blended with grass to compute the final color: color = ((1 − ramp1.alpha) ∗ color0) + (ramp1.alpha ∗ grassColorMap). A more



(a) (b)

Figure 11.41. (a) The result of using the blend ramp in Figure 11.40 to blend grass and stone color maps. (b) The blend ramp is applied to terrain as a color ramp.

memory efficient, but slightly harder to author, approach is to use a single blend ramp, where the integer part of the alpha value determines what color maps to blend between and the fractional part is the alpha value used for blending. In this example, $0.x$ would indicate to blend between water and sand, and $1.x$ would indicate to blend between sand and grass.

In some cases, blending can occur offline, generating a single color map, so only a single texture read in the fragment shader is required at runtime. This approach lacks the flexibility and memory efficiency of independently repeating small color maps across a terrain tile. Since the terrain's height is used to create the offline color map, the color map cannot be repeated across the tile.

Slope-based shading. All the terrain-shading techniques that use height can also use slope. In games, this is useful to make flat areas grassy and steep areas rocky. In GIS applications, slope-based shading allows the user to quickly approximate steepness.

We define a slope of zero to indicate a steep, 90° face and a slope of one to indicate flat land. This allows us to formulate the terrain's slope as the cosine of the angle between the terrain's normal and the ground plane's normal, which is the dot product of the two:

$$\text{slope} = \hat{\mathbf{n}}_{\text{terrain}} \cdot \hat{\mathbf{n}}_{\text{groundplane}}.$$

When the ground plane is the xy-plane, the slope is just the z-component of the terrain's normal. Visualizing slope, as in Figure 11.42, requires a very short fragment shader, `fragmentColor = vec3(normal.z)`, assuming `normal` is normalized.

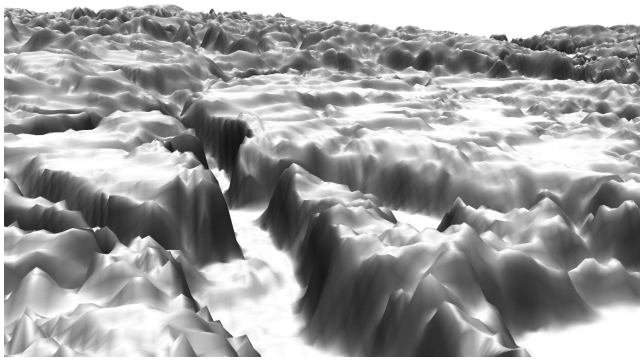


Figure 11.42. Terrain shaded by slope. Flat areas are white, steeper areas fade to black. Slope is computed as the dot product of the terrain's normal and the ground plane's normal.

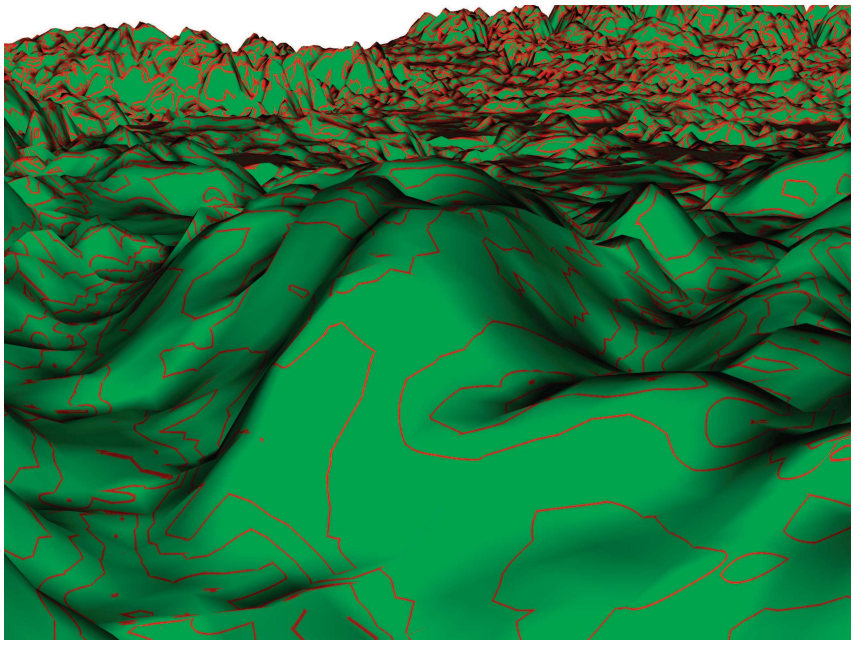


Figure 11.43. Slope contours spaced every 15°.

Contour lines of constant slope are useful to identify areas of similar steepness and areas where steepness changes (see Figure 11.43). These slope contours are rendered in the same manner as height contours. The potential point of confusion is that our definition of slope is not an angle. It is the cosine of the angle between the terrain's normal and the ground plane's normal. Therefore, to render contour lines every x degrees, acos must be used to determine the angle of the slope for the fragment. See the first line in Listing 11.23. Compare this listing to the code for rendering height contours in Listing 11.20.

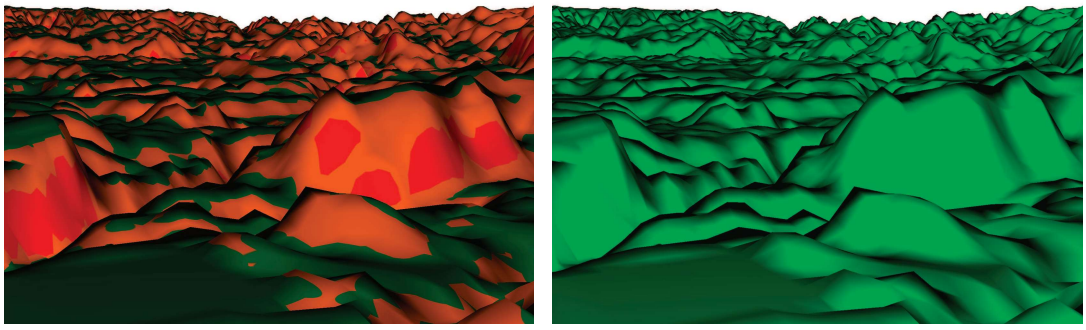
Slope-based color ramps are useful in GIS applications. A user planning new roads may want to easily determine steep areas so they can be avoided or the required switchbacks can be added. Likewise, a military user plan-

```
float slopeAngle = acos(normal.z);
float distanceToContour = mod(slopeAngle, u_contourInterval);
float dx = abs(dFdx(slopeAngle));
float dy = abs(dFdy(slopeAngle));
float dF = max(dx, dy) * u_lineWidth;
fragmentColor = mix(vec3(0.0, intensity, 0.0),
                    vec3(intensity, 0.0, 0.0),
                    (distanceToContour < dF));
```

Listing 11.23. Creating slope contours.

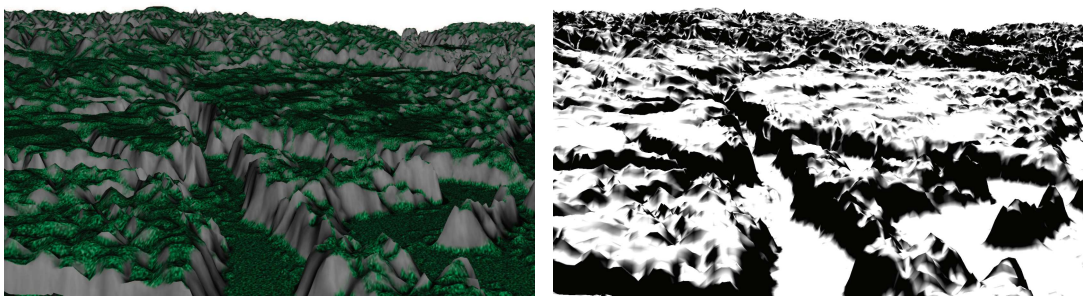
Figure 11.44. Color ramp used for slope-based shading.

ning a mission may want to identify areas that are too steep to walk along. The color ramp in Figure 11.44 was colored to warn of steep areas. A slope of 60° or greater is red and a slope between 30° and 60° is orange. Given the fragment shader has the cos of the slope, the color ramp was authored such that the bottom half is red ($\cos 60° = 0.5$), the next area is orange (0.5 to $\cos 30° \approx 0.866$), and the top section, representing the flattest areas, is green. If the color ramp were authored linearly in degrees, an acos would be required in the shader. Since exact intervals are desired, the color ramp doesn't have smooth transitions.



(a) (b)

Figure 11.45. Slope-based shading using the color ramp from Figure 11.44. (a) Slope-based color ramp. (b) Just lighting. It is much easier to approximate the steepness of the terrain in (a) than in (b).



(a) (b)

Figure 11.46. (a) A fragment's slope is used to look up an alpha value used to blend grass and stone such that flat areas are grassy and steep areas are rocky. (b) The blend ramp is applied to terrain as a color map.

Figure 11.45 shows the result of slope-based color-ramp shading. The slope is used to read the color ramp in the same way the height is used to read a color ramp in Listing 11.21.

Try This ○○○◯

Run Chapter11TerrainShading and select "Color Ramp By Slope." Use the up and down arrow keys to change the height exaggeration, which in turn adjusts the slope. Color-ramp shading makes it easy to see areas of different steepness. Try adjusting height exaggeration without it to see the difference.

Slope-based blend ramps are useful for games and simulations. Just like height can be used to look up an alpha value to blend two color maps, slope can also be used. The code is identical to the height-based approach shown in Listing 11.22 except that slope is used. Perhaps the most common use, shown in Figure 11.46, is to make flat areas grassy and steep areas rocky as they are unlikely to grow vegetation. Slope-based blend ramps create quite a bit of visual complexity using very little memory. Slope-based shading is also useful for other techniques, such as snow accumulation, as is done in the Frostbite engine [4].

Comparison of height- and slope-based shading. Table 11.2 shows the major height-based and slope-based shading techniques. These techniques can be combined. For example, a user may want to see height contours on slope-based color-ramp-shaded terrain. Likewise, height- and slope-based shading can be combined with other shading techniques like detail maps.

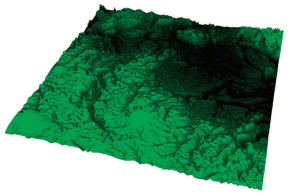
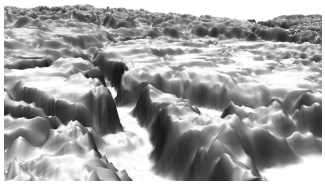
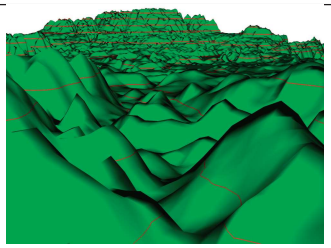
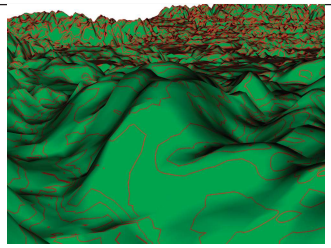
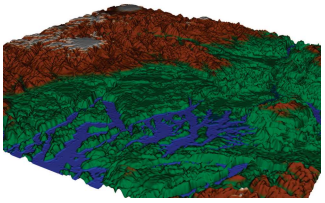
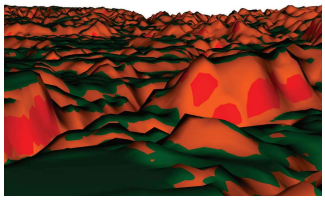
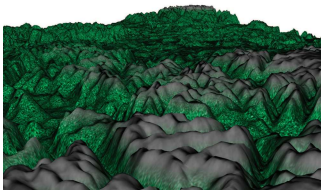
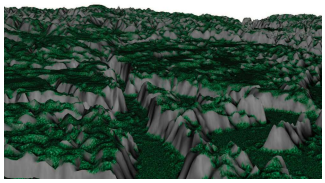
	Height	Slope
Shading by height		
Contour lines		
Color ramp		
Blend ramp		

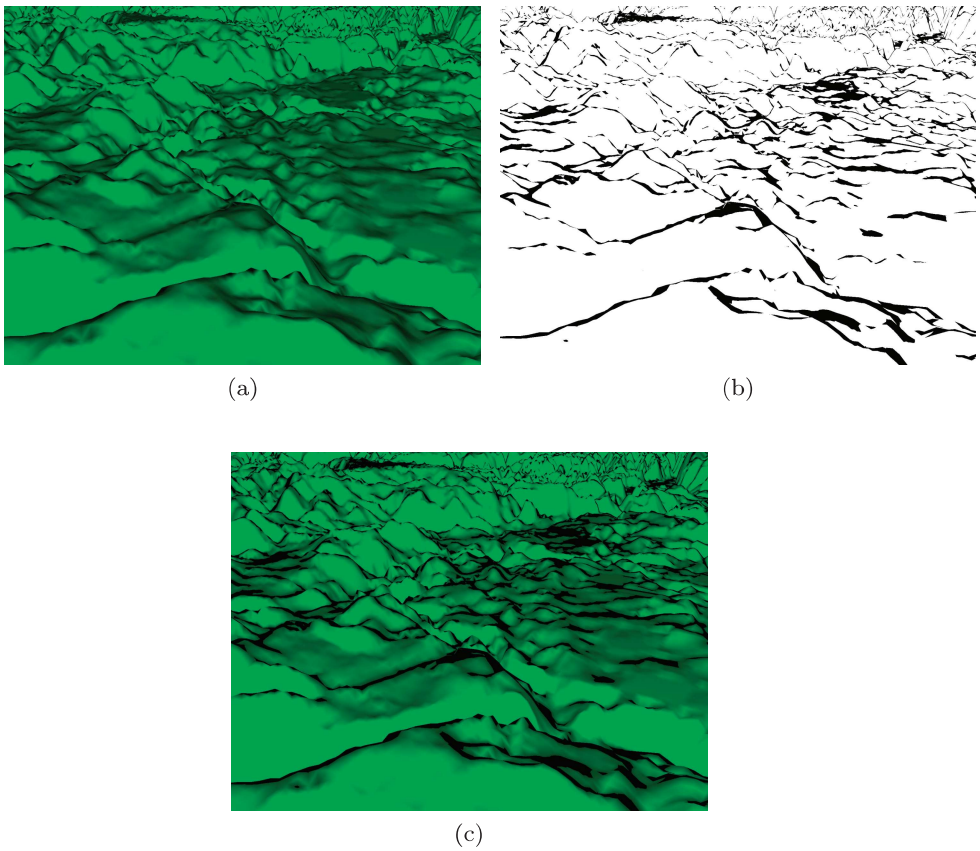
Table 11.2. Comparison of procedural height-based and slope-based shading techniques.

Besides shading based on height and slope, it is also common to shade based on the terrain's normal. In GIS applications, a normal-based color ramp may be used to help the user determine which compass direction terrain faces. In games, normal-based blend ramps may be used to render denser vegetation on terrain faces that capture more sun. Modify Chapter11TerrainShading to use one of these techniques.

Try This

Silhouette-edge rendering. A silhouette edge with respect to the viewer is an edge in which one triangle is front-facing and the other is back-facing. Loosely speaking, it can be thought of as the outline of an object (including interior outlines), as shown in Figures 11.47(a) and 11.47(b). Silhouette-edge rendering is an area of non-photorealistic rendering (NPR). It is commonly combined with toon shading to provide a cartoon-like appearance. It can also be used in terrain rendering to emphasize peaks, slopes, and other features.

A simple technique for rendering silhouette edges is surface angle silhouetting. A silhouette is detected in a fragment shader by checking the dot product of the surface normal and the vector to the viewer. When the dot product is zero, the vectors are perpendicular, and thus an edge is detected. In practice, the dot product is checked to be near zero, as in



(a) (b)

(c)

Figure 11.47. (a) Terrain shaded with a solid color. (b) The same terrain with only its silhouette edges rendered. (c) Shading combined with silhouette edges.

```
if (abs(dot(normal, positionToEye)) < delta)
{
  fragmentColor = vec3(0.0);
  return;
}
```

Listing 11.24. Fragment-shader snippet for surface angle silhouetting.

Listing 11.24, or used to index into a 1D texture, similar to using height or slope to index into a color ramp.

Surface angle silhouetting only requires a single rendering pass, but it fails to capture some silhouettes, while smearing black on flat areas (see areas in the distance in Figure 11.47(c)). This is one of many silhouette-edge-rendering techniques. Other techniques include silhouette-edge detection in a geometry shader, edge detection via image processing, and multipass rendering. A multipass wireframe approach has been evaluated for use with terrain [16]. Akenine-Moller et al. include an excellent survey of NPR, including silhouette-edge rendering [3].

Blend maps. Procedural shading is an elegant approach to shading terrain. Before programmable fragment shaders, techniques like slope-based color ramps required offline computation. Now, a small shader will do the trick. Unfortunately, some terrain-shading techniques cannot utilize procedural shading. A prominent example in virtual globes is rendering high-resolution satellite imagery. The fragment shader is usually not much more than a texture read and lighting, and perhaps some atmospheric effects. The real challenge is in managing the enormous amount of imagery. It is also difficult to utilize procedural shading when complete control over blending is desired. In this case, blend maps can be used.

Height- and slope-based blend ramps use terrain features to select an alpha value for blending. There are times when a user wants complete control over where and how color maps are blended. In the case of a game, an area of terrain may have been impacted by an explosion, or a vehicle coming to a screeching halt may have left brake marks behind. Cases like these call for a blend map as shown in Figure 11.48. With a blend map, an alpha value in a 2D alpha texture is used to blend between two color maps.

```
fragmentColor = intensity * mix(
    texture(u_grass, repeatTextureCoordinate).rgb,
    texture(u_stone, repeatTextureCoordinate).rgb,
    texture(u_blendMask, textureCoordinate).r);
```

Listing 11.25. Using a blend map.

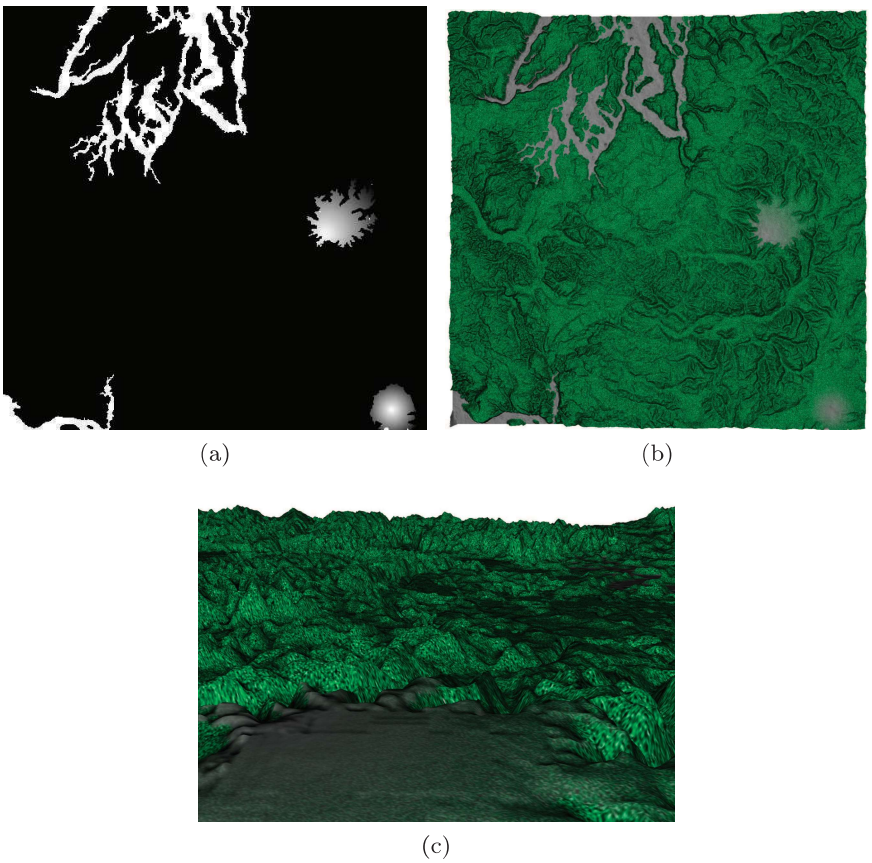


Figure 11.48. (a) Blend map used to blend between grass and stone color maps. (b) Top-down view of terrain rendered with the blend map. Black areas in the blend map correspond to grass and white areas correspond to stone. (c) A horizon view of the same terrain showing blending between grass and stone.

As shown in Listing 11.25, the code for a blend map is nearly identical to using a blend ramp. A blend map requires more memory than procedural techniques. To reduce memory consumption, the blend map does not need to be high resolution or cover an entire terrain tile, just the parts that require blending. It is also possible to use a sparse quadtree texture representation to significantly reduce memory usage [5].

For more information on blending color maps, see Jenks's article [77] and the early work on texture splatting by Bloom [17], which uses multipass rendering to composite textures in the framebuffer via alpha blending.

Texture stretching. When texture coordinates are computed based on a simple linear mapping of the vertex's xy-world-space position to texel space,

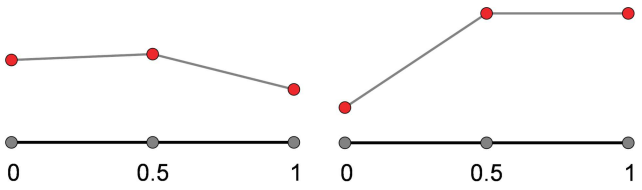


Figure 11.49. (a) When adjacent posts have similar heights, little texture stretching occurs because texel space is nearly evenly assigned to world space. (b) When adjacent posts have significant height differences, texture stretching occurs. Compare the world-space difference between the first two posts and the last two posts. Stretching will occur between the first two posts because not enough texel space is used.

texture-stretching artifacts occur on steep slopes. As shown in Figure 11.49, stretching occurs because uniformly distributing texture coordinates results in an equal amount of texels between each post, even if the height difference, and thus world-space distance, between adjacent posts is large. The same amount of texel space is assigned to long, steep areas as to shorter, flat areas, causing the texture to stretch on steep slopes.

There are many ways to minimize stretching. If detail mapping is used, stretching may not even be noticed. Since detail maps are tiled more frequently than color maps, there is less mismatch between a detail map's texel space and world space. The result of blending the detail map and color map tends to create enough detail on all but the steepest slopes.

Alternative solutions to stretching include assigning texture coordinates based on the steepness of adjacent posts, although this can shift the distortion to flatter areas. *Triplanar texturing* can reduce texture stretching by using the terrain's normal to blend between textures from one of three planes: the xy-plane, the xz-plane, and the yz-plane [58]. Of course, triplanar texturing comes with additional runtime overhead. Two-sided triplanar texturing is used in the C4 Engine to allow texturing of terrain features such as steep cliffs and cave interiors [165]. Finally, a precomputed *indirection map* can minimize stretching artifacts at the cost of a precomputation step that iteratively relaxes a spring network and uses one texture map and texture read at runtime [112].

11.5 Resources

As their popularity increases, a wealth of information is becoming available on non-height-map terrain representations. For voxels, Rosa's article

on destructible volumetric terrain in Miner Wars is a recent description of how voxels were used in a game [145]. Geiss's article on GPU-based procedural terrain is an excellent example of modeling terrain with an implicit function [59]. It is freely available online, along with all of *GPU Gems 3*.

Although most of the terrain-rendering literature focuses on LOD, there is still plenty of information on the fundamentals of rendering individual tiles. Vistnes's article is a great source for a complete implementation of displacement mapping in a vertex shader [177]. For ray-casting terrain, the work of Dick et al. includes a detailed analysis [38]. The DirectX SDK includes an implementation of GPU ray casting using cone step mapping [115].

Kris Nicholson's master's thesis provides a nice overview of GPU-based procedural terrain texturing [119]. Perhaps the ultimate resource to continue exploring terrain shading is Johan Andersson's description of terrain rendering in Frostbite [5].

12

Massive-Terrain Rendering

Virtual globes visualize massive quantities of terrain and imagery. Imagine a single rectangular image that covers the entire world with a sufficient resolution such that each square meter is represented by a pixel. The circumference of Earth at the equator and around the poles is roughly 40 million meters, so such an image would contain almost a quadrillion (1×10^{15}) pixels. If each pixel is a 24-bit color, it would require over 2 million gigabytes of storage—approximately 2 petabytes! Lossy compression reduces this substantially, but nowhere near enough to fit into local storage, never mind on main or GPU memory, on today's or tomorrow's computers. Consider that popular virtual globe applications offer imagery at resolution higher than one meter per pixel in some areas of the globe, and it quickly becomes obvious that such a naïve approach is unworkable.

Terrain and imagery datasets of this size must be managed with specialized techniques, which are an active area of research. The basic idea, of course, is to use a limited storage and processing budget where it provides the most benefit. As a simple example, many applications do not need detailed imagery for the approximately 70% of Earth covered by oceans; it makes little sense to provide one-meter resolution imagery there. So our terrain- and imagery-rendering technique must be able to cope with data with varying levels of detail in different areas. In addition, when high-resolution data are available for a wide area, more triangles should be used to render nearby features and sharp peaks, and more texels should be used where they map to more pixels on the screen. This basic goal has been pursued from a number of angles over the years. With the explosive growth in GPU performance in recent years, the emphasis has shifted from

minimizing the number of triangles drawn, usually by doing substantial computations on the CPU, to maximizing the GPU's triangle throughout.

We consider the problem of rendering planet-sized terrains with the following characteristics:

- They consist of far too many triangles to render with just the brute-force approaches introduced in Chapter 11.
- They are much larger than available system memory.

The first characteristic motivates the use of terrain LOD. We are most concerned with using LOD techniques to reduce the complexity of the geometry being rendered; other LOD techniques include reducing shading costs. In addition, we use culling techniques to eliminate triangles in parts of the terrain that are not visible.

The second characteristic motivates the use of out-of-core rendering algorithms. In out-of-core rendering, only a small subset of a dataset is kept in system memory. The rest resides in secondary storage, such as a local hard disk or on a network server. Based on view parameters, new portions of the dataset are brought into system memory, and old portions are removed, ideally without stuttering rendering.

Beautifully rendering immense terrain and imagery datasets using proven algorithms is pretty easy if you're a natural at spatial reasoning, have never made an off-by-one coding error, and scoff at those who consider themselves "big picture" people because you yourself live for the details. For the rest of us, terrain and imagery rendering takes some patience and attention to detail. It is immensely rewarding, though, combining diverse areas of computer science and computer graphics to bring a world to life on your computer screen.

Presenting all of the current research in terrain rendering could fill several books. Instead, this chapter presents a high-level overview of the most important concepts, techniques, and strategies for rendering massive terrains, with an emphasis on pointing you toward useful resources from which you can learn more about any given area.

In Chapters 13 and 14, we dive into two specific terrain algorithms: *geometry clipmapping* and *chunked LOD*. These two algorithms, which take quite different approaches to rendering massive terrains, serve to illustrate many of the concepts in this chapter.

We hope that you will come away from these chapters with lots of ideas for how massive-terrain rendering can be implemented in your specific application. We also hope that you will acquire a solid foundation for understanding and evaluating the latest terrain-rendering research in the years to come.

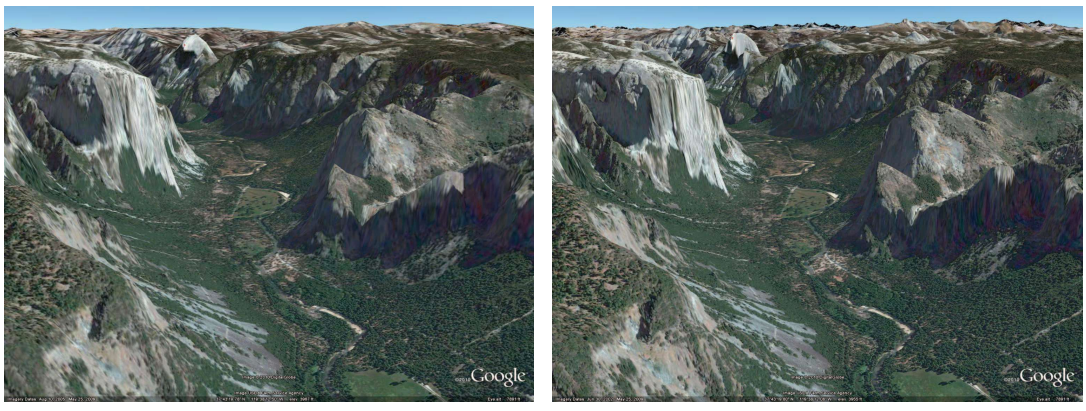
12.1 Level of Detail

Terrain LOD is typically managed using algorithms that are tuned to the unique characteristics of terrain. This is especially true when the terrain is represented as a height map; the regular structure allows techniques that are not applicable to arbitrary models. Even so, it is helpful to consider terrain LOD among the larger discipline of LOD algorithms.

LOD algorithms reduce an object's complexity when it contributes less to the scene. For example, an object in the distance may be rendered with less geometry and lower resolution textures than the same object if it were close to the viewer. Figure 12.1 shows the same view of Yosemite Valley, El Capitan, and Half Dome at different geometric levels of detail.

LOD algorithms consist of three major parts [3]:

- *Generation* creates different versions of a model. A simpler model usually uses fewer triangles to approximate the shape of the original model. Simpler models can also be rendered with less-complex shaders, smaller textures, fewer passes, etc.
- *Selection* chooses the version of the model to render based on some criteria, such as distance to the object, its bounding volume's estimated pixel size, or estimated number of nonoccluded pixels.
- *Switching* changes from one version of a model to another. A primary goal is to avoid *popping*: a noticeable, abrupt switch from one LOD to another.



(a) (b)

Figure 12.1. The same view of Yosemite Valley, El Capitan, and Half Dome at (a) low detail and (b) high detail. The differences are most noticeable in the shapes of the peaks in the distance. Image USDA Farm Service Agency, Image ©2010 DigitalGlobe. (Figures taken using Google Earth.)

Furthermore, we can group LOD algorithms into three broad categories: *discrete*, *continuous*, and *hierarchical*.

12.1.1 Discrete Level of Detail

Discrete LOD is perhaps the simplest LOD approach. Several independent versions of a model with different levels of detail are created. The models may be created manually by an artist or automatically by a polygonal simplification algorithm such as vertex clustering [146].

Applied to terrain, discrete LOD would imply that the entire terrain dataset has several discrete levels of detail and that one of them is selected for rendering at each frame. This is unsuitable for rendering terrain in virtual globes because terrain is usually both "near" and "far" at the same time.

The portion of terrain that is right in front of the viewer is nearby and requires a high level of detail for accurate rendering. If this high level of detail is used for the entire terrain, the hills in the distance will be rendered with far too many triangles. On the other hand, if we select the LOD based on the distant hills, the nearby terrain will have insufficient detail.

12.1.2 Continuous Level of Detail

In continuous LOD (CLOD), a model is represented in such a way that the detail used to display it can be precisely selected. Typically, the model is represented as a base mesh plus a sequence of transformations that make the mesh more or less detailed as each is applied. Thus, each successive version of the mesh differs from the previous one by only a few triangles.

At runtime, a precise level of detail for the model is created by selecting and applying the desired mesh transformations. For example, the mesh might be encoded as a series of *edge collapses*, each of which simplifies the mesh by removing two triangles. The opposite operation, called a *vertex split*, adds detail by creating two triangles. The two operations are shown in Figure 12.2.

CLOD is appealing because it allows a mesh to be selected that has a minimal number of triangles for a required visual fidelity given the viewpoint or other simplification criteria. In days gone by, CLOD was the best way to interactively render terrain. Many historically popular terrain-rendering algorithms use a CLOD approach, including Lindstrom et al.'s CLOD for height fields [102], Duchaineau et al.'s real-time optimally adapting mesh (ROAM) [41], and Hoppe's view-dependent progressive meshes [74]. Luebke et al. have excellent coverage of these techniques [107].

Today, however, these have largely fallen out of favor for use as runtime rendering techniques. CLOD generally requires traversing a CLOD data

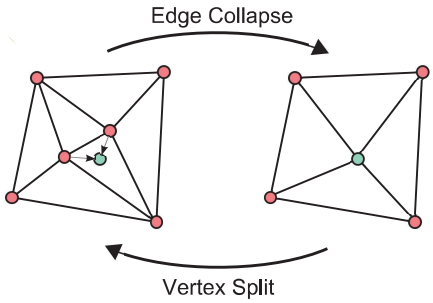


Figure 12.2. For an edge interior to a mesh, edge collapse removes two triangles and vertex split creates two triangles.

structure on the CPU and touching each vertex or edge in the current LOD. On older generations of hardware, this trade-off made a lot of sense; triangle throughput was quite low, so it was important to make every triangle count. In addition, it was worthwhile to spend extra time refining a mesh on the CPU in order to have the GPU process fewer triangles.

Today's GPUs have truly impressive triangle throughput and are, in fact, significantly faster than CPUs for many tasks. It is no longer a worthwhile trade-off to spend, for example, 50% more time on the CPU in order to reduce the triangle count by 50%. For that reason, CLOD-based terrain-rendering algorithms are inappropriate for use on today's hardware.

Today, these CLOD techniques, if they're used at all, are instead used to preprocess terrain into view-independent blocks for use with hierarchical LOD algorithms such as chunked LOD. These blocks are static; the CPU does not modify them at runtime, so CPU time is minimized. The GPU can easily handle the additional triangles that it is required to render as a result.

A special form of CLOD is known as infinite level of detail. In an infinite LOD scheme, we start with a surface that is defined by a mathematical function (e.g., an implicit surface). Thus, there is no limit to the number of triangles we can use to tessellate the surface. We saw an example of this in Section 4.3, where the implicit surface of an ellipsoid was used to render a pixel-perfect representation of a globe without tessellation.

Some terrain engines, such as the one in Outerra,[1] use fractal algorithms to procedurally generate fine terrain details (see Figure 12.3). This is a form of infinite LOD. Representing an entire real-world terrain as an implicit surface, however, is not feasible now or in the foreseeable future. For that reason, infinite LOD has only limited applications to terrain rendering in virtual globes.

[1] http://www.outerra.com

(a) (b)

Figure 12.3. Fractal detail can turn basic terrain into a beautiful landscape. (a) Original 76 m terrain data. (b) With fractal detail. (Images courtesy of Brano Kemen, Outerra.)

12.1.3 Hierarchical Level of Detail

Instead of reducing triangle counts using CLOD, today's terrain-rendering algorithms focus on two things:

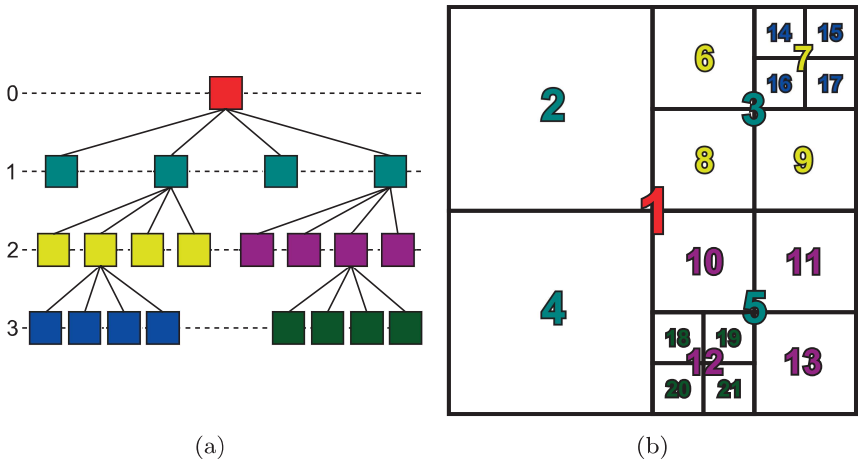
- Reducing the amount of processing by the CPU.
- Reducing the quantity of data sent over the system bus to the GPU.

The LOD algorithms that best achieve these goals generally fall into the category of hierarchical LOD (HLOD) algorithms.

HLOD algorithms operate on *chunks* of triangles, sometimes called *patches* or *tiles*, to approximate the view-dependent simplification achieved by CLOD. It some ways, HLOD is a hybrid of discrete LOD and CLOD. The model is partitioned and stored in a multiresolution spatial data structure, such as an octree or quadtree (shown in Figure 12.4), with a drastically simplified version of the model at the root of the tree. A node contains one chunk of triangles. Each child node contains a subset of its parent, where each subset is more detailed than its parent but is spatially smaller. The union of all nodes at any level of the tree is a version of the full model. The node at level 0 (i.e., the root) is the most simplified version. The union of the nodes at maximum depth represents the model at full resolution.

If a given node has sufficient detail for the scene, it is rendered. Otherwise, the node is refined, meaning that its children are considered for rendering instead. This process continues recursively until the entire scene is rendered at an appropriate level of detail. Erikson et al. describe the major strategies for HLOD rendering [48].

HLOD algorithms are appropriate for modern GPUs because they help achieve both of the reductions identified at the beginning of this section.



(a) (b)

Figure 12.4. In HLOD algorithms, a model is partitioned and stored in a tree. The root contains a drastically simplified version of the model. Each child node contains a more detailed version of a subset of its parent.

In HLOD algorithms, the CPU only needs to consider each chunk rather than considering individual triangles, as is required in CLOD algorithms. In this way, the amount of processing that needs to be done by the CPU is greatly reduced.

HLOD algorithms can also reduce the quantity of data sent to the GPU over the system bus. At first glance, this is somewhat counterintuitive. After all, rendering with HLOD rather than CLOD generally means more triangles in the scene for the same visual fidelity, and triangles are, of course, data that need to be sent over the system bus.

HLOD, however, unlike CLOD, does not require that new data be sent to the GPU every time the viewer position changes. Instead, chunks are cached on the GPU using static vertex buffers that are applicable to a range of views. HLOD sends a smaller number of larger updates to the GPU, while CLOD sends a larger number of smaller updates.

Another strength of HLOD is that it integrates naturally with out-of-core rendering (see Section 12.3). The nodes in the spatial data structure are a convenient unit for loading data into memory, and the spatial ordering is useful for load ordering, replacement, and prefetching. In addition, the hierarchical organization offers an easy way to optimize culling, including hardware occlusion queries (see Section 12.4.4).

12.1.4 Screen-Space Error

No matter the LOD algorithm we use, we must choose which of several possible LODs to use for a given object in a given scene. Typically, the

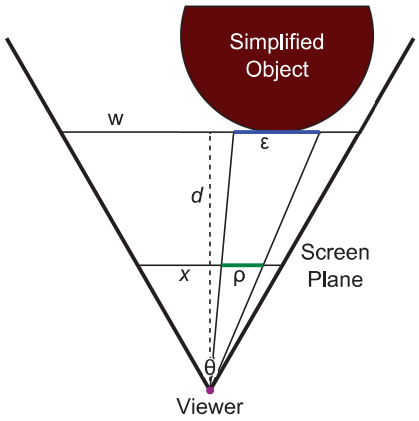


Figure 12.5. The screen-space error, ρ, of an object is estimated from the distance between the object and the viewer, the parameters of the view, and the geometric error, ϵ, of the object.

goal is to render with the simplest LOD possible while still rendering a scene that looks good. But how do we determine whether an LOD will provide a scene that looks good?

A useful objective measure of quality is the number of pixels of difference, or screen-space error, that would result by rendering a lower-detail version of an object rather than a higher-detail version. Computing this precisely is usually challenging, but it can be estimated effectively. By estimating it conservatively, we can arrive at a guaranteed error bound; that is, we can be sure that the screen-space error introduced by using a lower-detail version of a model is less than or equal to a computed value [107].

In Figure 12.5, we are considering the LOD to use for an object a distance d from the viewer in the view direction, where the view frustum has a width of w. In addition, the display has a resolution of x pixels and a field of view angle θ. A simplified version of the object has a geometric error ϵ; that is, each vertex in the full-detail object diverges from the closest corresponding point on the reduced-detail model by no more than ϵ units. What is the screen-space error, ρ, that would result if we were to render this simplified version of the object?

From the figure, we can see that ρ and ϵ are proportional and can solve for ρ:

$$\frac{\epsilon}{w} = \frac{\rho}{x},$$
$$\rho = \frac{\epsilon x}{w}.$$

The view-frustum width, w, at distance d is easily determined and substituted into our equation for ρ:

$$w = 2d \tan \frac{\theta}{2},$$

$$\rho = \frac{\epsilon x}{2d \tan \frac{\theta}{2}}. \tag{12.1}$$

Technically, this equation is only accurate for objects in the center of the viewport. For an object at the sides, it slightly underestimates the true screen-space error. This is generally considered acceptable, however, because other quantities are chosen conservatively. For example, the distance from the viewer to the object is actually larger than d when the object is not in the center of the viewport. In addition, the equation assumes that the greatest geometric error occurs at the point on the object that is closest to the viewer.

Kevin Says

For a bounding sphere centered at c and with radius r, the distance d to the closest point of the sphere in the direction of the view, $\mathbf{v}$, is given by

$$d = (c - \text{viewer}) \cdot \mathbf{v} - r.$$

By comparing the computed screen-space error for an LOD against the desired maximum screen-space error, we can determine if the LOD is accurate enough for our needs. If not, we refine.

12.1.5 Artifacts

While the LOD techniques used to render terrain are quite varied, there's a surprising amount of commonality in the artifacts that show up in the process.

Cracking. *Cracking* is an artifact that occurs where two different levels of detail meet. As shown in Figure 12.6, cracking occurs because a vertex in a higher-detail region does not lie on the corresponding edge of a lower-detail region. The resulting mesh is not watertight.

The most straightforward solution to cracking is to drop vertical skirts from the outside edges of each LOD. The major problem with skirts is that they introduce short vertical cliffs in the terrain surface that lead to texture stretching. In addition, care must be taken in computing the normals of the skirt vertices so that they aren't visible as mysterious dark or light

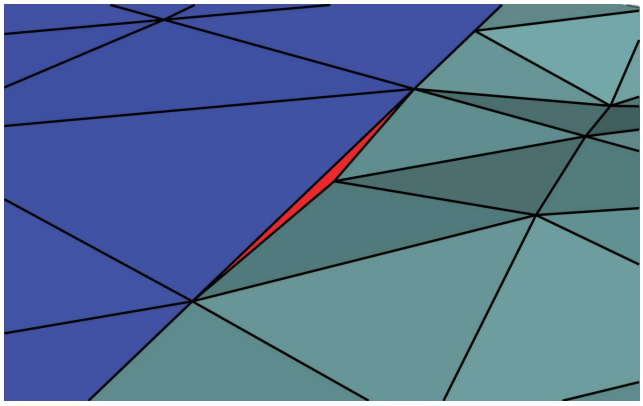


Figure 12.6. Cracking occurs when an edge shared by two adjacent LODs is divided by an additional vertex in one LOD but not the other.

lines around an LOD region. Chunked LOD uses skirts to avoid cracking, as will be discussed in Section 14.3.

Another possibility is to introduce extra vertices around the perimeter of the lower LOD region to match the adjacent higher LODs. This is effective when only a small number of different LODs are available or when there are reasonable bounds on the different LODs that are allowed to be adjacent to each other. In the worst case, the coarsest LOD would require an incredible number of vertices at its perimeter to account for the possibility that it is surrounded by regions of the finest LOD. Even in the best cases, however, this approach requires extra vertices in coarse LODs.

A similar approach is to force the heights of the vertices in the finer LOD to lie on the edges of the coarser LOD. The geometry-clipmapping terrain LOD algorithm (see Chapter 13) uses this technique effectively. A danger, however, is that this technique leads to a new problem: *T-junctions*.

T-junctions. T-junctions are similar to cracking, but more insidious. Whereas cracking occurs when a vertex in a higher-detail region does not lie on the corresponding edge of a lower-detail region, T-junctions occur because the high-detail vertex *does* lie on the low-detail edge, forming a T shape. Small differences in floating-point rounding during rasterization of the adjacent triangles lead to very tiny pinholes in the terrain surface. These pinholes are distracting because the background is visible through them.

Ideally, these T-junctions are eliminated by subdividing the triangle in the coarser mesh so that it, too, has a vertex at the same location as the vertex in the finer mesh. If the T-junctions were introduced in the first place in an attempt to eliminate cracking, however, this is a less than satisfactory solution.

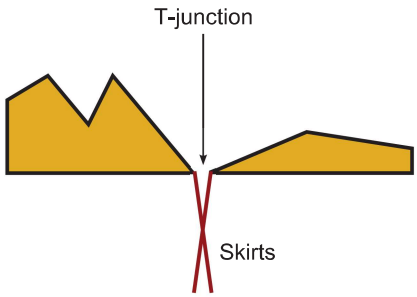


Figure 12.7. A side view of an exaggerated T-junction between two adjacent LODs. Skirts can hide T-junctions if they are angled slightly outward.

Another possibility is to fill the T-junctions with degenerate triangles. Even though these degenerate triangles mathematically have no area and thus should produce no fragments, the same rounding errors that cause the tiny T-junction holes to appear in the first place also cause a few fragments to be produced from the degenerate triangles, and those fragments fill the holes.

A final possibility, which is effective when cracks are filled with skirts, is to make the skirts of adjacent LODs overlap slightly, as shown in Figure 12.7.

Popping. As the viewer moves, the level of detail of various objects in the scene is adjusted. When the LOD of a given object is abruptly changed from a coarser one to a finer one, or vice versa, the user may notice a "pop" as vertices and edges change position.

This may be acceptable. To a large extent, virtual globe users tend to be more accepting of popping artifacts than, say, people playing a game. A virtual globe is like a web browser. Users instinctively understand that a web browser combines a whole lot of content loaded from remote servers. No one complains when a web browser shows the text of a page first and then "pops" in the images once they have been downloaded from the web server. This is much better than the alternative: showing nothing until all of the content is available.

Similarly, virtual globe users are not surprised that data are often streamed incrementally from a remote server and, therefore, are also not surprised when it suddenly pops into existence. As virtual globe developers, we can take advantage of this user expectation even in situations where it is not strictly necessary, such as in the transition between two LODs, both cached on the GPU. In many cases, however, popping can be prevented.

One way to prevent popping is to follow what Bloom refers to as the "mantra of LOD": a level of detail should only switch when that switch

will be imperceptible to the user [18]. Depending on the specific LOD algorithm in use and the capabilities of the hardware, this may or may not be a reasonable goal.

Another possibility is to blend between different levels of detail instead of switching them abruptly. The specifics of how this blending is done are tied closely to the terrain-rendering algorithm, so we cover two specific examples in Sections 13.4 and 14.3.

12.2 Preprocessing

Rendering a planet-sized terrain dataset at interactive frame rates requires that the terrain dataset be preprocessed. As much as we wish it were not, this is an inescapable fact. Whatever format is used to store the terrain data in secondary storage, such as a disk or network server, must allow lower-detail versions of the terrain dataset to be obtained efficiently.

As described in Chapter 11, terrain data in virtual globes are most commonly represented as a height map. Consider a height map with 1 trillion posts covering the entire Earth. This would give us approximately 40 m between posts at the equator, which is relatively modest by virtual globe standards. If this height map is stored as a giant image, it will have 1 million texels on each side.

Now consider a view of Earth from orbit such that the entire Earth is visible. How can we render such a scene? It's unnecessary, even if it were possible, to render all of the half a trillion visible posts. After all, half a trillion posts is orders of magnitude more posts than there are pixels on even a high-resolution display.

Terrain-rendering algorithms strive to be *output sensitive*. That is, the runtime should be dependent on the number of pixels shaded, not on the size or complexity of the dataset.

Perhaps we'd like to just fill a 1,024 × 1,024 texture with the most relevant posts and render it using the vertex-shader displacement-mapping technique described in Section 11.2.2. How do we obtain such a texture from our giant height map? Figure 12.8 illustrates what is required.

First we would read one post. Then we would seek past about 1,000 posts before reading another post. This process would repeat until we had scanned through nearly the entire file. Seeking through a file of this size and reading just a couple of bytes at a time would take a substantial amount of time, even from local storage.

Worse, reading just one post out of every thousand posts would result in aliasing artifacts. A much better approach would be to find the average or maximum of the 1,000 × 1,000 "skipped" post heights to produce one

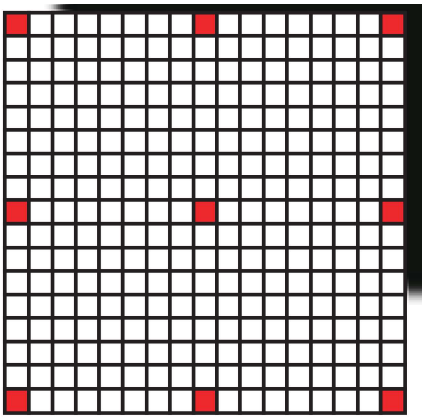


Figure 12.8. Rendering a low-detail representation of a height map when only a high-detail representation is available requires scanning past extraneous posts (white) to read the relevant ones (red). With a large height map, this is unworkable. Instead, we preprocess the height map.

height in the target low-resolution height map. Thus, rendering the lowest-detail view of the terrain requires that we read the entire high-detail terrain dataset. Clearly this is unworkable for real-time rendering.

Instead, we must preprocess the terrain dataset so that the low-detail representation is available quickly as a contiguous unit.

12.2.1 Height Maps to Mipmaps to Clipmaps

Thinking of a height map as an image, it is natural to address the problem described above by creating a *mipmap* from the image.

Mipmapping is a technique commonly used with textures. A mipmap consists of a number of mip levels, as shown in Table 12.1, each of which is conceptually a separate image. Mip level 0 contains the entire high-resolution image. Mip level 1 has half the texels of level 0 in each direction while still covering the entire extent of the texture. Mip level 2 has half again the texels of level 1. The highest mip level consists of just one pixel.

Many online sources of worldwide terrain and imagery present imagery as a giant mipmap. The Esri World Imagery map service,[2] for example, has 20 mip levels. The highest mip level covers the entire world with a single 256×256 image. The lowest mip level covers select areas with less than a meter between texels. If the entire world were instead covered by a single image with the highest resolution, it would contain quadrillions of pixels.

[2] http://server.arcgisonline.com/ArcGIS/rest/services/World_Imagery/MapServer

0	16,384 × 16,384	8	64 × 64
1	8,192 × 8,192	9	32 × 32
2	4,096 × 4,096	10	16 × 16
3	2,048 × 2,048	11	8 × 8
4	1,024 × 1,024	12	4 × 4
5	512 × 512	13	2 × 2
6	256 × 256	14	1 × 1
7	128 × 128		

Table 12.1. The mipmap levels and associated resolutions for an example 16,384× 16,384 texture. Creating a mipmap from a terrain height map allows faster access to low-resolution representations of the height map.

Consider rendering the 16,384×16,384 mipmapped texture in Table 12.1 to a full-screen quad on a 1,024×1,024 pixel display. In this case, mip level 4 (1,024 × 1,024) will be selected. As we zoom out, higher mip levels will be selected, consisting of fewer texels from the original texture. As we zoom in, a lower mip level will be selected (consisting of more texels), but only a subset of it will be visible.

So far, this is just a simplified explanation of mipmapping. However, there is an important insight in this that led to the development of a technique called clipmapping: the visible subset of the lower mip levels is limited by the resolution of the display. In fact, in our example, no more than 2,048 texels, or twice the resolution of the display, will ever be required from any given mip level. For this reason, it is not necessary to keep the entire 16,384 × 16,384 texture in video memory. We can simply compute the region of each mip level that can possibly be selected for a given scene, and "clip" the mipmap to that region.

Clipmapping is just such a method of clipping a mipmap to the subset that is needed to render the current scene [164]. A clipmap can be graphically depicted as a stack of levels forming an inverted pyramid, as in Figure 12.9. The topmost levels in the stack—the most detailed levels—are clipped to the same number of texels, which is determined from the display resolution.

In contrast with the usual convention for mip levels, clipmap levels are numbered from 0, the coarsest, least-detailed level, to $L-1$, the finest, most-detailed level. This convention is convenient for virtual globes because the number of clipmap levels often varies from region to region.

Virtual globes such as Google Earth use techniques like clipmapping to allow you to seamlessly navigate through enormous quantities of imagery [12]. In fact, it is not only unnecessary for the entire worldwide texture to fit into video memory—which is good because the texture is several thousands of gigabytes in size—but it also does not need to reside

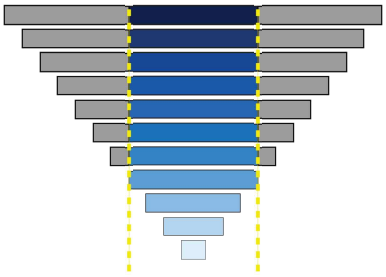


Figure 12.9. A clipmap is graphically depicted as a stack of levels forming an inverted pyramid. The topmost levels in the stack are all clipped to the same number of texels.

in system memory or even on a local disk. Virtual globes stream the clipped mipmap from a web server on demand.

While clipmapping was originally envisioned by Tanner et al. as a texture-mapping technique [164], a similar technique can be applied to terrain elevation data in a height map, which will be discussed in detail in Chapter 13.

12.2.2 Tiling

Mipmapping is a convenient way of preprocessing both height map and imagery data in order to render them at interactive frame rates. The alert reader may have noticed, however, that mipmapping alone does not completely solve the "seek a lot, read a little" problem that was our original motivation for creating mipmaps from the height-map data.

The problem, shown in Figure 12.10, occurs at the most detailed mip levels.

While it might make sense to render the entirety of a low-detail mip level, rendering all of a high-detail mip level is just as impractical as rendering the entire original height map. If each mip level is a single file, we still need to do a lot of seeking in order to read the subset of the level that is necessary to render the current scene.

For that reason, it is common to break each mip level into tiles. A tile has a specific number of texels, such as 256×256, but the area covered by the tile decreases with finer mip levels. For example, the coarsest mip level might cover the entire world ($(-180°, -90°)$ to $(180°, 90°)$ in longitude and latitude, respectively) with a single 256×256 tile. The next level has four 256×256 tiles with extents $(-180°, -90°)$ to $(0°, 0°)$, $(0°, -90°)$ to $(180°, 0°)$, $(-180°, 0°)$ to $(0°, 90°)$, and $(0°, 0°)$ to $(180°, 90°)$. Each successive level doubles the number of tiles in each direction and halves the extent

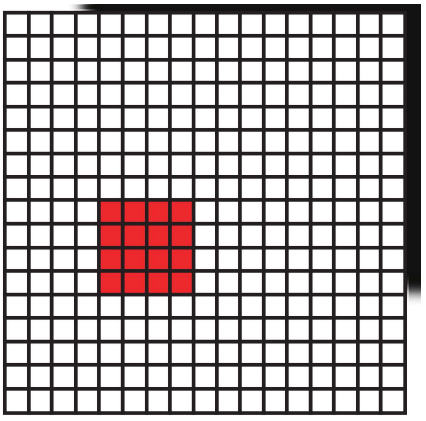


Figure 12.10. If an entire mip level is stored in a single file, rendering a subset of the mip level still requires scanning past clipped texels (white) to read the unclipped ones (red). This problem can be addressed by separating mip levels into tiles.

covered by each tile, as shown in Figure 12.11. Any given tile can be found quickly given its file name, well-known offset in a file, or, for the common case where the terrain or imagery is hosted on a server, its URL.

NASA, Esri, and other organizations arrange their terrain and imagery data in this manner and offer their enormous datasets through publicly accessible web servers. Though they differ in certain details, such as the size of each tile, the number of mip levels, the resolution of data available, and the scheme used for locating individual tiles, it's generally possible to adapt a given virtual globe rendering engine to render any of these datasets.

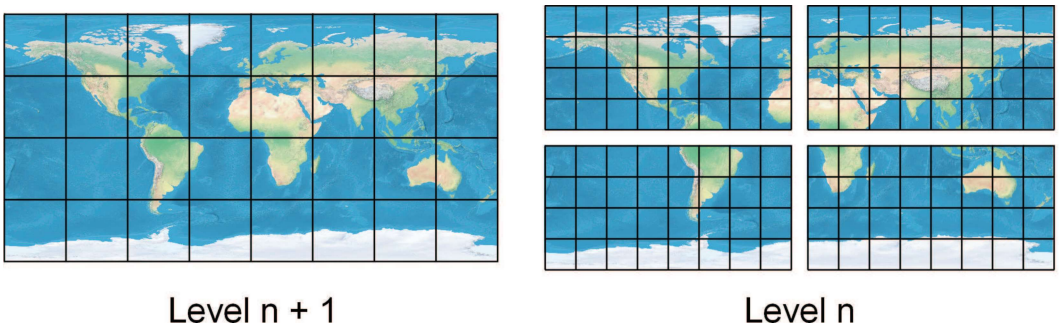


Figure 12.11. A mipmapped image is broken into tiles, all of which have the same number of texels. Each successive mip level has twice the number of tiles in each direction, and each tile covers one-quarter of the area.

Of course, there is some efficiency to be gained if the tiling scheme closely aligns with the rendering engine's expectations.

12.2.3 Mesh Simplification

We consider mipmapping and tiling to be the minimum amount of preprocessing necessary to render planet-sized height maps at interactive frame rates. However, some terrain-rendering algorithms require or benefit from additional preprocessing.

Mesh simplification is used to eliminate redundancy in an input terrain. For example, a height map for a flat region, such as the Great Plains of the United States, will have many more vertices than are necessary to faithfully represent the terrain.

There are many techniques and algorithms that can be used to simplify meshes, including *vertex decimation* [150], *quadric error metrics* [55], and *vertex clustering* [146]. These algorithms vary in their performance, their complexity, and the visual fidelity of the resulting mesh. In addition, some algorithms are only applicable to certain types of models. Height-map-based terrain, however, with its manifold topology, can be simplified with any of these simplification algorithms. These algorithms and more are covered by Luebke et al. [107].

In Section 14.5, we describe in detail one effective technique for simplifying a height-map terrain for use with the chunked LOD terrain-rendering algorithm.

12.3 Out-of-Core Rendering

Virtual globes invariably render terrain using out-of-core (OOC) algorithms. This means that the algorithms keep a subset of the terrain dataset in memory at any given time, while the rest of the dataset resides in secondary storage, such as a local hard disk or a cluster of network servers. Considering that virtual globe terrain datasets are often measured in terabytes, it comes as no surprise that the entire dataset cannot be in memory all at once.

Even with a relatively small terrain dataset, or on a machine with an unusually generous quantity of RAM, loading an entire terrain dataset into memory is impractical. An application that can't render its first scene until multiple gigabytes of data are loaded from a hard disk will have a long start up time. While a virtual globe might be a useful tool to visualize the progression of geologic time, it's best if we aren't tempted to use geologic time scales to measure its start up time.

The subset of data in memory at any given time is referred to as the *working set*, and in many cases, it is a very small fraction of the total

dataset. The goal of OOC rendering is to seamlessly bring new data into the working set as they are needed.

To that end, we need a policy for deciding which items to bring into the working set and in what order, which we call a *load-ordering policy*. Because the working set is not infinite in size, we eventually need to decide which items to remove from the working set in order to make room for new items; this is known as a *replacement policy*. Ideally, we also include an effective *prefetching* strategy so that items are brought into the working set just before the user notices that they're missing.

OOC rendering is not restricted to bringing data into system memory, however. In practice, the OOC data management system is responsible for moving data within a hierarchy of caches.

12.3.1 Cache Hierarchies

Effective OOC rendering of massive-terrain datasets requires a hierarchy of caches. A cache hierarchy consists of multiple types of memory along a continuum. At one end of the continuum is small, fast memory. At the other end, memory is much larger but also much slower. The multiple types of memory are used together to create the illusion of large, fast memory by taking advantage of coherence between memory accesses.

Consider an example cache architecture for a virtual globe application, as shown in Figure 12.12.

At the lowest level, all terrain and imagery data are stored on a network server, or perhaps a cluster of servers. The data are measured in tens or hundreds of terabytes, so accessing the entire dataset is an extremely time-consuming process. In addition, network latency and the need to serve multiple users simultaneously severely limits the speed at which the client application can retrieve data from the server.

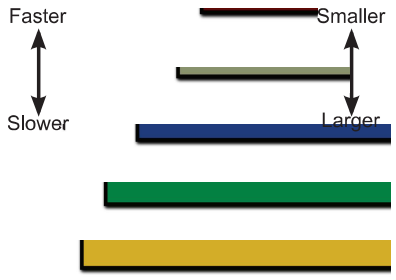


Figure 12.12. Virtual globe applications store terrain and imagery data in a hierarchy of caches. Caches at the bottom are the largest and slowest, while caches at the top are fast and relatively small.

Perhaps this network server is far away or can only be accessed over a slow network. In that case, it's useful to include another server in the hierarchy of caches. This server is accessible over a faster network, perhaps a LAN, and serves fewer users simultaneously. It caches a large subset of the data on the primary network server, but not all of it.

Next, the data are cached on the local hard disk of the computer that is running the virtual globe client application. This cache is rather large, perhaps tens of gigabytes, but much smaller than the tens or hundreds of terabytes of data stored on the network servers.

A subset of the data available on disk is stored in the process memory of the virtual globe client application. This level of the cache hierarchy might be measured in hundreds of megabytes or less, a substantial reduction again from the amount of data stored on disk.

Finally, the smallest subset of the dataset is stored in GPU memory, where it is available extremely quickly for rendering a frame.

A cache miss at any of these levels necessitates going to the next-lower cache in the hierarchy. As we go down the hierarchy, the cost of a miss gets higher, and the time until we actually have the data we need increases. For example, if a chunk of terrain data exists in system memory but is not yet in GPU memory, we can realistically copy it to the GPU within the current render frame. Of course, we still don't want to do that too often, though, or our frame rate will plummet.

On the other hand, if we need to go all the way to a remote network server to obtain the chunk of data, we may not actually be able to use it to render for several seconds. In the meantime, we must continue rendering using the best data we have available. This is an important point. In conventional computer architecture, a cache miss may result in a stall; we sit there and wait until the data are available. Although modern CPU architectures can frequently turn a stall into a context switch and allow the CPU to continue doing useful work while waiting for the data, the instruction using that data cannot execute until the data are returned. In terrain rendering, however, a cache miss does not result in a stall. Instead, it results in decreased visual quality as we use a lower-detail representation or simply omit the missing data.

The best way to decouple cache population from rendering is to use multiple threads. One or more worker threads download a new chunk of terrain data from a network server, load it into memory, and create the GPU buffer, all while the rendering thread renders potentially hundreds of frames. An alternative architecture where the rendering thread tries to do just a bit of this work each frame would be much more complicated.

In addition, a multithreaded architecture enables effective use of today's multicore CPUs. This is especially important if the worker thread does more than just shuffle data around. For example, it might decompress

data from its stored format, recompress it for the GPU, compute normals, etc. Job systems take this to an extreme and use several threads executing a large number of tasks related via a dependency graph [6].

Chapter 10 describes how a multithreaded architecture can be used to prepare resources, and it is just as applicable to terrain as it is to vector data.

12.3.2 Load-Ordering Policies

In the event of a cache miss during rendering, the item goes to the *request queue*. Perhaps several such misses occur while rendering a particular frame, and in the next frame, the viewer has moved, so new cache misses occur. How do we decide in what order to load the various items that have been requested?

Loading the cache items in the order they're requested is probably not a great strategy. When the viewer is moving quickly and generating a lot of cache misses, the worker thread is simply unable to keep up. By the time an item is loaded, the viewer may have moved such that the item is not even visible anymore.

A better strategy is to first load the item that was requested most recently and work backward in time from there. Typically, "most recently" means the most recent render frame. This way, the items that were needed for rendering in the last frame are loaded first, which makes sense because those same items are likely to be needed again in the next frame.

In some ways, the request queue, shown in Figure 12.13, acts more like a stack than a queue. A new item is pushed onto the head as a result of a cache miss, and the loading thread pops an item off the head to select the next item to load. One difference, however, is that the request queue does not operate in a strictly last-in, first-out (LIFO) order. Instead, existing

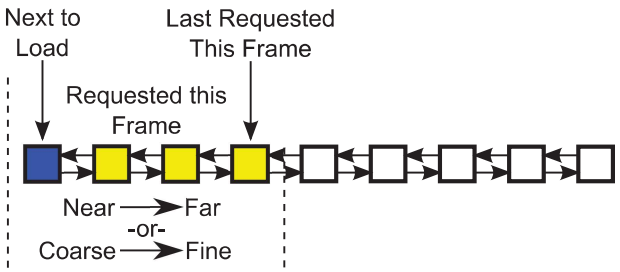


Figure 12.13. The request queue is a priority queue where the priority is determined by the frame number in which the item was requested and optionally by another metric, such as the distance of the viewer to the item. In many cases, the request queue can be implemented as a doubly linked list.

items in the queue are moved to the head each time they are requested by the rendering thread. In fact, the request queue is a *priority queue* in which the priority is determined by the frame number in which the item was last requested.

The priority of a request is determined by more than just the frame number in which it was last requested, however. For example, one useful intraframe ordering gives the items closest to the viewer higher priority in order to maximize the detail near the viewer. Alternatively, the items with the largest world-space extent, which are usually the items with the coarsest detail, can be loaded first in order to maximize the average detail across the scene.

Priority queues are usually implemented using tree-like data structures that provide $O(\log n)$ performance for inserting new items and for removing the highest-priority item. In the case of the request queue, however, we can implement the priority-based queuing using a double-linked list instead, gaining both performance and simplicity in the process. This is possible because the priority in the queue is monotonically increasing. New items always have the highest priority at the time they're added. An existing item that is already in the queue is also bumped to being the highest-priority item if it is requested again.

This property of the request queue makes it easy to implement using a doubly linked list. Newly requested items are added to the head of the list. If a requested item is already in the list, it is removed from its current position in the list and reinserted at the head. The loading thread, for its part, removes its next item to load from the head of the list. All of these operations happen in $O(1)$ time.

This request queue, as designed so far, will load first the items that were requested most recently. What may not be obvious, however, is how it can be used to achieve the intraframe load ordering mentioned previously. Among all the items in a single frame, how can we ensure that the items closest to the viewer are loaded first? Or, how can we ensure that the items with the largest extent are loaded first?

Fortunately, the natural structure of most terrain LOD algorithms makes it relatively easy to sort terrain items by these criteria. In quadtree-based algorithms, for example, it is easy to traverse child nodes in near-to-far order (see Section 12.4.5). Similarly, a breadth-first traversal of a quadtree results in visiting first the nodes with the largest extents.

These traversals result in orders that are exactly opposite to what we require, however. In our simplified priority queue, the priority of an item is determined from the order in which the item was added to the queue. So, in order to give closer items higher priority, we need to request those items last in the render frame. We do have one trick up our sleeves, however. Within a frame, we can request items in the order we'd like them loaded,

instead of in the opposite order, by simply keeping track of the *last* item requested in the frame and adding the new item after it. At the start of each render frame, we reset the reference to the *last* so that the first item requested is placed at the head of the list. This is the main reason we use a linked list instead of an array: the linked list allows us to insert items in the middle of the collection in constant time.

It is often useful to limit the number of items that can be waiting in the request queue in order to prevent the queue from growing indefinitely. For example, if the queue contains over 200 items, an old item at the tail of the list is removed each time a new one is added at the head.

12.3.3 Replacement Policies

When a cache is full and we need to unload an existing item in order to make room for a new one, we need to decide which item to unload. This is called a replacement policy because we are selecting which item to replace in the cache.

Replacement policies are important because a cache without a replacement policy is just another name for a memory leak. Data are added to the cache as needed and according to the load-ordering policy. Without a replacement policy, however, data are never removed from the cache, and the size of the cache grows indefinitely.

As you might imagine, the load-ordering policy and the replacement policy are closely related. It makes sense that the item we're loading right now should be last to be unloaded among the items currently loaded. In a particularly egregious violation of this principle, an item might be loaded, only to be immediately replaced with the next item loaded. In the next frame, the first item is loaded again, only to be unloaded again shortly thereafter. This effect, called *ping-ponging* or *cache thrashing*, is a waste of resources and should, of course, be avoided whenever possible.

It's not strictly true, though, that items should be replaced in the order in which they're loaded. Changes in viewer position affect the relative priority of items in the cache. For example, viewer movement to the other side of the Earth makes the cache items on the first side more eligible for replacement.

The canonical replacement policy for any cache is the least-recently used (LRU) replacement policy. The idea is simple and effective: the next item to be replaced is the one that was least recently needed for anything. Using this replacement policy requires us to keep track of the last time each cache item was accessed. Cache-item replacement is controlled by a *replacement queue*, shown in Figure 12.14, which is a restricted priority queue implemented as a linked list, much like the one we used for the request queue.

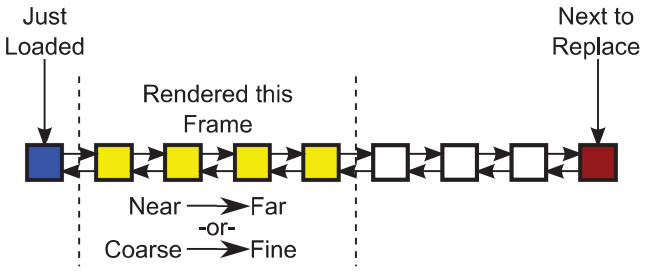


Figure 12.14. Newly loaded items are added to the head of the replacement queue, and the next item to replace is selected from the end. The items that were used to render the most recent frame are located near the head of the list and are sorted in one of two ways.

Once an item is loaded, it is added to the head of the replacement queue. When an item needs to be unloaded in order to make room for a new item, the item to unload is selected from the tail of the replacement queue. Putting newly loaded items at the head of the replacement queue minimizes cache thrashing because only items rendered in subsequent frames will be less eligible for replacement than the item just loaded. Each time a loaded item is used for rendering, it is moved to the head of the replacement queue, which delays its replacement.

It's important, however, that a new item is only loaded if it was last used more recently than the LRU item in the replacement queue. Otherwise, items near the tail of the request queue that were requested a very long time ago can replace much more useful items that were used relatively recently.

Over the course of several frames, the more recently used items migrate toward the head of the list and are less likely to be replaced. The less recently used items lag behind at the end of the list and are more likely to be replaced. If the same traversal strategy is used to update item positions in the replacement queue as is used to update item positions in the request queue, the items near the head of the replacement queue will be sorted near to far or coarse to fine, matching the priority with which the items would be loaded. At the head of the queue are the items that were just loaded since the last render frame.

12.3.4 Prefetching

As the viewer moves, different subsets of terrain and imagery data are required to render the scene most effectively. When required data are not yet available, a lower-resolution version of the terrain is rendered. For the best user experience, we should try to minimize the frequency with which this happens.

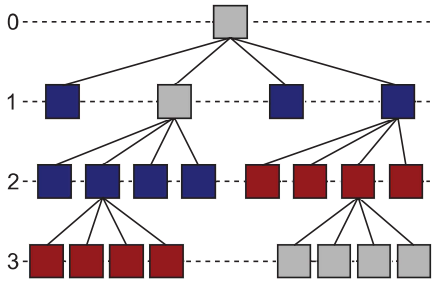


Figure 12.15. In an HLOD algorithm, an effective prefetching strategy is to prefetch the child nodes (red) of the nodes currently being rendered (blue).

In the normal course of events, we notice a missing piece of terrain or imagery data just at the moment we'd like to render it, so we request that it be loaded. At this point, we've already lost, though. The data will not be available for several frames. This is reactive; we notice that data are missing and react by requesting that they be loaded.

Prefetching, on the other hand, is proactive. We try to predict the subset of data that is likely to be needed in the near future and load it before it is needed. If we do this well, and perhaps with a little bit of luck, the data will already be loaded before we actually need it during rendering.

Prefetching for terrain is often dependent on the specific LOD algorithm used, but there are some common themes.

Ideally, the same metric is used for refinement and prefetching. Thus, items that are closer to becoming visible as a result of refinement are more likely to be prefetched. A simple but useful technique is to prefetch the data that are closest to the current position of the viewer, up to the limit of a specified cache budget. This maximizes detail near the viewer. Varadhan and Manocha describe a priority-based prefetching algorithm based on predicting when objects are going to switch LODs [175].

For HLOD algorithms, a simple and effective prefetching strategy is to prefetch the child nodes of nodes currently being rendered. Since the child nodes represent a more detailed representation of the current node, prefetching them is preparing for the eventuality that the viewer moves closer to the node and thus more detail is required. This is shown in Figure 12.15.

Similarly, it may be worthwhile to prefetch the parent node of a rendered node. This is optimizing for the case where the user zooms out or moves away from the node. While these coarser data are somewhat less likely to be missed by the user, rendering without it can cause aliasing artifacts and low frame rates because faraway detail is rendered with a high level of

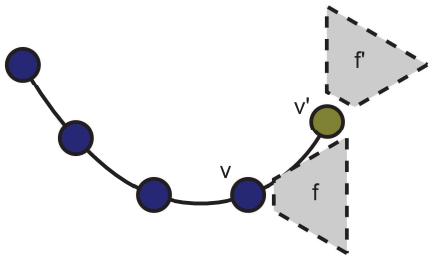


Figure 12.16. In the current frame, the viewer is at position v. The position of the next frame, v', is predicted, and the predicted view frustum, f', is used for prefetching.

detail. Also, the coarser data in parent nodes occupy far less memory than the fine data in child nodes because there are fewer parent nodes.

Another common technique for prefetching is to determine the data items that are needed, and load them based on an inflated view frustum. This is especially useful for LOD algorithms that are dependent on the direction of the viewer in addition to its position. It minimizes the chances of visual artifacts when rotating the camera in place.

A particularly interesting prefetching technique is to extrapolate the viewer's next position from his or her current position and linear and angular speeds and prefetch the data that will be needed for the new view. In this scheme, shown in Figure 12.16, a background thread analyzes the predicted next view frustum and issues requests to prefetch the necessary data. Correa et al. describe such an approach [29].

Depending on the speed of the viewer, the quantity of memory dedicated to prefetched data, and the accuracy with which the viewer's motion can be predicted, prefetching may completely eliminate cache misses during rendering. More commonly, however, especially in virtual globes, the maximum speed of viewer motion significantly exceeds the rate at which new data can be loaded. The goal is to improve the user experience, particularly when navigating within a relatively small area.

12.3.5 Compression

Compression reduces the size of geometry and texture data and is valuable at all levels of the cache hierarchy. Smaller data are retrieved from a remote server more quickly, they're faster to read from disk, and they take less time to send to the GPU over the system bus. Size is speed, especially when it comes to I/O. In addition, compressed data take less space in system and GPU memory, effectively allowing more data to fit in the cache or leaving more memory available for other purposes.

Instead of reducing memory usage and increasing performance, compression can also be used to increase visual quality within the same performance and storage budget. For example, a 512×512 DXT compressed RGBA texture consumes the same amount of memory as an uncompressed 256×256 RGBA texture and provides better image quality. DXT decompression is implemented in hardware on modern GPUs.

One simple form of compression is to store information implicitly rather than explicitly. A great example of this is the representation of terrain as a height map, as described in Section 11.1.1. A height map can be thought of as a compressed mesh where heights are stored explicitly and the horizontal coordinates are implicitly defined by the position of the height in the map. As a result, a height map occupies approximately one-third of the space required for an equivalent mesh. Of course, height maps are not appropriate for representing arbitrary meshes.

A related technique is mesh simplification, described in Section 12.2.3. Mesh simplification can be thought of as a form of compression because it reduces the number of vertices in a mesh without overly impacting its appearance. In fact, mesh simplification is a form of lossy compression because the original mesh cannot be reconstructed perfectly from its simplified representation. However, the simplified mesh is close enough to the original, and the space savings are substantial enough, that the lossiness is considered acceptable.

These two techniques compress by eliminating unnecessary information, for example, the horizontal coordinates in a height map or unimportant vertices in a simplified mesh. Most well-known compression algorithms, however, instead operate by eliminating redundancy or efficiently encoding patterns in the data. Such algorithms include everything from the lossless Deflate algorithm commonly used to compress ZIP and PNG files to lossy algorithms such as DXT and JPEG. Recent GPUs make it now possible to send compressed geometry to the GPU and decompress it in a geometry shader [101].

Section 10.3.3 describes an architecture for multithreaded decompression and recompression of textures and other resources.

12.4 Culling

In many scenes, only a small portion of the triangles composing the scene are actually visible. The rest are invisible, either because they are hidden behind other triangles or because they are outside of the field of view. For example, when looking at a location in Europe, it's unnecessary to render the Rocky Mountains in the western United States. Also, when zoomed in close to a mountain, it is unnecessary to render the foothills hidden

behind it. Culling reduces the amount of detail that needs to be rendered by eliminating these triangles that don't contribute to the scene.

12.4.1 Back-Face Culling

Perhaps the simplest form of culling is back-face culling, in which triangles that are facing away from the viewer are not rasterized (see Figure 12.17). Back-face culling can be used when the viewer is outside of a closed opaque object, such as a cube, without resulting in missing triangles. If a viewer were to enter the cube, for example, back-face culling would make the cube disappear; instead, front-face culling should be used when the viewer enters.

In virtual globes, back-face culling is also used for polygons rendered on the globe (see Section 8.2.6) and terrain. Missing back-facing triangles will be evident if a viewer were to go underneath terrain, but most virtual globes use collision detection to avoid this.

Back-face culling is simple to implement: a call to `glEnable` and `glCullFace` in OpenGL is all that is required to configure the rasterization pipeline for back-face culling. The only other requirement is that the vertices making up each triangle be specified in a consistent order, clockwise or counter-clockwise, so that the rasterization pipeline can identify which side of a triangle is considered to be the front.

Back-face culling happens late in the rasterization pipeline, however, so its benefits are limited. It is most valuable in applications like games with expensive fragment shaders. In applications that typically use simple fragment shaders, like virtual globes, the performance benefits are modest.

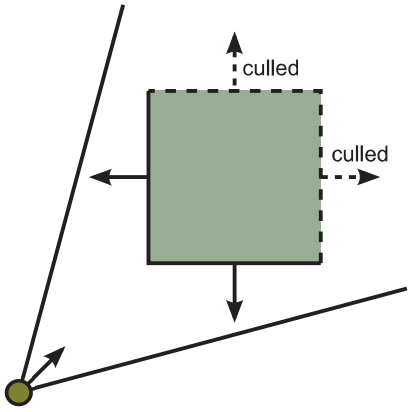


Figure 12.17. A top-down view showing two of four faces culled by back-face culling. Back-face culling is simple to implement and offers modest performance benefits.

12.4.2 View-Frustum Culling

View-frustum culling is a simple and effective culling technique that is used in virtually all terrain-rendering algorithms. Before rendering an object, the object is tested against the six planes of the view frustum. The test may indicate that the object is entirely inside or outside the frustum or that it intersects with one of the planes. If the object is inside or intersects, it is rendered; otherwise, it is discarded. Since the test happens on the CPU, the object is not processed at all by the GPU if it is not visible.

The object's true geometry is almost never tested against the view frustum. Instead, the test is done much more efficiently by fitting a bounding volume, such as a bounding sphere or AABB, around the object and testing that against the view frustum.

In terrain rendering, the number of objects to test against the view frustum can quickly become prohibitive. For example, a planet-sized terrain divided into average-sized chunks can result in millions of chunks to test against the view frustum. To optimize view-frustum culling, objects are organized into spatial data structures such as quadtrees. The bounding volumes of nodes in the spatial data structure are organized such that, if a node is not visible, neither are its children. This allows large parts of the scene to be culled quickly, minimizing CPU overhead. Furthermore, if a node is completely within the view frustum, so are its children, so the children do not need to be checked against the view frustum.

View-frustum culling is of limited value for terrain rendering when the viewer is far away from the globe and looking toward it. When the view encompasses the entire globe, the entire terrain is inside the frustum and nothing is culled. Level of detail, however, becomes important; at such an altitude, few, if any, terrain details are visible. When zoomed in, however, view-frustum culling is much more effective, potentially eliminating a large percentage of the terrain chunks that would otherwise be rendered.

12.4.3 Horizon Culling

Despite the effectiveness of view-frustum culling, objects inside the view frustum are not necessarily visible. In particular, objects are not visible if they're hidden behind other objects. Culling out objects that are occluded by other objects is known as *occlusion culling*.

When it comes to rendering terrain on a globe, one very big and important occluder is worth special consideration: the Earth itself. We'd like a way to quickly determine that a region of terrain is below the horizon, as shown in Figure 12.18, so that we don't waste any time rendering it. When the viewer is far away from the planet, horizon culling obviates the need to render about half of the planet. When the viewer is close to the

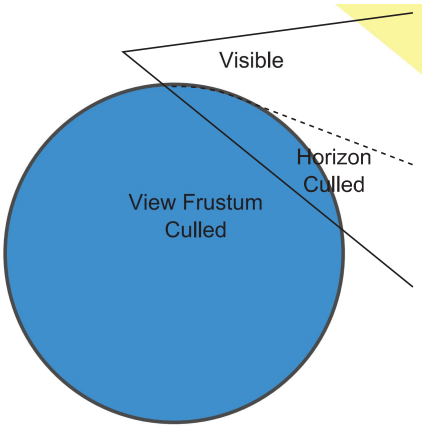


Figure 12.18. Horizon culling eliminates objects that are invisible because they are below the horizon of a globe's surface.

planet's surface, horizon culling really shines, potentially eliminating even more of the planet from consideration. While back-face culling eliminates the fragments on the back side of the planet, horizon culling allows us to eliminate the geometry as well.

Ohlarik presents a fast way to perform horizon culling given a bounding sphere for the potentially occluded object [125]. In Figure 12.19, the sphere centered at o is the bounding sphere of an object, perhaps a terrain tile, that is potentially occluded. The sphere centered at e is the object doing the occluding, such as the Earth. The point v is the position of the viewer.

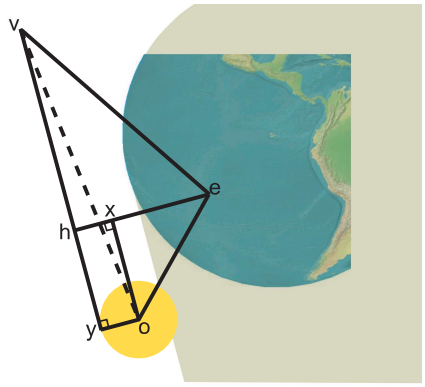


Figure 12.19. Occlusion of a smaller sphere by a larger one can be determined as a function of the distances between the spheres, their radii, and the distance from each to the viewer.

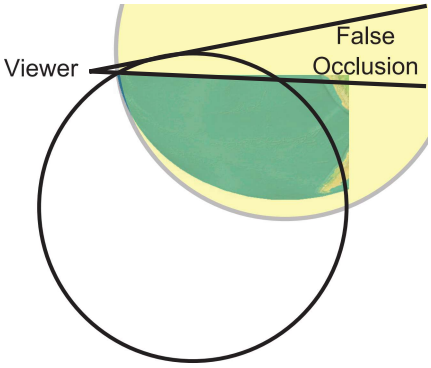


Figure 12.20. An occlusion sphere (yellow) that is larger than the Earth occludes the red region, but the Earth itself does not. Instead, the occlusion sphere must fit inside the Earth.

Anything in the tan region is occluded by the Earth from the viewpoint of the viewer.

If we were to wrap the Earth in a bounding sphere, we would use a sphere that completely encloses the Earth. In this case, though, we're using the sphere for occlusion, so we must use a sphere that fits entirely within the Earth's actual shape. If the occlusion sphere poked through the actual Earth surface anywhere, the portion above the surface could falsely occlude objects, as shown in Figure 12.20.

In Figure 12.19, the object o is invisible, but it is just on the edge of being visible. If it were to rotate clockwise around e, while remaining the same distance from it, it would move closer to v and also become visible. Given the distances between the two spheres; their radii, r_e and r_o; and the distance from the viewer to the center of the Earth, we can compute the smallest distance from the viewer to the object, $\|\mathbf{vo}\| = \|v - o\|$, for which the object is occluded by the Earth. If the distance to the viewer is smaller, at least part of the bounding sphere is not occluded.

First, observe that by the Pythagorean theorem

$$\|\mathbf{vo}\|^2 = (\|\mathbf{vh}\| + \|\mathbf{hy}\|)^2 + {r_o}^2. \tag{12.2}$$

Also, since $\mathbf{vh}$ and $\mathbf{eh}$ are perpendicular at point h, the Pythagorean theorem can be used again to compute $\|\mathbf{vh}\|$ and $\|\mathbf{hy}\|$:

$$\begin{aligned} \|\mathbf{vh}\| &= \sqrt{\|\mathbf{ve}\|^2 - {r_e}^2}, \\ \|\mathbf{hy}\| &= \sqrt{\|\mathbf{eo}\|^2 - (r_e - r_o)^2}. \end{aligned}$$

Note that $\|\mathbf{vh}\|$ is only a function of Earth, not of the object that is being tested for occlusion by Earth. Thus, when testing many objects for

occlusion, we can compute that quantity once and reuse it for all objects. Substituting $\|\mathbf{hy}\|$ into Equation (12.2), we get

$$\|\mathbf{vo}\|^2 = \left(\|\mathbf{vh}\| + \sqrt{\|\mathbf{eo}\|^2 - (r_e - r_o)^2}\right)^2 + {r_o}^2.$$

If $\|\mathbf{vo}\|^2$ is less than the square of the actual distance between the viewer and the center of the object's bounding sphere, the object is occluded. Otherwise, it is at least partially visible.

For terrain tiles, a bounding sphere that encompasses all of the vertices in the tile is not a very tight fit, which results in the horizon test determining that the tile is visible even when it is not. Ohlarik also presents a useful technique for choosing a very small bounding sphere—a single point, in fact—that can be used with this horizon-culling algorithm to cull terrain tiles more accurately [128].

Many virtual globe engines perform horizon and view-frustum culling hierarchically. Terrain chunks and other objects are arranged in a hierarchy of bounding volumes so that entire subtrees of objects can be tested for culling quickly. If a parent bounding volume is entirely below the horizon or outside the view frustum, all of its children are as well, so they do not need to be tested individually. If the parent bounding volume is entirely above the horizon and inside the view frustum, so are its children. Only if a parent bounding volume lies across the horizon or view-frustum boundary do its children need to be tested.

12.4.4 Hardware Occlusion Queries

When rendering terrain in a scene where the viewer is near the terrain surface and looking out toward the horizon, nearby terrain features are likely to occlude more distant terrain features. In an extreme case, imagine a viewer near the floor of the Grand Canyon. Terrain beyond the canyon walls is invisible to the viewer because it is occluded by the canyon walls themselves. Similarly, buildings or a dense forest might occlude much of the terrain in a scene.

Ideally, we would like to be able to take this occlusion into account in order to avoid rendering the invisible terrain. With large numbers of irregularly shaped occluders like trees and terrain features, however, determining occlusion is an extremely CPU-intensive process. Most applications that perform occlusion culling on the CPU make simplifying assumptions about the scene and the occluders. For example, in a city scene, we might assume that all buildings are rectangles extruded perpendicular to the ground. Horizon culling is a useful technique for the special case where the occludee is a large ellipsoid-shaped planet like Earth.

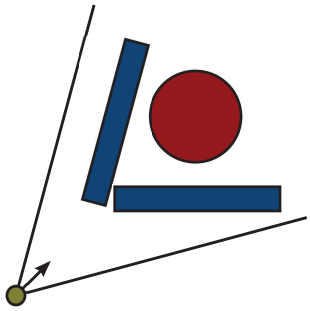


Figure 12.21. Neither of the two rectangles in this scene occlude the red circle, but the combination of both of them makes the red circle invisible to the viewer. This is called occluder fusion.

The problem is especially difficult when a region of terrain is only occluded by the fusion of multiple occluders, as shown in Figure 12.21. One tree, for example, is unlikely to occlude any reasonably sized terrain region, but an entire forest is another matter altogether. For general scenes like these, and in particular for the case of terrain occluding other terrain, the best approach to occlusion culling is to use the GPU to determine which objects are occluded.

When we render a scene, many of the fragments resulting from our rasterized triangles are discarded because they are clipped; that is, the triangles fall outside the view frustum. Other fragments are discarded because they fail a depth or stencil test or are explicitly discarded by the fragment shader. The rest pass all the way through the rasterization pipeline and modify the framebuffer. Hardware occlusion queries (HOQs) enable us to ask the GPU how many fragments were written to the framebuffer over the course of one or more draw calls. If no fragments were written while rendering an object, that object is occluded and does not need to be drawn in subsequent frames from the same viewpoint.

HOQs use the rasterization power of the GPU to determine if an object is visible. Typically, HOQs are performed by testing a bounding volume for occlusion rather than testing a detailed version of the object because an object's bounding volume usually has much less geometry than the object. In addition, HOQs are performed with color and depth writes disabled, allowing today's GPUs to use a higher-performance rendering path.

Because HOQs involve a roundtrip from the CPU, through the entire GPU pipeline, and back to the CPU, their naïve use creates two problems: *CPU stalls* and *GPU starvation*. If the CPU issues a query and then immediately waits for the result, the CPU is stalled; it does no useful work during the potentially lengthy period that the GPU is executing the query.

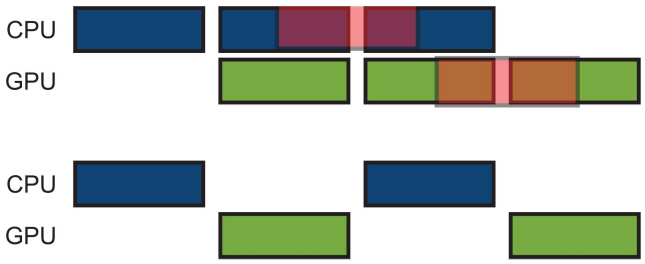


Figure 12.22. Improper use of HOQs stalls the CPU as it waits on the result of a query. The stalled CPU cannot issue commands to the GPU, which starves the GPU.

While the CPU is stalled, it does not issue new commands to the GPU, and thus starves the GPU. This is shown in Figure 12.22.

It can be useful to perform HOQs using multiple bounding volumes for an object instead of just one. This improves the tightness of the fit of the bounding volume around the object so it is more likely to be found to be occluded, while only minimally increasing the vertex costs.

Kevin Says

Effective occlusion algorithms based on HOQs exploit temporal coherence between frames to improve the parallelism between the CPU and GPU [153]. The basic idea is to issue a query in one frame but not check the result of the query until a later frame. Items that were visible in the earlier frame are assumed to be visible in the later one as well.

In addition to temporal coherence, effective occlusion algorithms take advantage of the spatial coherence of visibility within a frame [181]. A spatial data structure such as a quadtree, octree, or kd-tree is used to cull large occluded segments of a scene with minimal overhead.

HOQs are useful in scenes with high depth complexity, that is, scenes where many fragments compete for the same pixel. In urban walkthroughs, for example, large buildings near the viewer occlude most of the scene, so only a small subset of objects actually needs to be rendered. For terrain rendering, HOQs are potentially valuable with ground-level views. When the viewer position is high above the terrain, however, the terrain's depth complexity is low and occlusion culling offers little benefit.

12.4.5 Rendering Front to Back

Another simple culling technique is to render the scene from front to back; triangles closest to the viewer are rendered first, and triangles rendered thereafter are increasingly far from the viewer.

This functions as occlusion culling because of the characteristics of the GPU's depth buffer. Only fragments are culled, however, not triangles. A fragment that fails the depth test is not written to the framebuffer. By rendering front to back, more fragments fail the depth test, so less memory bandwidth is spent writing to the framebuffer.

More importantly, however, today's GPUs often do the depth test *before* invoking the fragment shader, an optimization called *early-z* [123, 136]. In this case, the fragment shader is not invoked if the fragment fails the depth test, which is desirable because there is no point in shading a fragment that never becomes a pixel. When complex fragment shaders are used, this offers substantial performance improvements.

Early-z is a fine-grained per-fragment test. Today's GPUs also implement *z-cull*, also called *hierarchical z*, which is a coarse-grained check that quickly tests a tile (e.g., an 8×8 block of fragments). These optimizations are enabled by default, but can be disabled in certain circumstances, most notably when a fragment shader outputs depth or uses discard as done for GPU ray casting in Section 4.3.

Obviously, to get the most out of early-z and z-cull, we should use these fragment-shader features sparingly. We can take it a step further: the more fragments that fail the depth test, the more effective these optimizations become. In many scenes, several fragments with the same screen-space location fight to win the depth test, and only one fragment becomes a pixel. The number of fragments per pixel is called the *depth complexity*. Since one fragment occludes the other fragments for a particular pixel (ignoring translucency), we must render triangles yielding the front-most fragments first so later fragments are not shaded or written to the framebuffer.

This is done by rendering in ascending order based on the distance to the viewer. Sorting individual triangles on the CPU is typically impractical, and precisely sorting entire objects may even be too expensive. Instead, we must balance the amount of time spent on the CPU optimizing for the GPU. A bucket sort can provide a coarse front-to-back ordering. Certain data structures also lend themselves to efficient front-to-back traversal.

Approximate front-to-back sorting is straightforward using quadtrees and octrees (see Figure 12.23). A node's children do not need to be explicitly sorted. Instead, the traversal order can be looked up in a table based on the viewer's position only; orientation isn't even needed. For a quadtree, only four unique child-traversal orders are required for front-to-back sorting, as shown in Table 12.2. The quadrant of the viewer relative

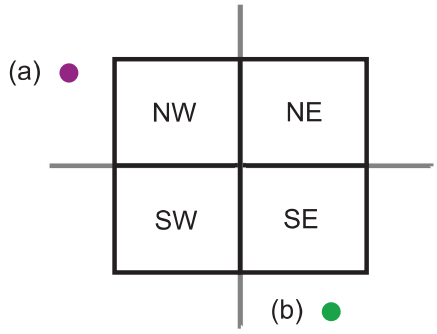


Figure 12.23. The front-to-back traversal order of a quadtree is easily determined from the quadrant of the viewer. For viewer location (a), the traversal is NW, SW or NE, SW or NE, SE. For (b), it is SE, SW or NE, SW or NE, NW.

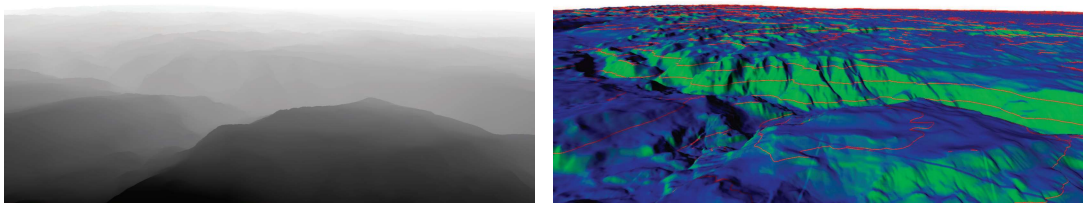
to the node is used to look up the traversal order. First, the child in the same quadrant as the viewer is rendered, followed by the two children adjacent to that child, and finally the child adjacent to the two children. The order of the second and third children does not matter; neither occludes the other, but they both potentially occlude the final child.

Another technique for taking advantage of early-z and z-cull is to render the scene in two passes. In the first pass, the scene is rendered to the depth buffer only (see Figure 12.24(a)). This pass is efficient because very simple fragment shaders can be used since no shading is required. In addition, GPUs also support an optimization called *double speed z-only* that improves rendering performance when color writes are disabled. This depth pass can also benefit from a coarse front-to-back sorting to reduce the number of depth buffer writes.

The second pass renders the entire scene again to perform the actual shading (see Figure 12.24(b)). This pass does not need to write to the depth

Viewer Quadrant	First	Second	Third	Fourth
Southwest	Southwest	Northwest or Southeast	Northwest or Southeast	Northeast
Southeast	Southeast	Southwest or Northeast	Southwest or Northeast	Northwest
Northwest	Northwest	Southwest or Northeast	Southwest or Northeast	Southeast
Northeast	Northeast	Northwest Southeast	Northwest Southeast	Southwest

Table 12.2. A quadtree can be rendered in front-to-back order by choosing one of four traversal orders based on the quadrant of the viewer.



(a) (b)

Figure 12.24. Rendering a (a) depth-only pass before (b) shading takes advantage of GPU depth buffer optimizations and effectively reduces the shading depth complexity to one.

buffer. Instead it reads from the depth buffer using a less than or equal to depth test. Only one fragment per pixel is shaded, effectively reducing the depth complexity to one. This shading pass can also be sorted by renderer state, reducing the tension between sorting by distance and sorting by state (see Section 3.3.6).

The downside is the scene's geometry is transformed twice. This can be mitigated by only rendering major occluders during the initial depth pass. Deferred shading is another rendering technique, which has become quite popular, that shades only nonoccluded fragments [142].

Early-z and z-cull, and occlusion culling in general, are particularly effective for horizon views like that of Figure 12.24(b) because the depth complexity is high. Other techniques can be used to reduce the depth complexity, such as defining a maximum view distance and using fog to gracefully fade out rendering over an interval at this distance. For some applications this is appropriate, but for many virtual globe use cases (e.g., simulations), users want the full view distance.

12.5 Resources

The Virtual Terrain Project[3] is an excellent resource for the aspiring (or even experienced) terrain-engine author. In particular, the "LOD Papers" page has an extensive catalog of published terrain LOD algorithms.

Level of Detail for 3D Graphics has thorough coverage of a number of topics relevant to massive-terrain rendering, including mesh simplification, error metrics, continuous LOD, and the perceptual aspects of LOD [107]. Because of advances in GPU hardware, however, the specific advice for terrain rendering is not as useful as it was when the book was originally

[3] http://www.vterrain.org/

published. Also, *Real-Time Rendering* has an extensive survey of culling techniques [3].

Erikson et al. cover many of the important techniques for rendering scenes using hierarchical LOD [48]. Gobbetti et al. survey useful techniques for visualizing massive models like terrain, including culling, cache-coherent layouts, data management, and simplification [63].

Many authors cover out-of-core rendering and prefetching, including Correa et al. and Varadhan and Manocha [29, 175]. Szofran offers a fascinating look at how terrain is managed in Microsoft Flight Simulator [163].

The blogs of folks working on virtual globes and terrain rendering are excellent sources of terrain-rendering tidbits, as well as inspiring screen shots. Some of our favorites include Outerra[4] and X-Plane.[5] Visit our website at http://www.virtualglobebook.com for more.

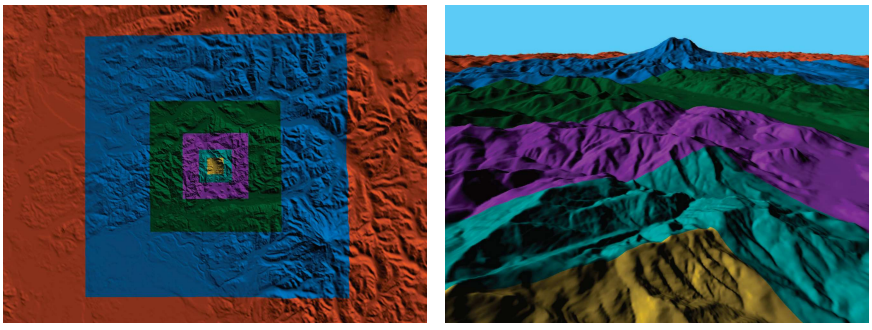
[4]http://outerra.blogspot.com/
[5]http://www.x-plane.com/blog/

13

Geometry Clipmapping

Geometry clipmapping is a terrain LOD technique in which a series of nested, regular grids of terrain geometry are cached on the GPU. Each of the grids is centered around the viewer and is incrementally updated with new data as the viewer moves. The grid levels form concentric squares, as shown in Figure 13.1.

Geometry clipmapping renders rasterized elevation data in the form of a height map. Prior to rendering, the height map is prefiltered into a mipmap pyramid, as described in Section 12.2. Each concentric square, called a clipmap level, corresponds to a level in the mipmap. Clipmap levels each have a texture in GPU memory that holds the subset of posts in the corresponding mip level that is closest to the viewer.



(a) (b)

Figure 13.1. (a) A geometry clipmap is rendered as a set of nested rings centered around the viewer. Each successive ring radiating away from the viewer has half the resolution of the ring before it while covering four times the area. (b) The clipmap from the viewer's perspective.

The innermost clipmap level closest to the viewer corresponds to the most detailed mip level of the height map, and successive levels radiating away from the viewer are increasingly less-detailed mip levels. Each clipmap level is square, and all levels have exactly the same number of posts, while covering four times the area of the next-finer level. All of the levels, except the innermost one, are rendered as hollow rings encircling the next, more-detailed level. At the finest level, the entire grid is rendered.

The clipmap levels are aligned with world space, meaning that the horizontal direction in the grid always corresponds to the x direction in world space, and the vertical direction in the grid corresponds to the y direction in world space.

Each level is rendered using the vertex-shader displacement-mapping technique described in Section 11.2.2. The grid (x, y) coordinate is input to the vertex shader, and the height at the grid point is read from the level's height-map texture using vertex texture fetch (VTF). The vertices in the grids are precisely aligned with posts in the associated mip level of the height map, so interpolation between texels is not required.

As the viewer moves, the clipmap levels are updated such that the clipmap pyramid remains centered on the viewer. Two-dimensional wrap-around texture addressing, also known as *toroidal addressing*, is used to eliminate the need to rewrite the entire height-map texture each time the viewer moves. Instead, only the newly visible posts are copied into the clipmap.

Geometry clipmapping has some nice features:

- *Simplicity.* The LOD and rendering algorithm are straightforward to implement.
- *Effective use of the GPU.* The algorithm makes good use of the GPU and is friendly to the CPU, leaving it largely available for other tasks.
- *Visual continuity.* Transitions between levels of detail are seamless and are easily implemented in shader programs.
- *Consistent rendering rate.* The number of triangles used to render a region is independent of the roughness or other characteristics of the terrain. Thus, the frame rate remains relatively steady.
- *Graceful degradation.* The rendering load can be reduced, when necessary, by reducing the size of each clipmap level or by disabling finer clipmap levels for a fast-moving viewer.
- *Minimal preprocessing.* Terrain is efficiently rendered from a simple mipmapped height map. Expensive or difficult to implement preprocessing steps are not required.

- *Compression and synthesis.* The hierarchy of nested, regular grids naturally lends itself to efficient compression and synthesis of terrain data.

On the other hand, geometry clipmapping has a few disadvantages:

- *More triangles.* Geometry clipmapping uses more triangles to attain a similar visual quality than do other terrain algorithms, such as the chunked LOD algorithm described in Chapter 14. This is because of the use of completely regular grids. Other terrain algorithms use irregular meshes that can reduce detail in flat areas of the terrain while maintaining high resolution in the bumpy sections. Effectively, geometry clipmapping assumes a worst-case terrain that is highly detailed throughout and optimizes for that case.

- *Modern GPU required.* Efficient implementation of geometry clipmapping requires a GPU that can quickly sample a texture from the vertex shader. While mainstream GPUs have had this capability for several years now, it may be a concern if older hardware must be supported.

- *Loose screen-space error guarantees.* Geometry clipmapping aims to produce triangles that are approximately the same size in screen space at each level. However, if the terrain has steep slopes, the triangles can be stretched vertically to arbitrary size. Thus, the error in screen space is a function of the terrain data itself and cannot be directly controlled by the algorithm. For applications that need a very precise representation of the terrain, this may be unacceptable.

- *It's patented.* US patent number 7436405, held by Microsoft, covers some aspects of this technique.

In some sense, geometry clipmapping continues the inevitable march toward GPU-centric terrain rendering. It requires the GPU to process more triangles than prior techniques. In exchange, it reduces CPU time. Perhaps surprisingly, memory usage and bus bandwidth are also reduced because the regular grid structure allows terrain data to be represented much more compactly.

This chapter begins with a detailed explanation of how to implement geometry clipmapping to render terrain extruded from a flat, horizontal plane. Then, we discuss how this technique can be used in a virtual globe application where terrain is extruded instead from an ellipsoid or sphere.

13.1 The Clipmap Pyramid

The entire clipmap pyramid, shown in Figure 13.2, consists of L levels, where L is based primarily on the number of mip levels of terrain data available. In contrast to the usual convention for mip levels, clipmap levels are numbered from 0, the coarsest, least detailed level, to $L-1$, the finest, most detailed level. This convention is convenient for virtual globes because the number of clipmap levels often varies from region to region. The NASA World Wind `mergedElevations` terrain dataset, for example, has 12 mip levels, numbered 0 through 11. It has 10 m resolution terrain data for most of the United States, 900 m resolution for the oceans, and 90 m resolution data for most of the rest of the world. Other World Wind terrain datasets include detail at 1 m resolution for Denmark and 30 m resolution for the rest of the world.

Each of the L clipmap levels has an $n \times n$ height-map texture associated with it. There is some flexibility in the value of n, subject to a couple of constraints.

First, n must be an odd number. As shown in Figure 13.3, this allows the posts in a coarser level to be coincident with those in the next-finer level at the boundary where the two meet. This is vital to ensure that there are no cracking artifacts between clipmap levels.

Second, n should be chosen with some consideration for texture sizes that are efficient for the GPU, and that generally means sizes that are nearly powers of two. The texture size is rounded up to the next power of two, and the extra row and column are unused. Asirvatham and Losasso both use $n = 255$, but values such as 511 and 1,023 are reasonable as well [9, 105]. Clipmap sizes that are not near powers of two can be used as well without impacting the algorithm, and they need not be rounded up to the next power of two, but it may lead to less efficient use of the GPU.

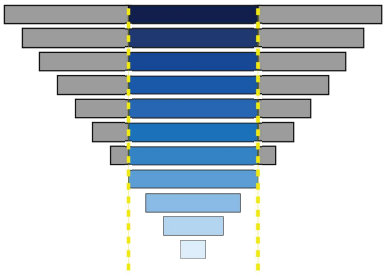


Figure 13.2. A geometry clipmap is graphically depicted as a stack of levels forming an inverted pyramid. The topmost levels in the stack are all clipped to the same number of posts.

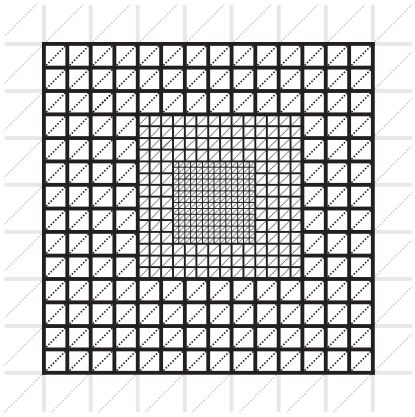


Figure 13.3. Vertices at the boundary between adjacent clipmap levels must be coincident to prevent cracks between the levels. This requires that clipmap levels have an odd number of posts.

The value n can be tweaked to trade rendering quality for performance on slower hardware.

The L levels of the pyramid are arranged in concentric squares, as shown in Figure 13.1. Level 0 covers the largest region of the world. Level 1 covers half of the world extent in each direction, or one-fourth of the area. Level 2 covers half again of the world extent in each direction, and so on. Each level is approximately centered around the viewer, subject to the overriding constraint that the posts of a finer clipmap level must be coincident with the posts of the next coarser-level at the clipmap level boundary.

Given a desired longitude and latitude on which to center the clipmap pyramid, the post indices of the southwest and northeast corners—the extent—of the finest, most detailed clipmap level are computed in Listing 13.1.

```
double centerLongitude = /* ... */;
double centerLatitude = /* ... */;

Level level = _clipmapLevels[_clipmapLevels.Length - 1];
double longitudeIndex =
    level.Terrain.LongitudeToIndex(centerLongitude);
double latitudeIndex =
    level.Terrain.LatitudeToIndex(centerLatitude);

int west = (int)(longitudeIndex - _clipmapPosts / 2);
if ((west % 2) != 0) ++west;
int south = (int)(latitudeIndex - _clipmapPosts / 2);
if ((south % 2) != 0) ++south;

level.NextExtent.West = west;
```

```
level.NextExtent.East = west + _clipmapPosts - 1;
level.NextExtent.South = south;
level.NextExtent.North = south + _clipmapPosts - 1;
```

Listing 13.1. The extent of the finest clipmap level within the terrain level is computed such that it is approximately centered around a given location.

```
int levelOffset = (_clipmapPosts + 1) / 4 - 1;

Level currentLevel = _clipmapLevels[i];
Level finerLevel = _clipmapLevels[i + 1];

level.NextExtent.West =
    finerLevel.NextExtent.West / 2 - levelOffset;
level.OffsetStripOnEast = (level.NextExtent.West % 2) == 0;
if (!level.OffsetStripOnEast) --level.NextExtent.West;
level.NextExtent.East =
    level.NextExtent.West + _clipmapPosts - 1;

level.NextExtent.South =
    finerLevel.NextExtent.South / 2 - levelOffset;
level.OffsetStripOnNorth = (level.NextExtent.South % 2) == 0;
if (!level.OffsetStripOnNorth) --level.NextExtent.South;
level.NextExtent.North =
    level.NextExtent.South + _clipmapPosts - 1;
```

Listing 13.2. The extent of a coarser clipmap level is computed from the extent of the finer level that it encloses.

Then, the origins of the remaining, coarser levels are computed in Listing 13.2.

Since each clipmap level's height-map texture has the exact same $n \times n$ size, and we now know the extent of the level in terms of terrain post indices, we are ready to fill the textures with post data.

Clipmap levels are incrementally filled with post data as the viewer moves, and this process is described in detail in Section 13.5. In addition, normals are incrementally computed from the post data and stored in a normal map for the level. For now, let's assume that the levels are already filled and move on to the matter of how to render them.

13.2 Vertex Buffers

With the height data stored in textures, the only per-vertex data we need to pass to the vertex shader is the (x, y) position of each grid point. Furthermore, these 2D vertex data are, up to a scale and translation, identical between clipmap levels. This is huge—it means that we can use a small number of static vertex and index buffers to render terrain for the entire scene. We create them once, hand them off to the GPU, and never have

to update them again. This is a major reason why the geometry-clipmap algorithm makes such efficient use of modern GPU hardware, and why it is so friendly to the CPU.

At a minimum, we need two vertex buffers. One vertex buffer contains vertices for an entire clipmap-level grid and is used for the finest-detail level that is rendered. The other vertex buffer is a ring with a hollow space in the middle for the next-finer level and is used for all clipmap levels except the finest. However, Asirvatham and Hoppe recommend breaking up the vertex buffer into a number of smaller patches, as shown in Figure 13.4 [9]. The major advantage of this approach, which is the one we use in OpenGlobe, is that it allows portions of a level to be culled by testing them against the view frustum. Asirvatham and Hoppe report that this results in a reduction of rendering load by a factor of about two to three. Another advantage is that less total vertex data are required, saving a bit of memory.

Modify OpenGlobe to test each patch against the view frustum before rendering it. By how much does this improve performance?

○○○○ Try This

Each clipmap level is broken up into the following patches:

- *Fill.* Most of the area covered by a clipmap level is filled by instances of a single $m \times m$ vertex buffer, where $m = \frac{(n+1)}{4}$. In the case of a clipmap size of $n = 255$, $m = 64$, so our fill patch has 64×64 vertices. We fill the outer ring of a non-finest clipmap level with 12 instances of this block, shown in yellow in Figure 13.4. For the finest level, we fill the interior of the ring with four additional instances. Adjacent blocks are overlapped at their edges to avoid gaps between blocks.

- *Horizontal and vertical fixups.* This leaves gaps two quads wide at the top and bottom of the ring and two quads high at the left and right of the ring. We fill these using vertex buffers of size $3 \times m$ and $m \times 3$, respectively, shown in blue in Figure 13.4. As before, these ring fixup buffers overlap the fill patches at their edges to avoid gaps. These fixup patches are also used to fill the gaps in the interior of the finest-detail ring.

- *Horizontal and vertical offset strips.* There is a one-quad gap along two inside edges of the outer ring, depending on how the next-finer clipmap level is situated within the ring. We render an offset strip on the south or north and west or east, as appropriate, shown in green

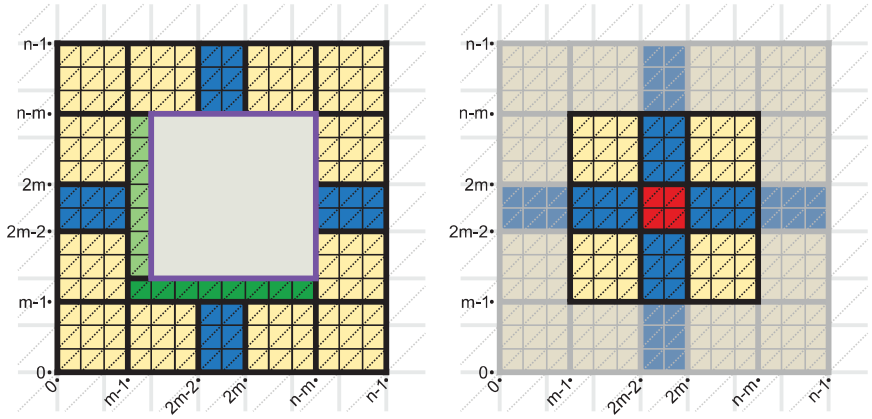


Figure 13.4. (a) All clipmap levels except the finest are rendered using multiple instances of six distinct, static vertex buffers. (b) The finest clipmap level is filled using a different configuration of three of the same static vertex buffers, plus one additional one. The labels on the axes indicate the indices of the posts within the level.

in Figure 13.4. The location of the offset strips is determined by the values of the `OffsetStripOnNorth` and `OffsetStripOnEast` properties that were computed in Listing 13.2.

- *Center.* The very center of the finest clipmap level is filled with a 3×3 patch, shown in red.

- *Degenerate triangles.* Finally, a ring of degenerate triangles is rendered where two different clipmap levels meet, shown as a purple border around the finer level in Figure 13.4. Even though these degenerate triangles technically have no area and thus should produce no fragments, we render them in order to avoid T-junctions in the clipmap mesh. As described in Section 12.1.5, T-junctions are places where an edge shared by two adjacent triangles is divided by an additional vertex when rendering one triangle but not when rendering the other. These T-junctions can show up as very small cracks between clipmap levels, so we fill them with degenerate triangles.

We use OpenGlobe's RectangleTessellator.`Compute` method to easily compute a vertex and index buffer for each of these patches.

Note that this particular breakdown of the clipmap level into patches imposes a requirement on the size, n, of the clipmap levels, in addition to

the requirement that it must be an odd number: $n + 1$ must be evenly divisible by four. Otherwise, the patches will not line up properly.

13.3 Vertex and Fragment Shaders

With the vertex buffers for the various patches ready to go, we now turn our attention to the shaders. A basic vertex shader is shown in Listing 13.3.

The only vertex attribute input to the vertex shader is `position`, which is a 2D vertex from one of the patches described above. The `position` is relative to the origin of the patch; in other words, the southwest vertex in the patch has position $(0, 0)$ and, for a "fill" patch, the northeast vertex is $(m - 1, m - 1)$.

```
in vec2 position;

out vec2 fsFineUv;
out vec3 fsPositionToLight;

uniform mat4 og_modelViewPerspectiveMatrix;
uniform vec3 og_sunPosition;
uniform vec2 u_patchOriginInClippedLevel;
uniform vec2 u_levelScaleFactor;
uniform vec2 u_levelZeroWorldScaleFactor;
uniform vec2 u_levelOffsetFromWorldOrigin;
uniform vec2 u_fineTextureOrigin;
uniform float u_heightExaggeration;
uniform float u_oneOverClipmapSize;
uniform sampler2D u_heightMap;

float SampleHeight(vec2 levelPos)
{
  fsFineUv = (levelPos + u_fineTextureOrigin) *
             u_oneOverClipmapSize;
  return texture(u_heightMap, fsFineUv).r *
         u_heightExaggeration;
}

void main()
{
  vec2 levelPos = position + u_patchOriginInClippedLevel;

  float height = SampleHeight(levelPos);
  vec2 worldPos = (levelPos * u_levelScaleFactor *
                   u_levelZeroWorldScaleFactor) +
                  u_levelOffsetFromWorldOrigin;
  vec3 displacedPosition = vec3(worldPos, height);

  fsPositionToLight = og_sunPosition - displacedPosition;

  gl_Position = og_modelViewPerspectiveMatrix *
                vec4(displacedPosition, 1.0);
}
```

Listing 13.3. A simple vertex shader for clipmap terrain rendering.

The first task of the shader is to translate the integer-valued components of the incoming vertex by the origin of the patch, `u_patchOriginInClippedLevel`. This gives us a new set of integer-valued coordinates that are expressed relative to the origin of the clipped level.

Next, the height of the vertex is sampled from the height-map texture. The texture coordinates are computed by adding the uniform `u_fineTextureOrigin` to the coordinates of the vertex within the level. Initially, `u_fineTextureOrigin` is set to $(0.5, 0.5)$ in order to make sure the vertex's height is sampled from the center of the corresponding height-map texel. As the viewer moves, the texture origin moves as well. This will be discussed in more detail in Section 13.5.

We now scale the vertex position up to world space by multiplying by `u_levelScaleFactor` and then by `u_levelZeroWorldScaleFactor`; `u_levelScaleFactor` is set by the CPU to 2^{-L}, where `L` is the zero-based index of the clipmap level currently being rendered. At level 0 (the coarsest), `u_levelScaleFactor` is 1. At level 1 it is 0.5, at level 2 it is 0.25, and so on. This reflects the fact that the distance between posts halves with each successive clipmap level. The object `u_levelZeroWorldScaleFactor` is the real-world space between posts at the coarsest clipmap level, level 0. Multiplying these two values together gives us the real-world space between posts at the current clipmap level.

Kevin Says ○○◯◯

Keeping `u_levelScaleFactor` and `u_levelZeroWorldScaleFactor` separate, rather than multiplying them together on the CPU and passing one uniform, ensures that coincident posts in adjacent levels actually end up coincident. Small integers like our post indices can be represented perfectly by a float, and they remain perfectly accurate when multiplied by powers of two like `u_levelScaleFactor`. Posts on the interlevel boundary exist in both levels. In the coarser level, `levelPos` is half the size it is in the finer level, but `u_levelScaleFactor` is twice as large. When multiplied with perfect accuracy in each level, the same answer is obtained both times, and `u_levelZeroWorldScaleFactor` is constant across all levels, so the posts really are coincident. However, the product of `u_levelScaleFactor` and `u_levelZeroWorldScaleFactor` is not necessarily representable with perfect accuracy. Errors due to rounding the two different values in the two levels can cause the common posts to not be coincident, creating cracks.

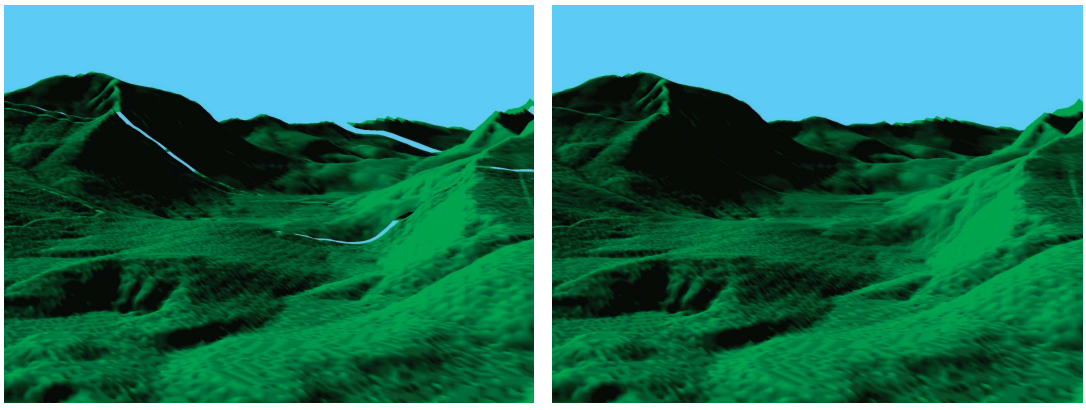
To this product, we add `u_levelOffsetFromWorldOrigin`, the world coordinates of the southwest corner of the clipped level, to get the 2D world

```
in vec2 fsFineUv;
in vec3 fsPositionToLight;

out vec3 fragmentColor;

uniform vec4 og_diffuseSpecularAmbientShininess;
uniform sampler2D u_normalMap;

vec3 SampleNormal()
{
  return normalize(texture(u_normalMap, fsFineUv).rgb);
}

void main()
{
  vec3 normal = SampleNormal();
  vec3 positionToLight = normalize(fsPositionToLight);

  float diffuse = og_diffuseSpecularAmbientShininess.x *
                  max(dot(positionToLight, normal), 0.0);
  float intensity = diffuse +
                    og_diffuseSpecularAmbientShininess.z;
  fragmentColor = vec3(0.0, intensity, 0.0);
}
```

Listing 13.4. The corresponding fragment shader for clipmap terrain rendering.

coordinates of the vertex. The final, displaced, world position of the vertex is simply this (x, y) position with the sampled height as the z-coordinate.

The only remaining steps for the vertex shader are to compute the vector to the sun for use by lighting calculations in the fragment shader and to transform the displaced position into clip coordinates using the model-view-perspective matrix.



(a) (b)

Figure 13.5. (a) Without blending, cracks (light blue) are clearly visible between clipmap levels. (b) Blending between clipmap levels eliminates the cracks.

The corresponding fragment shader is shown in Listing 13.4.

This simple fragment shader samples the level's normal map using the same texture coordinates that were used to sample the height map in the vertex shader. It then shades the fragment with a simplified version of the Phong-lighting computation explained in Section 4.2.1.

At this point, we can render nice scenes like the one shown in Figure 13.5(a). The cracks between clipmap levels are a blight on an otherwise picturesque landscape, though, so let's fix them.

13.4 Blending

Cracks occur because the vertices on the boundary between two clipmap levels are displaced twice; once when rendering the finer clipmap level and once when rendering the coarser one. The two heights read from the two different height maps are unlikely to be the same. An elegant solution to this problem is to introduce a "transition region" near the outer boundary of each clipmap level, as shown in Figure 13.6. Within the transition region, the vertex height is a blend of the heights obtained from the two height maps.

Losasso and Hoppe recommend that the width, w, of the transition region be $\frac{n}{10}$ [105]. In other words, vertices one-tenth of the way into the clipmap level will begin blending with the coarser level surrounding it. Specifically, if the distance from the viewer to the vertex along either

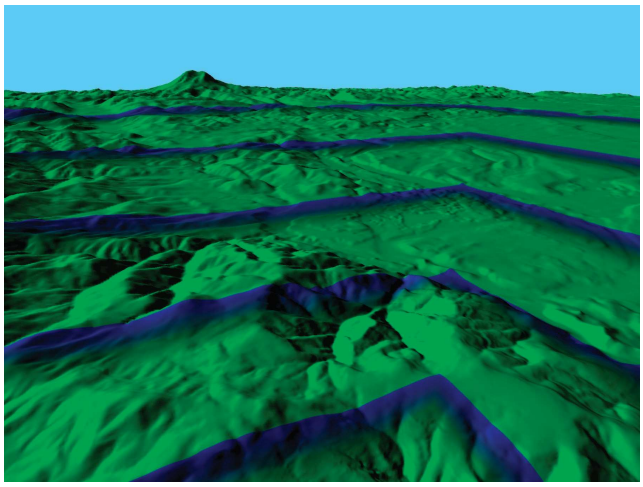


Figure 13.6. A transition region around the perimeter of each clipmap level smoothly blends the geometry with the next-coarser level.

dimension is greater than

$$\delta = \frac{n-1}{2} - w - 1,$$

then the vertex is within the transition region. Furthermore, we want vertices at the outer perimeter of this region to be displaced exclusively by the coarser height map. We compute a blend parameter, $\alpha = \max(\alpha_x, \alpha_y)$, where

$$\alpha_x = \text{clamp}\left(\frac{|x - v_x| - \delta}{w}, 0, 1\right), \tag{13.1}$$

and similarly for α_y. The position (v_x, v_y) is the position of the viewer expressed in the $[0, n-1]$ coordinate system of the clipped level.

We can easily modify the `SampleHeight` vertex-shader function in Listing 13.3 to compute α and blend the two texture samples. The updated function, plus the new required uniforms, are shown in Listing 13.5.

This function obtains the height of the vertex from the current, finer height map just as before. However, it also obtains a height from the next-coarser height map. The texture coordinates in the coarser height

```
out vec2 fsCoarseUv;
out float fsAlpha;

uniform vec2 u_fineLevelOriginInCoarse;
uniform vec2 u_viewPosInClippedLevel;
uniform vec2 u_unblendedRegionSize;
uniform vec2 u_oneOverBlendedRegionSize;
uniform sampler2D u_fineHeightMap;
uniform sampler2D u_coarserHeightMap;

float SampleHeight(vec2 levelPos)
{
  fsFineUv = (levelPos + u_fineTextureOrigin) *
             u_oneOverClipmapSize;
  fsCoarseUv = (levelPos * 0.5 + u_fineLevelOriginInCoarse) *
               u_oneOverClipmapSize;

  vec2 alpha = clamp((abs(levelPos - u_viewPosInClippedLevel) -
                      u_unblendedRegionSize) *
                     u_oneOverBlendedRegionSize, 0, 1);
  fsAlpha = max(alpha.x, alpha.y);

  float fineHeight = texture(u_fineHeightMap, fsFineUv).r;
  float coarseHeight =
      texture(u_coarserHeightMap, fsCoarseUv).r;
  return mix(fineHeight, coarseHeight, fsAlpha) *
         u_heightExaggeration;
}
```

Listing 13.5. The `SampleHeight` vertex-shader function is modified to sample heights from two adjacent levels and blend them.

map are obtained by halving the `levelPos` coordinates and then adding `u_fineLevelOriginInCoarse`, which is the origin of the fine clipmap level expressed in the coordinates of the coarse clipmap level.

The blend parameter `alpha` is computed according to Equation (13.1); `u_viewPosInClippedLevel` is (v_x, v_y), `u_unblendedRegionSize` is δ, and `u_oneOverBlendedRegionSize` is $\frac{1}{w}$. The blend parameter is used to linearly blend, using the GLSL mix function, between the `coarseHeight` at the outer perimeter of the transition region and the `fineHeight` at the inner perimeter of the transition region. Outside the transition region, `alpha` is zero so the `fineHeight` is used exclusively.

Note that we output both the coarse texture coordinates and the `alpha` value to the fragment shader.

Try This ○○◯◯

Reduce the number of required texture lookups by packing more information into a single texture. For example, the coarse height can be stored as the whole-number portion of the float, and the difference between the coarse and fine heights, divided by a suitable scale factor, can be stored as the fractional portion of the float. A simpler approach is to use a 16-bit floating-point texture instead of a 32-bit one.

In addition to blending the heights, we also need to blend the normals. The updated `SampleNormal` fragment-shader function is shown in Listing 13.6.

This blending technique does a nice job of hiding transitions between different levels of detail, as shown in Figure 13.5(b).

Kevin Says ○○◯◯

When I first implemented the clipmap-based terrain rendering in OpenGlobe, I thought I was clever. Instead of blending the coarse and fine normals, as shown here, I computed the normal in the vertex shader by finite differencing the blended heights, as shown in Section 11.3.1. This sounds like almost the same thing, but it's not. It created clear discontinuities at the level boundaries that looked like advancing waves as the viewer moved around the world.

Cracking between different levels of detail is a common problem with many terrain-rendering algorithms. The specifics of the clipmap-based rendering approach allow this problem to be dealt with in an extremely elegant

```
in vec2 fsCoarseUv;
in float fsAlpha;

uniform sampler2D u_fineNormalMap;
uniform sampler2D u_coarserNormalMap;

vec3 SampleNormal()
{
  vec3 fineNormal =
      normalize(texture(u_fineNormalMap, fsFineUv).rgb);
  vec3 coarseNormal =
      normalize(texture(u_coarserNormalMap, fsCoarseUv).rgb);
  return normalize(mix(fineNormal, coarseNormal, fsAlpha));
}
```

Listing 13.6. The `SampleNormal` fragment-shader function is modified to sample normals from two adjacent levels and blend them according to the `alpha` value computed by the vertex shader.

way. Unlike the skirts or flanges that are commonly used in other terrain-rendering algorithms, blending produces a mesh without discontinuities at the boundaries between levels of detail that can lead to texture stretching. Significantly, this extends to the surface normals used for lighting.

13.5 Clipmap Update

As the viewer moves, the clipmap levels move as well, remaining approximately centered around the viewer. The process of updating a clipmap level as a result of viewer motion is shown in Figure 13.7.

We keep a series of terrain height-map tiles cached in textures on the GPU. As explained in Section 12.2.2, terrain datasets are typically broken up into tiles so that a particular region can be accessed more quickly. When rendering such a dataset, these terrain tiles are convenient units in which to send terrain data to the GPU, particularly if they already have a power-of-two size. In the unlikely event that the terrain data source is not already tiled, it should be logically broken up into tiles anyway in order to allow data to be uploaded to the GPU in convenient tile-sized chunks.

Clipmap level updates take place almost entirely on the GPU by copying newly visible post data from one or more of these tiles to the appropriate place in the clipmap level's height-map texture.

The copy operation is performed by rendering a quad textured with the tile's height-map texture to a framebuffer with the clipmap level's height-map texture as a color attachment. A simple fragment shader does the actual copying. This strategy allows us to send the tile's height-map data over the system bus once, at least as long as the tile remains cached (see

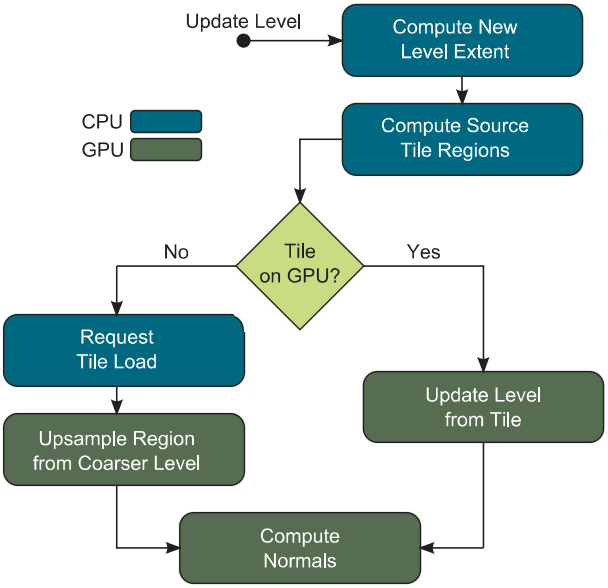


Figure 13.7. The overall process of updating a clipmap level in response to viewer motion. The colors of the boxes indicate whether the operation primarily takes place on the CPU or GPU.

Section 13.5.7), and then do clipmap updates with little additional bus traffic.

Similarly, the normal map for each level is updated by rendering a quad textured with the level's height map to a framebuffer with the level's normal map attached and computing the normal in the fragment shader.

Kevin Says ○○◯◯

In OpenGlobe, we use a single-channel, floating-point texture to store the heights, and a three-channel, floating-point texture to store the normals. For some applications, using 4 bytes per height and 12 bytes per normal will be overkill, in which case a more compact representation can be used instead.

If a tile is not yet resident on the GPU, it is queued for load by a worker thread, and the corresponding region in the level is filled by upsampling heights from the next-coarser level.

13.5.1 Toroidal Addressing

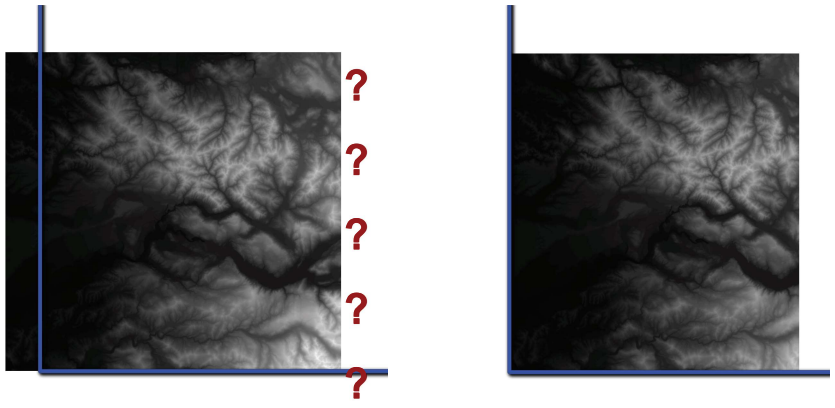
We use TextureWrap.`Repeat`, also known as 2D wraparound or toroidal addressing, to incrementally update clipmap levels. The texture can be thought of as occupying a torus, where the texture wraps around on itself in both directions.

Consider the case where the viewer moves toward the east, as shown in Figure 13.8. In this case, in order to remain centered around the viewer, the complete set of height data for the level must shift toward the left. The westernmost columns of height data fall off the edge of the level height texture, and the easternmost columns are filled with posts from the height map that were not previously visible.

Because every post shifts position when the viewer moves, it may appear at first that we need to rewrite the entire texture every time. Fortunately, this is not the case.

Instead, we toroidally address the height texture using a sampler configured to use TextureWrap.`Repeat`. Then, when the viewer moves, the origin of the level within the texture moves as well. In our example above, the origin is initially at $(0, 0)$. After the viewer moves one post to the east, the origin is changed to $(1, 0)$. This means that column 1, the second column, is now the westernmost column of posts in this clipmap level. By wraparound texture addressing, column 0, the first column, is the easternmost column of posts.

This is immensely convenient. It means that the region in the texture occupied by the *old* westernmost posts, which are no longer necessary when



(a) (b)

Figure 13.8. (a) As the viewer moves, clipmap levels are incrementally updated with new data. (b) Toroidal addressing eliminates the need to write the entire height-map texture. Instead, new posts replace old ones.

the clipmap level moves to the east, is the same region that is now needed for the *new* easternmost posts. Instead of rewriting the entire clipmap level, we only need to incrementally populate it with the newly visible posts, overwriting old posts that are no longer visible.

The origin of the level within the texture is passed to the vertex shader shown in Listing 13.3 as `u_fineTextureOrigin`, plus 0.5 in each direction so that heights are sampled from the centers of height texels.

13.5.2 The Update Region

Section 13.1 described how to compute the extent of a clipmap level from the viewer position. The displacement of this extent since the last render frame defines the region that needs to be updated. It also determines the origin of the clipmap level in the level's height map.

The code to compute the origin, `level.OriginInTextures`, is shown in Listing 13.7. The object `CurrentExtent` is the extent, in integer-valued terrain coordinates, of the clipmap level on the last render frame; `NextExtent` is the extent computed from the new viewer position. At the end of the update process, the `NextExtent` becomes the `CurrentExtent`.

The `OriginInTextures` is simply displaced by the change in the southwest corner of the clipmap level's extent since the last frame. If the origin falls off one side of the texture, it wraps around to the other side.

The region that needs to be updated is the region that is new to the clipmap level extent this render frame and is computed as shown in Listing 13.8.

If the viewer moved to the east (`delta X` > 0), the region that needs to be updated starts just after the easternmost post in the previous clipmap-

```
int deltaX = level.NextExtent.West - level.CurrentExtent.West;
int deltaY = level.NextExtent.South - level.CurrentExtent.South;

// if deltaX and deltaY are 0, no update is necessary

int newOriginX =
    (level.OriginInTextures.X + deltaX) % _clipmapPosts;
if (newOriginX < 0)
  newOriginX += _clipmapPosts;
int newOriginY =
    (level.OriginInTextures.Y + deltaY) % _clipmapPosts;
if (newOriginY < 0)
  newOriginY += _clipmapPosts;

level.OriginInTextures = new Vector2I(newOriginX, newOriginY);
```

Listing 13.7. The new origin of the clipmap level within the level's height-map texture is computed from the motion of the level's extent since the last frame.

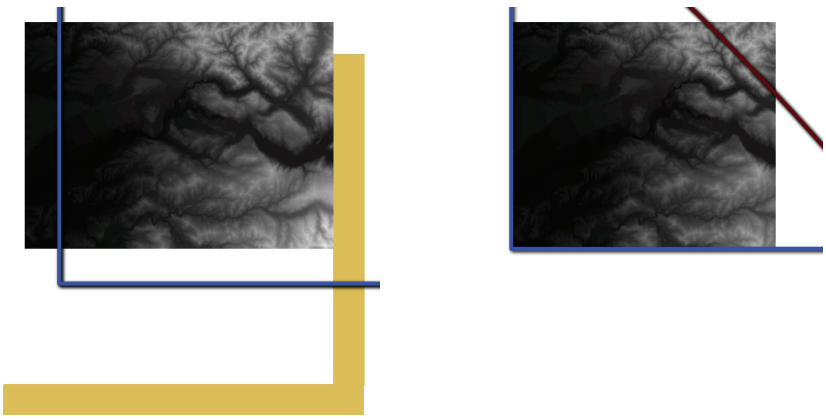
```
int minLongitude = deltaX > 0 ? level.CurrentExtent.East + 1
                              : level.NextExtent.West;
int maxLongitude = deltaX > 0 ? level.NextExtent.East
                              : level.CurrentExtent.West - 1;
int minLatitude = deltaY > 0 ? level.CurrentExtent.North + 1
                             : level.NextExtent.South;
int maxLatitude = deltaY > 0 ? level.NextExtent.North
                             : level.CurrentExtent.South - 1;

int width = maxLongitude - minLongitude + 1;
int height = maxLatitude - minLatitude + 1;

if (height > 0)
{
  ClipmapUpdate horizontalUpdate = new ClipmapUpdate(
      level,
      level.NextExtent.West,
      minLatitude,
      level.NextExtent.East,
      maxLatitude);
  _updater.Update(context, horizontalUpdate);
}

if (width > 0)
{
  ClipmapUpdate verticalUpdate = new ClipmapUpdate(
      level,
      minLongitude,
      level.NextExtent.South,
      maxLongitude,
      level.NextExtent.North);
  _updater.Update(context, verticalUpdate);
}
```

Listing 13.8. The regions to update are computed from the motion of the level's extent since the last frame.

level extent and it continues to the easternmost post in the next extent of the same clipmap level. If the viewer moved to the west (`deltaX` $<$ 0), the region starts at the next westernmost post and continues just shy of the previous westernmost post. The minimum and maximum latitudes that need to be updated are computed similarly from the `deltaY`, where `deltaY` < 0 is movement to the south and `deltaY` > 0 is movement to the north.

If the viewer moved in only one direction, east/west or north/south, the update region is a simple rectangle. If the viewer moved in both directions since the last render frame, the texels that need to be updated form an L shape, as shown in Figure 13.9, and we update the L using two rectangular updates.

Just like `CurrentExtent` and `NextExtent`, the update regions express a range of posts in terms of *terrain coordinates*. Terrain coordinates are the integer-valued coordinates of the post within a particular level of the terrain dataset, numbered sequentially starting from the southwest corner



(a) (b)

Figure 13.9. (a) When the viewer moves diagonally over the course of a single frame, the region that needs to be updated with new posts forms an L shape. (b) Wraparound addressing in both directions avoids the need to copy existing posts.

of the world. This is in contrast with *texture coordinates*, which are the coordinates of the post in the height-map texture for a particular clipmap level. Texture coordinates take into account the current `OriginInTextures` of the clipmap level, as well as the use of toroidal texture addressing.

For a given set of terrain coordinates, it is straightforward to determine the corresponding texture coordinates:

$$\text{texture} = (\text{textureOrigin} + (\text{terrain} - \text{levelOrigin}))\ \%\text{clipmapSize},$$

where *terrain* is the terrain coordinates of the post; *levelOrigin* is the coordinates of the southwest corner of the clipmap level expressed in terrain coordinates; *textureOrigin* is `OriginInTextures`, the location within the texture that corresponds to the *levelOrigin*; and *clipmapSize* is the length of one side of the clipmap height texture.

Of course, if the terrain coordinates of a post are outside the extent of the clipmap level, the post's texture coordinates are undefined.

13.5.3 Updating Heights

We use GPU rasterization to update a level's height-map texture. We attach the level's height-map texture as the color attachment of a framebuffer and then render a quad. This presents a problem, however.

Due to the `OriginInTextures`, it is likely that the clipmap updates determined in the previous section wrap around the height-map texture. In other words, the northeasternmost post in the update region will have texture coordinates less than the texture coordinates of the southwesternmost post. Effectively, it appears that we need to use toroidal addressing when writing to the framebuffer. This is a problem because framebuffers don't support toroidal addressing.

We work around this problem by using OpenGlobe's ClipmapUpdater.`SplitUpdateToAvoidWrapping` method to split an update into between one and four simple, rectangular, nonwrapping updates.

We're not done splitting the update yet, however. We also need to split the update based on the distinct tiles that are the sources of the height data. In OpenGlobe, this additional splitting is performed by RasterTerrainLevel.`GetTilesInExtent`. Given the extent of an update in terrain coordinates, it returns a list of regions of tiles that compose the update region. The union of all the tile regions is equivalent to the update region.

Finally, we're ready to instruct the GPU to do the height update. The height-update vertex shader is shown in Listing 13.9.

The input `position` vertex attribute comes from a simple unit quad and has one of four possible values: $(0.0, 0.0)$, $(1.0, 0.0)$, $(0.0, 1.0)$, and $(1.0, 1.0)$.

The uniform `u_updateSize` is the width and height of the region to be updated. Multiplying it by the `position` scales the unit quad up to the actual dimensions of the region.

The uniform `u_destinationOffset` is the offset in the clipmap level's height texture to which to write the update. Adding it to the scaled position locates the update within the destination texture.

```
in vec2 position;
out vec2 fsTextureCoordinates;

uniform mat4 og_viewportOrthographicMatrix;
uniform vec2 u_destinationOffset;
uniform vec2 u_updateSize;
uniform vec2 u_sourceOrigin;

void main()
{
  vec2 scaledPosition = position * u_updateSize;
  gl_Position = og_viewportOrthographicMatrix *
                vec4(scaledPosition + u_destinationOffset,
                     0.0, 1.0);
  fsTextureCoordinates = scaledPosition + u_sourceOrigin;
}
```

Listing 13.9. The vertex shader used to update heights in a clipmap.

```
in vec2 fsTextureCoordinates;
out float heightOutput;
uniform sampler2DRect u_tileHeightMap;

void main()
{
  heightOutput =
      texture(u_tileHeightMap, fsTextureCoordinates).r;
}
```

Listing 13.10. The fragment shader used to update heights in a clipmap.

The uniform `u_sourceOrigin` is the origin of the update in the source tile's height-map texture. Adding it to the scaled position locates the source of the update data.

Kevin Says

Don't forget to set the context's viewport to cover the entire height-map texture, from $(0, 0)$ to $(\text{width}, \text{height})$, and then restore it after rendering. Also note that, in this case, we don't need to offset `u_destinationOffset` or `u_sourceOrigin` by 0.5 in order to line up texels. The fragment shader will be invoked based on the centers of fragments, which are already aligned with the centers of tile height-map texels.

The corresponding fragment shader, shown in Listing 13.10, is as simple as can be. It simply writes the height sampled from the source texture to the color attachment associated with `heightOutput`. In other words, it copies a single value from one texture to another.

13.5.4 Updating Normals

After a level's height-map texture is completely updated, we're ready to update the normal map. The approach is basically the same: we render a quad to a framebuffer with the normal map configured as its color attachment. We compute the normals in the fragment shader using central differencing (see Section 11.3.2). The source texture, the one used to texture the quad, is the level's height map.

The vertex shader for updating normals, shown in Listing 13.11, is very similar to the one used to update heights. Unlike the height-update shader, however, this shader only has one origin, which is used to determine both the destination vertex coordinates and the source texture coordinates because normals are computed from the heights around the same

```
in vec2 position;
out vec2 fsTextureCoordinates;

uniform mat4 og_viewportOrthographicMatrix;
uniform vec2 u_updateSize;
uniform vec2 u_origin;
uniform vec2 u_oneOverHeightMapSize;

void main()
{
  vec2 coordinates = position * u_updateSize + u_origin;
  gl_Position = og_viewportOrthographicMatrix *
                vec4(coordinates, 0.0, 1.0);
  fsTextureCoordinates = coordinates * u_oneOverHeightMapSize;
}
```

Listing 13.11. The vertex shader used to update normals in a clipmap.

location. Another difference is that the texture coordinates are normalized to the range $[0, 1]$. This is necessary because the height-map texture is configured to use sampler2D instead of sampler2DRect.

The corresponding fragment shader is shown in Listing 13.12.

```
in vec2 fsTextureCoordinates;
out vec3 normalOutput;

uniform float u_heightExaggeration;
uniform float u_postDelta;
uniform vec2 u_oneOverHeightMapSize;
uniform sampler2D u_heightMap;

void main()
{
  float top = texture(
      u_heightMap,
      fsTextureCoordinates +
      vec2(0.0, u_oneOverHeightMapSize.y)).r;
  float bottom = texture(
      u_heightMap,
      fsTextureCoordinates +
      vec2(0.0, -u_oneOverHeightMapSize.y)).r;
  float left = texture(
      u_heightMap,
      fsTextureCoordinates +
      vec2(-u_oneOverHeightMapSize.x, 0.0)).r;
  float right = texture(
      u_heightMap,
      fsTextureCoordinates +
      vec2(u_oneOverHeightMapSize.x, 0.0)).r;

  vec2 xy = vec2(left - right, bottom - top) *
            u_heightExaggeration;

  normalOutput = vec3(xy, 2.0 * u_postDelta);
}
```

Listing 13.12. The fragment shader used to update heights in a clipmap.

The fragment shader computes the normal by sampling from the height map four times to obtain two partial derivatives by finite differencing and then taking their cross product. This is essentially the central-differencing technique used in Section 11.3.2. In order to compute an accurate normal, we need to account for the world-space distance between posts in the height map as well as the height exaggeration in use, so those quantities are passed to the fragment shader as `u_postDelta` and `u_heightExaggeration`, respectively.

Section 11.3.1 mentions a problem that can occur when computing normals by finite differencing, as we do in this fragment shader. At the topmost row of the height map, there is no texel available above the central one. Similarly, at the rightmost column there is no texel available to the right, at the bottom there is no texel available below, and at the left there is no texel to the left. With a sampling mode that clamps texture coordinates, the terrain at the edges will appear to be flat, creating a noticeable artifact.

Fortunately, we don't actually care very much about the accuracy of the normals at the edges of each clipmap level. Recall from Section 13.4 that the normals near the edges of each clipmap level are blended with the normals from the next-coarser clipmap level. In fact, the outer perimeter vertices all have an `alpha` value of zero, meaning that the normal is determined entirely from the coarser normal map. While some fragments just inside that outer perimeter will have an `alpha` value greater than zero, causing blending with the incorrect normal, the weighting will be so small as to be unnoticeable.

Care must be taken, however, when the viewer moves and the normal map is updated accordingly. At that time, harmless, incorrect normals at the perimeter effectively move toward the interior. Eventually, as the viewer continues to move, 100% weight will be given to the incorrect normal, leading to obvious artifacts.

The solution to this problem that we employ in OpenGlobe is to compute normals over a region that is slightly larger than the one that received updated heights. Adding a one-post border around the update region does the trick. Normals in bordering positions that were potentially computed without the benefit of correct height data are recomputed once the height data are available.

13.5.5 Multithreaded, Out-of-Core Update

Incremental clipmap updates are pretty fast. Assuming a height-map tile is already in GPU memory, copying a subset of it to the clipmap level's height texture and computing normals is fast enough that we can do it during the course of rendering.

But what if the tile is *not* in GPU memory already? Maybe it's in system memory and we need to stream it over the system bus to the GPU. That's going to take a little longer. Worse, maybe it's stored on disk and not even in system memory, yet. Initiating a read of the tile from disk in the render thread and then waiting for it to finish is definitely going to cause a noticeable stutter in the rendering. If you think it can't get any worse—what if that tile can only be found on a distant network server? It might take several seconds to download it, load it into memory, and send it to the GPU. In the meantime, the rendering thread better not be sitting there waiting for it.

For smooth rendering, we must, at a minimum, move the loading of new tiles into system memory off of the rendering thread and on to a worker thread.

In OpenGlobe, we take it one step further. The worker thread, in addition to loading the tile from disk or a network server, also creates a Texture2D and fills it with the tile's height data. This way, the tile's data are transferred from system memory to GPU memory under the control of the worker thread as well. Only when the tile is completely ready to go is it used by the render thread to update clipmap levels.

In effect, tiles cache height data in GPU memory so that it can be quickly used to update the clipmap. This is one part of a complete cache hierarchy for clipmap updates; additional tiles may be cached in system memory, on disk, or on a nearby network server.

Can even more of the update process be moved to a worker thread? How would we update a level's height map in a worker thread while it is simultaneously being used to render?

○○○○ Question

Our multithreaded tile-loading strategy follows the outline presented in Section 10.4.3. A background thread with its own OpenGL context is responsible for loading the tiles and creating Texture2D instances with their contents.

The threads communicate using two message queues: a request queue and a done queue. The messages placed on both queues are instances of the TileLoadRequest class shown in Listing 13.13.

Initially, when a tile load is requested, the `Texture` field is *null*. The other two fields identify the tile that needs to be loaded and the clipmap level that needs it.

The `MessageReceived` handler for the request queue is straightforward. It creates a Texture2D instance with the dimensions of the tile and fills it

```
private class TileLoadRequest
{
  public ClipmapLevel Level;
  public RasterTerrainTile Tile;
  public Texture2D Texture;
}
```

Listing 13.13. The class used to send a tile-load request to the background tile-loading thread. The same class is used to return the tile once it has been loaded.

with the height data of the tile using a WritePixelBuffer, loading it from disk or a network connection if necessary. In OpenGlobe, we create the texture using TextureFormat.`Red32f`, so a 32-bit floating-point value is used to represent each height. If the terrain data were stored in a compressed form, they would be decompressed here as well.

Once the texture is created, we pass the TileLoadRequest instance, now with a nonnull `Texture` field, back to the render thread on the done queue. However, before we do that, we need to synchronize the two GL contexts, as explained in Section 10.4.3, by creating a Fence and waiting on it. Without this fence, the render thread could start rendering from the texture before it has been populated with height data.

Try This ○○○◯

OpenGlobe's ClipmapUpdater waits on the fence in the worker thread with a call to `ClientWait` with an infinite time out. While it's waiting, the worker thread does not do any useful work. Modify it to periodically poll the fence with a time out of zero and do other work, such as loading additional tiles, while waiting.

The render thread adds requests to the *request* queue in the course of updating the clipmap, as shown in Listing 13.14.

First, we check to see if the tile we need to update a region of the clipmap is already loaded. If it is, we render it to the level's height-map texture as explained in Section 13.5.3.

If the tile is not yet loaded, and it hasn't been added to the request queue yet either, we add it to the request queue. We quickly identify tiles that are on the request queue because those tiles exist in the `_loadedTiles` hashtable with a null Texture2D instance. For that reason, the first task of `RequestTileLoad` is to add the tile to `_loadedTiles`. Then, it creates a TileLoad Request and posts it to the request queue.

Next, whether or not the tile was already on the request queue, we update the tile's region in the clipmap with data upsampled from the next-

```
Dictionary<RasterTerrainTileIdentifier, Texture2D> _loadedTiles =
    // ...
Context context = // ...
ClipmapLevel level = // ...
RasterTerrainTileRegion region = // ...
Texture2D tileTexture;
bool loadingOrLoaded =
    _loadedTiles.TryGetValue(region.Tile.Identifier,
                             out tileTexture);
if (loadingOrLoaded && tileTexture != null)
{
  RenderTileToLevelHeightTexture(context, level,
                                 region, tileTexture);
}
else
{
  if (!loadingOrLoaded)
  {
    RequestTileLoad(level, region.Tile);
  }
  UpsampleTileData(context, level, region);
}
```

Listing 13.14. Updating a clipmap level's height texture. If a required tile is not loaded, it is added to the request queue, and the missing data are upsampled from the next coarser clipmap level.

coarser clipmap level. Upsampling gives us a reasonable approximation of the terrain detail without waiting for the tile to load. This upsampling process is described in detail in Section 13.5.6.

As mentioned previously, the worker thread posts tiles to the done queue when it finishes loading them. The done queue is processed synchronously in the render thread each render frame. In OpenGlobe, the ClipmapUpdater.`ApplyNewData` method is responsible for processing the done queue and updating the corresponding clipmap level with the new tile data. It is called after the `NextExtent` and `OriginInTextures` for all levels are computed, as described in Section 13.5.3.

When a new tile is available, the corresponding region in the clipmap level, which was previously populated with data upsampled from the next-coarser level, needs to be updated with the new data. Specifically, the region to update is the intersection of the extent of the new tile and the extent of the clipmap level in terrain coordinates, as shown in Figure 13.10. Of course, this region of intersection may not be the same as the region that motivated the loading of this tile in the first place because the viewer may have moved in the meantime. In a particularly unfortunate scenario, the tile won't contribute any posts to the new extent of the clipmap level that is in play when the tile is finally loaded.

If the region of intersection is not empty, it is updated as before. In addition, for nonempty intersection regions, this tile updating process is applied recursively to any of this tile's child tiles that are *not* loaded.

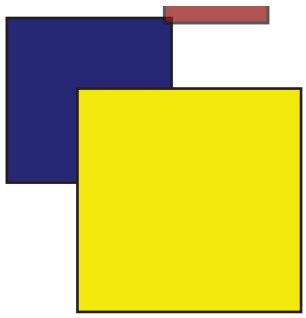


Figure 13.10. When a new tile is loaded, the intersection of the tile's extent (blue) with the clipmap extent (yellow) defines the region that needs to be updated (red).

This recursive application is necessary because, when the child tiles are not loaded, the data used to populate the clipmap in the regions covered by the child tiles are upsampled from this tile. Previously, this tile's data were upsampled from its parent tile. Now that this tile has better data, the children should be upsampled from the better data. Loaded tiles do not need their data reapplied, however, because either (a) the tile was loaded when the clipmap was last updated, so the clipmap already reflects the tile data, or (b) the tile is in the done queue and will be (or already has been) applied to the clipmap.

13.5.6 Upsampling

When a tile needed to update a clipmap level is not yet in GPU memory, we upsample that data from the next-coarser clipmap level. Upsampling is the process of predicting fine detail from a coarse representation. The upsampled heights will not match the real heights, of course, but rendering with imperfect heights is better than the alternatives, such as waiting for the tile to be loaded, rendering the tile as if all its heights were zero, or rendering nothing in the area of the tile.

In order to successfully upsample a subset of one clipmap level from the next-coarser clipmap level, it's important that the next-coarser clipmap level already be updated with correct height data. For that reason, we always update clipmap levels in coarse-to-fine order.

Consider the situation where we're updating the clipmap levels for the very first time, and only the tiles for clipmap level 0 are in GPU memory. Clipmap level 0 is updated as normal, as described in Section 13.5.3. Then, clipmap level 1 is to be updated. Its tiles are not yet resident in GPU memory, though, so instead we upsample the heights in clipmap level 0 and use them to fill clipmap level 1.

The tiles for clipmap level 2 are also not yet resident in GPU memory. So we upsample the heights in clipmap level 1. The heights in clipmap level 1, of course, were themselves upsampled from clipmap level 0. So effectively we upsample the level 0 tiles twice. This process will continue until all levels have been filled with our best approximation of their actual heights.

So how does upsampling actually work? Just as we've done previously with updating the height map and computing normals, we'll use the GPU to do the upsampling by rendering a quad to a framebuffer. In this case, the quad will be textured with the coarser height-map texture, and the framebuffer's color attachment will be the finer height-map texture. In other words, we're going to write heights into the finer height map by passing the coarser one through a fragment shader. The vertex shader for this scheme is shown in Listing 13.15.

This vertex shader is very similar to the one in Listing 13.11 that was used to compute normals. One key difference is that the position to read from in the source texture, the coarser height map, is not the same as the position to write to in the destination texture, the finer height map. Not only do the two textures have different origins but they also have different scales. Specifically, the coarser texture has double the real-world scale of the finer texture. The vertex shader takes all of these factors into account.

The fragment shader is more interesting because there are a number of reasonable things it could do. Perhaps the simplest approach is to just copy the nearest post in the coarse height map into the fine height map. A better approach is to linearly interpolate the fine heights from the coarse ones.

```
in vec2 position;
out vec2 fsTextureCoordinates;
uniform mat4 og_viewportOrthographicMatrix;
uniform vec2 u_sourceOrigin;
uniform vec2 u_updateSize;
uniform vec2 u_destinationOffset;
uniform vec2 u_oneOverHeightMapSize;
void main()
{
  vec2 scaledPosition = position * u_updateSize;
  gl_Position = og_viewportOrthographicMatrix *
                vec4(scaledPosition + u_destinationOffset,
                     0.0, 1.0);
  fsTextureCoordinates =
      (scaledPosition * 0.5 + u_sourceOrigin) *
      u_oneOverHeightMapSize;
}
```

Listing 13.15. The vertex shader for upsampling a clipmap level from the next coarser level.

```
in vec2 fsTextureCoordinates;
out float heightOutput;

uniform sampler2D og_coarseHeightMap;

void main()
{
  heightOutput = texture(og_coarseHeightMap,
                         fsTextureCoordinates).r;
}
```

Listing 13.16. A very simple fragment shader to upsample finer heights from coarser ones.

In fact, these two approaches can be implemented using the very same simple fragment shader, shown in Listing 13.16. The only difference between them is whether the sampler used to read the fine height map is `NearestClamp` or `LinearClamp`.

When this fragment shader is paired with a sampler that performs linear interpolation, the results are pretty good. Much more sophisticated fragment shaders are possible, however.

One possibility is to use cubic convolution. Cubic convolution over a height map involves computing five 1D cubic convolutions, using a total of 16 samples from the coarse height map [83].

Asirvatham and Hoppe recommend upsampling using the tensor product version of the well-known four-point subdivision curve interpolant and state that it has the desirable property of being C^1 smooth [9].

13.5.7 Replacement and Prefetching

Section 13.5.5 discussed how new height-map tiles are brought into GPU memory so that they can be used to update clipmap levels. But when are tiles removed from GPU memory?

Tiles need to be removed from GPU memory when the memory they occupy is needed for other, more important, purposes. For example, in an extreme case, a terrain tile halfway around the world is removed in order to make room for a terrain tile right beneath the viewer's feet. The policy for choosing which tiles to remove is known as a replacement policy, and several generalized replacement policies for terrain rendering are described in Section 12.3.3.

The arrangement of clipmap levels into concentric squares suggests a simple and effective replacement policy: the first tiles to be replaced are the ones that are farthest from the viewer.

Actually, we can do a bit better than this. Remember that the current height map for all clipmap levels is always stored in GPU memory, and we only load tiles into GPU memory in order to use them to update the

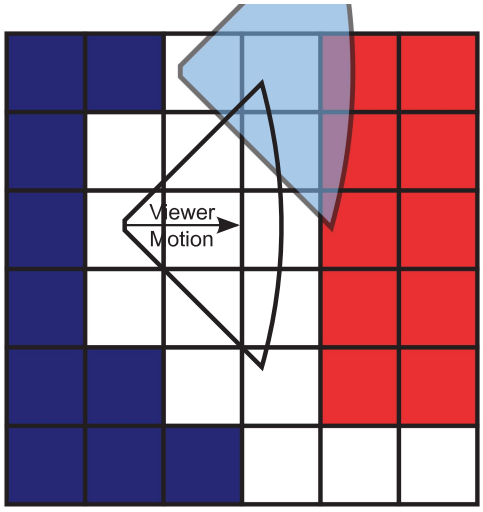


Figure 13.11. Viewer motion can be taken into account when determining replacement and prefetching of tiles. Tiles ahead of the viewer (red) are more likely to be useful in the near term than are tiles in the viewer's wake (blue).

clipmap levels. Furthermore, clipmap levels are always updated at their edges because interior heights already exist within the clipmap level and are moved via toroidal addressing, as described in Section 13.5.1. This means that the tiles we are most likely to need again soon are in fact the ones that are closest to the edges of the clipmap level, rather than the ones that are closest to the viewer. Of course, this distinction may be too small to worry about, especially if the clipmap size and tile size are similar.

Another possible improvement is to take into account the motion of the viewer, as shown in Figure 13.11. If the viewer has been heading east for the past few rendered frames, tiles to the west of the clipmap level are less likely to be needed again soon than are tiles to the east of the clipmap level.

These same principles can be applied to prefetching as well. The simple reactive tile-loading mechanism shown in Listing 13.14 is replaced with a more proactive process. Given a budget for tile data in GPU memory, we load all of the closest tiles up to that limit, optionally skewing for the motion of the viewer.

Prefetching offers the possibility of eliminating upsampling when viewer motion is relatively slow and is within a relatively confined area. In these cases, it significantly improves the user experience. Of course, in virtual globes in particular, prefetching is unlikely to completely eliminate all upsampling because camera motion is simply too fast and unpredictable.

13.5.8 Compression and Synthesis

Losasso and Hoppe, the inventors of the geometry-clipmapping technique, tested their implementation using a 40-gigabyte dataset covering the conterminous United States with approximately 30 m between posts [105]. While not huge by virtual globe standards, it is a bigger dataset than we would ordinarily expect to be able to fit into memory. Yet, they didn't implement any out-of-core data management techniques, and instead fit the dataset into a mere 355 megabytes. How did they do it?

The answer, in case you haven't already guessed, is that they used compression. Their compression scheme involved two parts, one of which relied on the unique structure of geometry clipmaps.

First, they used a particular, lossy image-compression algorithm called progressive transform coder (PTC) that is able to efficiently decode subsets of an image [108]. This allowed them to store the entire height-map image in system memory and decode portions of it as necessary to send to the GPU as textures. Since then, PTC has been standardized as part of JPEG XR.

Second, they noted that the unique organization of the clipmap levels allowed the height information to be stored more compactly.

Recall from Section 13.5.6 that we update clipmap levels in coarse-to-fine order and that this allows us to predict the heights of a fine clipmap level from the heights of a coarse one. As presented previously, we only do this prediction when the fine data are not available.

Instead, we can always predict the fine detail from the coarse while updating clipmap levels. This way, we don't need the actual heights in order to update a clipmap level; we only need the differences between the actual heights and the heights predicted from the coarser level. These differences, called *residuals*, are much more compressible than the complete height data. In fact, the data that Losasso and Hoppe compressed with PTC were residual data, not height data, and this is how they fit a 40-gigabyte uncompressed dataset into 355 megabytes.

For virtual globe-sized datasets, even these techniques will probably not allow us to completely avoid the need to store data on disk and on network servers. Furthermore, imagery data tend to be much less compressible than height data, so it's unlikely that we can fit an entire high-resolution color map of the United States in memory, even using these techniques. However, each byte that we don't need to use to represent the height data is one less byte that we need to read from slow hard disks or send over slow network connections, which means we can show more detail to our users sooner or preserve I/O bandwidth for other purposes.

The main cost of this approach is the additional time spent in the preprocessing step to compute the residuals and compress them. There is additional cost at runtime, too, because the residuals must be decompressed on

the CPU. However, the additional CPU time necessary for decompression is typically more than made up for by the time saved reading the smaller dataset from disk or a network server.

Interestingly, this residual-based approach offers the possibility that the residuals for more detailed clipmap levels not be stored at all. Instead, the residuals are procedurally synthesized at runtime using fractal techniques. Losasso and Hoppe used uncorrelated Gaussian noise to synthesize fine detail when zoomed in close to the terrain surface. This produces a realistic-looking rough surface when the resolution of the terrain data is exceeded, even though it does not, strictly speaking, reflect the real world.

13.6 Shading

Terrain rendered by geometry clipmapping can be shaded in all of the ways described in Section 11.4. Perhaps the most common way to shade terrain in virtual globe applications, however, is to apply a color map derived from satellite imagery. In this case, the color map is probably even larger than the height map!

Recall from Section 12.2.1 that geometry clipmapping is derived from clipmapping, a similar technique used to manage enormous textures. Thus, it's natural to extend geometry clipmapping to also manage a color map.

If the color map and height map have identical extents, mipmap levels, and tile structures, things couldn't be easier. We simply extend the fragment shader in Listing 13.4 to sample from a color map in addition to the normal map. We use the `alpha` blend parameter from Listing 13.5 to blend between the finer and coarser color maps, just as we did with the normal map. Color maps are accessed toroidally and incrementally updated by rendering quads textured with tiled data. Color-map tiles are loaded into textures on the GPU in lockstep with the height-map tiles.

However, the color and height maps are unlikely to line up quite so nicely. They may come from different suppliers that use different schemes to tile their data. Or, one color-map sample per height might simply be too blurry. They could have significantly different resolutions. In these cases, we need to take steps to ensure that differences are accounted for in the shaders.

First, we select an appropriate size for the color clipmap. It must cover at least the world extent covered by the height clipmap, as shown in Figure 13.12. We compute the world location of the southwest post in the height clipmap and then transform it into texel indices in the color map.

If height-map posts don't line up precisely with color-map texels, the computed indices will not be integers, in which case the south and west

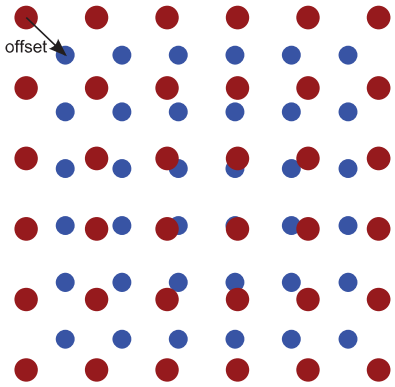


Figure 13.12. The color map used to shade a clipmap level must cover at least the extent covered by the height map. If it's slightly larger, offsets are computed and used in the vertex and fragment shaders to adjust texture coordinates. In this figure, color-map texels are shown as red dots and height-map texels are shown as blue dots.

coordinates are rounded toward the south and west. Similarly, the north and east coordinates are rounded toward the north and east.

The difference between the southwest corner of the color map and the southwest corner of the height map, both expressed in the coordinates of the color map, is passed to the shaders as a uniform. In addition, the ratio of distances between texels in the two maps, in world space, is passed to the shaders. Thus, to compute texture coordinates in the color map, multiply the texture cooordinates in the height map by the ratio between them and then add the offset.

13.7 Geometry Clipmapping on a Globe

Geometry clipmapping as presented thus far only renders terrain extruded from a flat, horizontal plane. The common myth notwithstanding, a flat Earth wasn't considered very accurate even before Christopher Columbus made his famous voyage to the Americas. How can we extend the algorithm to extrude terrain from a sphere or, better yet, an ellipsoid?

Many of the strengths of geometry clipmapping come from its regularity:

- Very little horizontal coordinate data are needed. A small set of vertices can express the horizontal coordinates for an entire terrain.
- The horizontal geometry does not inherently have a specific location in the world; only a translation added in the vertex shader specifies its

absolute position. This translation can easily be expressed relative to the viewer rather than in world coordinates, meaning that the geometry itself is independent of the viewer's location. Thus, the geometry can be incrementally updated as the viewer moves because only the shader uniform changes, while avoiding the jittering that results from using absolute world coordinates in a world the size of Earth.

- The nested structure allows an efficient compression mechanism; only residuals need to be stored at each level.
- Terrain vertices are precisely aligned with height-map texels. If the height map is sampled at a different rate, aliasing artifacts will occur.

The challenge of extending geometry clipmapping to globes is in maintaining these strengths.

13.7.1 Mapping to the Ellipsoid in the Vertex Shader

Perhaps the most direct approach is to map vertices to the ellipsoid in the vertex shader. Consider the common case where the source terrain data are expressed with a WGS84 geographic projection, which means height samples form a regular grid in coordinates of longitude and latitude.

Using this regular grid, we can render the entire world using geometry clipmapping by interpreting the incoming (x, y) position in the vertex shader as longitude and latitude. Initially, this means that the entire world occupies a plane.

```
vec3 GeodeticSurfaceNormal(vec3 geodetic)
{
  float cosLatitude = cos(geodetic.y);

  return vec3(
      cosLatitude * cos(geodetic.x),
      cosLatitude * sin(geodetic.x),
      sin(geodetic.y));
}

vec3 GeodeticToCartesian(vec3 globeRadiiSquared, vec3 geodetic)
{
  vec3 n = GeodeticSurfaceNormal(geodetic);
  vec3 k = globeRadiiSquared * n;
  vec3 g = k * n;
  float gamma = sqrt(g.x + g.y + g.z);

  vec3 rSurface = (k * n) / gamma;
  return rSurface + (geodetic.z * n);
}
```

Listing 13.17. GLSL function for transforming from geographic to Cartesian coordinates.

To map to an ellipsoid, we only need to add code to the vertex shader to transform the geographic vertex position (i.e., longitude, latitude, and height) into Cartesian coordinates. This can be done by porting Ellipsoid.`ToVector3D` from Section 2.3.2 from C# to GLSL, as shown in Listing 13.17.

There are two significant problems with this approach. The first is that it introduces precision problems. When using single-precision floating-point numbers, which is common on today's GPUs, `GeodeticToCartesian` produces a reasonable representation of the Cartesian positions of posts spaced at about 30 m.

For higher-resolution terrain, or for more accurate post placement, a different technique is required on 32-bit GPUs, such as emulating double precision as surveyed by Thall [167]. We've already explored similar techniques in Section 5.4 for emulating subtraction. With the growing availability of 64-bit, double-precision GPU hardware, however, this approach will become increasingly practical in the future.

The second significant problem with this approach, shown in Figure 13.13, is that the height samples, which are equally spaced in geodetic coordinates, are not equally spaced in Cartesian coordinates. In Cartesian space, the samples get closer together as we approach the poles. At the north and south poles, an entire row of height samples corresponds to a single point on the Earth.

This also means that the world extents of the clipmap regions become very small near the poles. In the extreme, imagine a viewer positioned

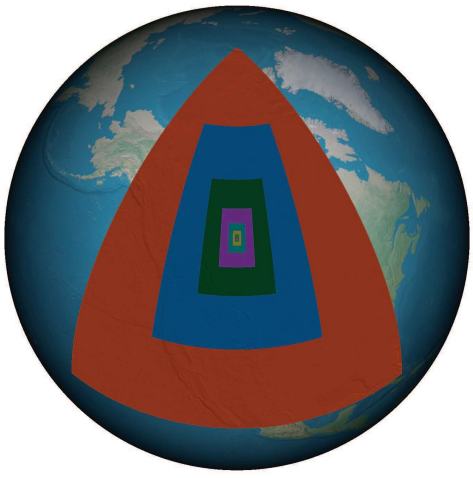


Figure 13.13. As we approach the north or south pole, clipmap vertices get closer together. At the poles, an entire row of height-map texels occupies the same location.

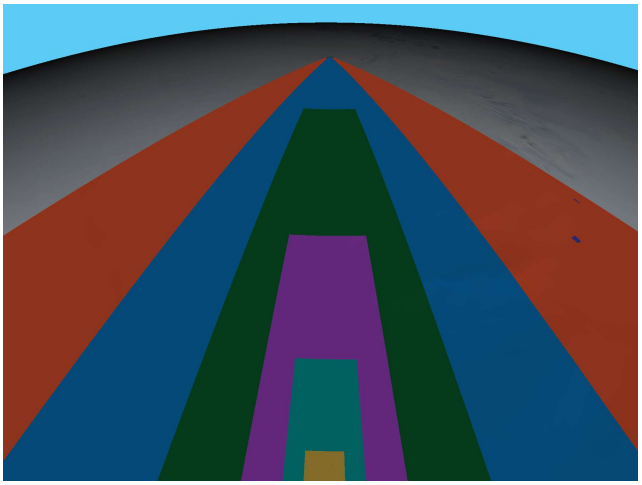


Figure 13.14. A viewer standing near the south pole and looking south can easily see the end of even the coarsest clipmap level.

just south of the north pole and looking north. The terrain will appear to come to a point at the north pole and drop off thereafter. This is shown in Figure 13.14.

In some sense, expressing the source data in a geographic projection implies that the poles are not particularly important, since geographic projections have oversampling problems at the poles. For that reason, it may be acceptable to use a rendering algorithm, such as the one described here, that has its own problems at the poles. If necessary, we can limit how close the clipmap can get to the poles by discarding fragments above or below a threshold latitude in the fragment shader, or by moving vertices outside of the latitude range to the center of the Earth in the vertex shader.

Unfortunately, this technique may be as good as it gets for preserving all of the advantages of geometry clipmapping for terrain on a globe. In a sense, we are sacrificing visual quality near the poles, and possibly elsewhere due to precision problems, in order to preserve all of the benefits of geometry clipmapping. This is a reasonable trade-off for many virtual globes. The remaining techniques to be described, while better than this one in certain ways, sacrifice one or more of these advantages.

13.7.2 Spherical Clipmapping

Spherical clipmapping covers the visible hemisphere of a spherical planet with a series of concentric rings centered around the viewer. Like in conventional, planar geometry clipmapping, the vertices that are displaced

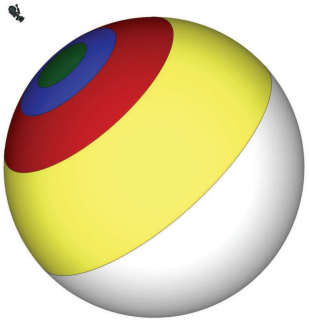


Figure 13.15. In spherical clipmapping, a series of circular, concentric rings are centered around the viewer. These rings are static with respect to the viewer.

by reading from a height map are static with respect to the viewer. The arrangement of clipmap levels around the viewer is shown in Figure 13.15.

The details of implementing spherical clipmapping are described by Clasen and Hege [27]. Vertex data are created from a spherical parameterization. Like in the original geometry-clipmap algorithm, the vertex data do not need to change as the viewer moves. Unlike planar geometry clipmapping, however, spherical-clipmap vertices are not aligned with height-map texels. This leads to the major problem with spherical clipmaps: aliasing.

Consider a simple terrain peak, shown in the cross-section in Figure 13.16. The shape of the resulting terrain changes significantly depending

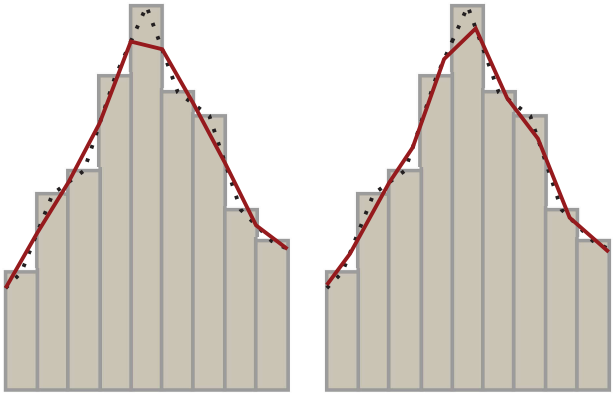


Figure 13.16. Because spherical clipmap vertices are not precisely aligned with height samples, the terrain can appear to change shape as the viewer moves. Two different views of the same peak are shown where different sampling creates a drastically different shape (red lines). The bars represent the height samples and the dashed black lines represent linear interpolation of the height samples.

on how the vertices land on height-map samples. As the viewer moves, the vertices slide across the height map and the rendered geometry changes as a result.

This effect becomes less noticeable as the resolution of the spherical clipmap is increased, and it will become nearly unnoticeable when triangles shrink to approximately the size of a pixel. Unfortunately, rendering such a spherical clipmap is very expensive.

Another problem with spherical clipmaps is that they are, as the name implies, designed to render terrain on a spherical globe. Extending the technique to an ellipsoid model of Earth, such as the WGS84 oblate spheroid, is challenging. For some applications, a spherical globe may be acceptable. Terrain and imagery rendered on a spherical versus WGS84 globe are subjectively identical. When combined with other data, though, the difference can be enormous. For example, a satellite or aircraft positioned with a precise geocentric, Cartesian coordinate system will appear to have an altitude as much as 21.3 km different from its actual altitude.

13.7.3 Coordinate Clipmapping

The most promising approach for mapping geometry clipmaps to a globe, which we call *coordinate clipmapping*, is described by Frühstück [53]. Instead of storing just heights in the vertex texture associated with each clipmap level, complete sets of Cartesian coordinates are stored in a three-channel floating-point texture.

The static clipmap geometry serves only as texture coordinates that are used to retrieve the full xyz-position of the vertex from the texture and to describe how the vertices form triangles. This approach enables an impressive degree of flexibility in how vertices in a clipmap level are arranged. They can represent a mapping of the height map onto a plane, sphere, ellipsoid, or any number of other shapes.

At first glance, this may not appear much different from the conventional rendering approach of storing a triangle mesh in a vertex buffer. In fact, by simply using a vertex buffer, we would avoid the need to do a vertex texture fetch in the vertex shader. There is an important difference, however. By storing our vertex positions in a texture, we can utilize toroidal texture addressing (see Section 13.5.1) to incrementally upload new positions to the GPU as the viewer moves around.

The biggest downside to this approach compared to conventional, planar geometry clipmapping is that the clipmap levels occupy at least three times the memory. In an environment where size is speed, this is a significant disadvantage.

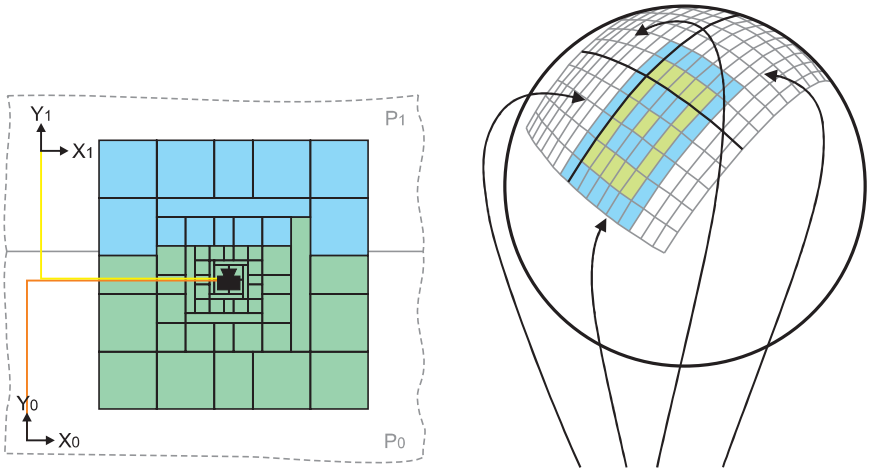
Another concern is the precision with which the vertices can be represented. In order to support incremental updates, the vertices cannot be

defined relative to the viewer. If they were, their positions would need to be updated every time the viewer moved, which would require rewriting the entire texture. If we define the vertices relative to the center of the Earth, however, we run into the same precision problems that we saw when mapping vertices to the ellipsoid in the vertex shader.

One solution is to use a floating origin. Vertices are defined relative to an origin that is near the viewer. As the viewer moves, the origin remains unchanged. At some point, when the viewer is too far from the origin, a new origin is selected and all of the vertices are updated to be relative to the new origin. Double buffering can be used to perform the update over the course of several frames rather than trying to update all clipmap levels in one frame, which would cause a noticeable stutter in the rendering. Microsoft Flight Simulator uses a similar technique [163] as described in Section 5.5.

Frühstück presents a variation of this idea in which the world is divided into regions, each of which has a fixed origin [53]. Depending on the position of the viewer, up to four regions need to be rendered, as shown in Figure 13.17.

Another solution is to store the vertices using simulated double precision and transform them to be relative to the viewer in the vertex shader, as described in Section 5.4. This solution is rather elegant, but it does double



(a) (b)

Figure 13.17. Precision problems in coordinate clipmapping can be addressed by dividing the world into multiple regions, each with their own origin. (a) Rendering clipmaps that overlap two terrain regions. (b) When the viewer is near the corner of a region, up to four regions need to be rendered. (Images courtesy of Anton Malischew.)

again the quantity of per-height data that are needed. Each height sample is represented by three components, x, y, and z, each of which is 8 bytes, for a total of 24 bytes per vertex, compared to 4 bytes per vertex for planar geometry clipmapping. An interesting variation is to use the precision LOD technique described in Section 5.4.2 to only render vertices with double precision when it is necessary. For example, the finest one or two levels are rendered with double precision and the rest are rendered with single precision.

In either approach, longitude, latitude, and height samples are transformed to Cartesian space in double precision on the CPU. When using the GPU RTE approach, the tiles sent to the GPU represent the vertex positions in full precision. When using a floating origin, the tile vertex positions on the GPU are relative to the floating origin as well, which means that they must be refreshed when the floating origin changes.

Coordinate clipmapping represents an effective solution to the problem of applying geometry clipmapping to a globe and offers a great deal of flexibility in exactly where the vertices are placed. In addition, it retains most of the advantages of planar geometry clipmapping. The primary weakness is that it requires that much more data be streamed to the GPU and stored in GPU memory.

13.8 Resources

Losasso and Hoppe's original paper on geometry clipmaps is a great resource [105], as is Asirvatham and Hoppe's article on extending it to make better use of the GPU [9]. The latter is freely available online, along with all of *GPU Gems 2*.

Clasen and Hege describe spherical clipmapping in detail [27]. In his master's thesis, Frühstück provides a detailed explanation of the technique that we've termed coordinate clipmapping [53].

14

Chunked LOD

Chunked LOD is a hierarchical LOD system that breaks an entire terrain into a quadtree of tiles, called *chunks* (see Figure 14.1). The root chunk is a low-detail representation of the entire world. The four child chunks of the root evenly divide the world into four equal-sized areas and provide a higher-detail representation. Each of those chunks then has four child chunks itself, which further divide the world.

Each node in the quadtree is generated in a preprocessing step by simplifying a subset of the overall terrain mesh to achieve a specific level of geometric error. At each lower-detail level of the quadtree, the geometric

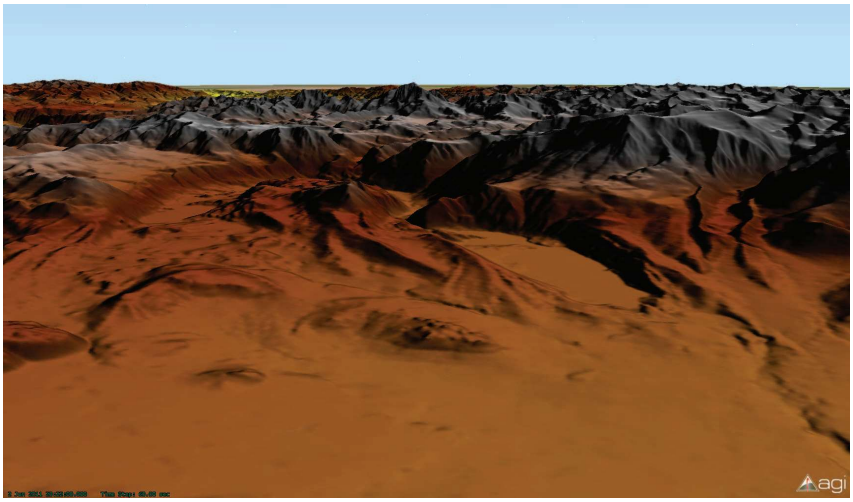


Figure 14.1. A scene near the Sierra Nevada mountains rendered using chunked LOD. Chunked LOD renders massive terrains with precise control over the screen-space error of the terrain and makes good use of the GPU. (Image taken using STK.)

error is double what it is in the next-finer level. This is the generation portion of the LOD algorithm, as described in Section 12.1.

At runtime, chunks are selected for rendering by projecting their geometric error to screen space to compute the screen-space error, in pixels, of the chunk. If the error is too high, its children are visited instead, which is called *refinement*. This is the selection portion of the LOD algorithm. When two chunks of different LODs are adjacent to each other, their edge vertices will not necessarily coincide, leading to cracks between the chunks. A particular concern is how to fill these cracks so that the mesh appears seamless.

When zooming in toward a chunk, there comes a moment where the error estimate for the chunk gets too high, and at that moment, the chunk will subdivide into its children. Similarly when zooming out, four child chunks may be suddenly replaced with their parent chunk. This is likely to create an obvious and objectionable popping artifact. Chunked LOD addresses this problem by gradually morphing between the levels of detail. This is the switching portion of the LOD algorithm.

In many ways, chunked LOD is the direct application of the hierarchical LOD approach to terrain rendering (see Section 12.1.3). It is efficient on modern GPUs because it uses relatively large, static vertex buffers to render each chunk. Only a minimal amount of CPU time is required to determine which chunks to render in a given scene. It also does an excellent job of rendering a scene with guaranteed bounds on the amount of screen-space error introduced by LOD.

Ulrich introduced chunked LOD at SIGGRAPH in 2002 during a course on rendering massive terrains [170]. His approach pulled together various threads of research, plus a number of his own innovations, to create a very practical system for rendering terrain with out-of-core datasets. Even better, he backed it up with working, public domain source code.[1]

Since then, chunked LOD has seen significant success in virtual globes, games, and other applications that need to render massive terrains. Perhaps more surprisingly, considering the significant changes in GPUs since 2002, chunked LOD is still highly relevant today.

Kevin Says ○○◯◯

Chunked LOD has been the basis for the terrain rendering in Insight3D and STK for a number of years now; prior to that, we used ROAM. I suspect that most commercial virtual globes available today use an algorithm similar to chunked LOD for their terrain rendering.

[1] http://tulrich.com/geekstuff/chunklod.html

14.1 Chunks

The process of generating chunks is largely orthogonal to the process of rendering them. In principal, chunks can be created from any input terrain dataset, including terrains with vertical or overhung features. However, the chunks, once created, must have the following characteristics:

- *Rectangular shape.* Each chunk must have a vertex in all four corners of the chunk's extent. In addition, there must be an edge between the chunk's adjacent corners. This guarantees that chunks form a watertight mesh, at least in the horizontal plane. We'll discuss the solution to cracks caused by differences in height between adjacent chunks in Section 14.2.
- *Monotonic geometric error.* We must know the maximum geometric error for each chunk. Geometric error is the maximum distance of any vertex in the full-detail model to the closest corresponding point in the reduced-detail model, and it must be monotonic relative to the chunk level. In other words, rendering the four children of a chunk must always result in *less* error than rendering the chunk itself. The geometric error is computed at the time the chunk is generated.
- *Known bounding volume.* Each chunk must be contained within a known bounding volume. The bounding volume, along with the geometric error, is used to compute the maximum screen-space error of the chunk. Furthermore, for best results, the bounding volume of child chunks should be fully contained within parent chunks. In practice, an AABB or bounding sphere for the chunk is computed while generating the chunks.

As shown in Figure 14.2, each chunk has a vertex buffer indicating the location of the vertices in the chunk and usually other per-vertex information, such as normals and texture coordinates. It also has an index buffer specifying how the vertices are connected to form the terrain surface. Finally, the maximum geometric error and bounding volume for each chunk are known.

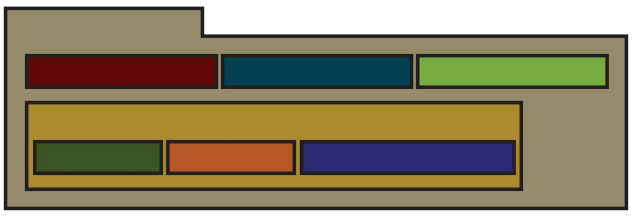


Figure 14.2. The data associated with each chunk.

In Section 14.5, we will show how a chunked LOD quadtree meeting these requirements is generated. First, we will assume that it already exists and show how to go about rendering it.

14.2 Selection

To render a chunked LOD quadtree, we first need to choose a value for our maximum tolerable screen-space error, `tau`. The value of `tau` can be tweaked for faster performance on slower hardware or for greater detail on fast hardware.

We render the chunk quadtree starting at the root, using an algorithm like the one shown in Listing 14.1. For each chunk, the screen-space error of the chunk is calculated from the chunk's maximum geometric error, as described in Section 12.1.4, and compared to `tau`. If the screen-space error of the chunk is under the allowable limit, the chunk is rendered. Otherwise, we refine by recursing on the chunk's four child chunks. This continues until the entire terrain is rendered at the lowest level of detail that meets our screen-space error requirements. Figure 14.3 shows the quadtree nodes rendered and refined in an example frame.

We prefer to render child nodes in front-to-back order to take advantage of GPU depth buffer optimizations. This is surprisingly efficient and easy to do in a quadtree structure like the one used in chunked LOD, as described in Section 12.4.5. It is also useful to perform view-frustum and horizon culling at this stage, as described in Section 12.4.2.

At this point, our entire terrain is rendered at appropriate levels of detail. This simple implementation exhibits two major artifacts, however: cracking and popping.

```
public void Render(ChunkNode node, SceneState sceneState)
{
  if (ScreenSpaceError(node, sceneState) <= tau)
  {
    RenderChunk(node, sceneState);
  }
  else
  {
    foreach (ChunkNode child in node.Children)
    {
      Render(child, sceneState);
    }
  }
}
```

Listing 14.1. Chunks to render are selected based on their screen-space error.

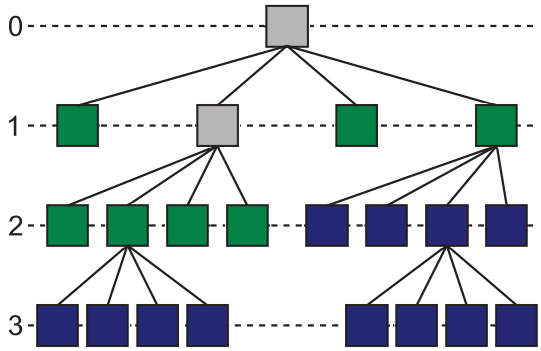


Figure 14.3. In this example, gray quadtree nodes did not meet screen-space error requirements and were refined. Green nodes did meet the error requirements and were rendered. Blue nodes were not visited because an ancestor was rendered.

14.3 Cracks between Chunks

When we render two chunks of different levels of detail next to each other, there is no guarantee that the heights along the adjacent edges are the same. In fact, the two chunks may not even have the same number of vertices along their edges!

These height differences lead to clearly visible cracks between the chunks. How can we fill these cracks? As discussed in Section 12.1.5, this is a common problem with hierarchical LOD algorithms, and there are several approaches to solving it. Ulrich fills the cracks by adding a "skirt" around the perimeter of each chunk [170].

A skirt is a strip of triangles that starts at the perimeter of the chunk and drapes down below it, as shown in Figures 14.4 and 14.5. To hide pinholes in the mesh caused by T-junctions between adjacent chunks, the

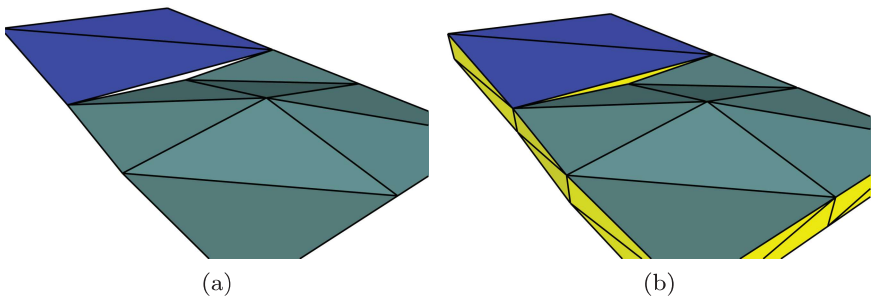
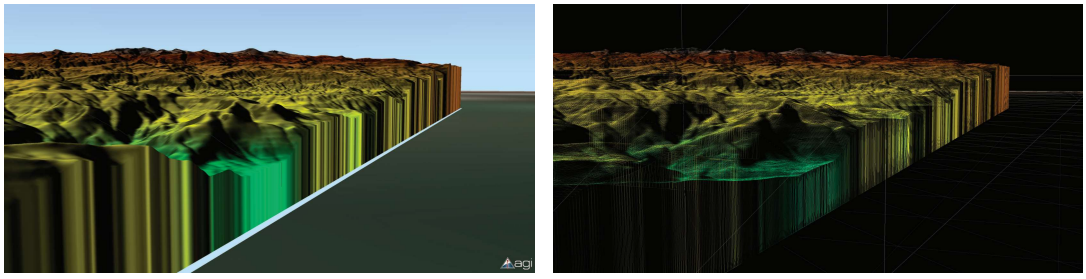


Figure 14.4. (a) Crack. (b) A skirt around the perimeter of each chunk fills the cracks that occur when adjacent chunks have different levels of detail.



(a) (b)

Figure 14.5. The skirt around tiles in a chunked LOD implementation. Notice that the skirts are shaded just like the chunk edges. (Images from STK.)

bottom edge of the skirt is angled slightly away from the center of the chunk. While this doesn't fill the pinholes, it hides them because the user sees the skirt instead of the background through the pinholes.

The length of the skirt is somewhat arbitrary, but it must be long enough to cover the height difference between the vertices in the adjacent chunks. In addition, we want it to be as short as possible to minimize the fill rate for triangles largely hidden under the chunks.

At chunk-generation time, we know the height at a given position at all levels of detail. So, we can precisely compute the length of the skirt and bake it into the chunk's static vertex buffer. Optionally, we can impose a limit on the maximum LOD difference between two adjacent chunks and take this limit into account as well when generating the skirt. At render time, the skirt is rendered along with the chunk, filling the chunks, with no additional draw calls.

14.4 Switching

As implemented so far, our chunked LOD algorithm can select an appropriate LOD based on the desired screen-space error and seamlessly join the chunks with skirts. Combined with an algorithm for bringing chunks into GPU memory as they're needed, as described in Section 14.7, this algorithm can be used to render enormous terrains with high visual fidelity. One problem remains, however.

As the viewer moves, the level of detail of the terrain will change as chunks are refined and merged. The changes will be small. In fact, the positions of individual vertices in screen space are guaranteed not to change by more than `tau` pixels, at least at the center of the screen. In most cases,

the change resulting from an LOD switch will be much smaller than `tau`. If `tau` is a small value, the change might be nearly imperceptible.

The trouble, however, is that many vertices will change all at once as the region "pops" from one LOD to the other. While one vertex would likely be imperceptible, hundreds of vertices popping slightly all at once will be distracting.

In chunked LOD, the popping problem is actually easier to solve than it sounds. The solution is to subtly morph between LODs instead of switching all at once.

At chunk-generation time, we need to compute an extra value for each vertex in a given chunk: the morph delta for the vertex. The morph delta is the difference between the height of the vertex in the chunk and the height at the same (x, y) position in the parent chunk. Of course, the morph delta for the coarsest-detail chunk is zero, reflecting the fact that it has no coarser LOD to which to morph.

Then, at render time, we compute a morph parameter for the entire chunk according to Equation (14.1):

$$\text{morph} = \text{clamp}\left(\frac{2\rho}{\tau} - 1, 0, 1\right), \tag{14.1}$$

where ρ is the maximum screen-space error of the chunk as computed in Equation (12.1), and τ is the maximum allowed screen-space error before the chunk is split into its four child chunks, or `tau` in previous code examples.

This equation produces $morph = 0$ at the distance that the chunk's parent is split to produce this chunk. It produces $morph = 1$ at the distance that the chunk itself is split into its four child chunks. As the viewer moves

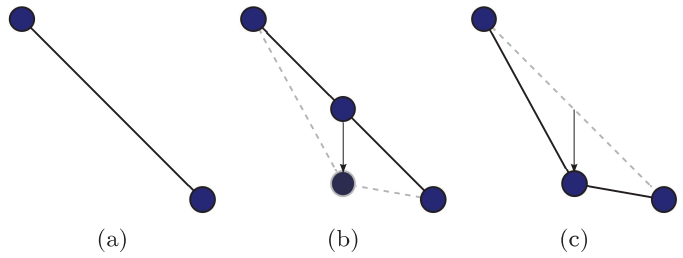


Figure 14.6. A side view of a small segment of terrain. Chunked LOD avoids vertex "pops" by smoothly morphing between LODs in the vertex shader. (a) A segment of a low-detail chunk. (b) When the chunk is first refined, a new vertex is added exactly where it appeared to be in the low-detail chunk, so no change is visible to the user. (c) As the viewer moves, the vertex smoothly morphs to its true position.

```
in vec3 position;
in float morphDelta;
// ...

uniform float u_morph;
// ...

void main()
{
  // ...

  float fineHeight = position.z;
  float coarseHeight = position.z + morphDelta;
  float height = mix(coarseHeight, fineHeight, u_morph);

  vec3 morphedPosition = vec3(position.xy, height);

  // ...
}
```

Listing 14.2. Chunks are smoothly morphed from one LOD to another with a simple modification to the vertex shader.

between those two distances, the value of *morph* smoothly moves between $morph = 0$ and $morph = 1$. This is shown in Figure 14.6.

This method of computing *morph* results in the morph occurring over the entire range of screen-space error for which the chunk is displayed, but other approaches are possible. It may be more desirable to quickly morph to full detail over a fraction of the chunk's screen-space error range. This will get more detail into the scene sooner, at the cost of a potentially more visible transition.

Another approach is to morph based on wall clock time instead of screen-space error. For example, $morph = 0$ when the chunk's parent is first refined and smoothly ramps to $morph = 1$ over the course of the next two seconds.

The morph itself is done in the vertex shader. The morph parameter, *morph*, is passed to the shader as a uniform. The morph delta for each vertex is passed to the shader as a vertex attribute. The vertex position is then computed in the shader as shown in Listing 14.2.

This morph based on the viewer position makes the LOD switch completely undetectable to the user.

14.5 Generation

While chunked LOD can be used to render terrain created from virtually any data source, Ulrich focused on generating chunks from height maps. Given that height maps are by far the most common terrain representation in virtual globe applications, we will follow that approach as well.

In addition to creating individual quadtree chunks with the spatial properties described previously, all the chunks at a given level are decimated to the same geometric error relative to the original height map. Specifically, given a level l, the geometric error, ϵ, at the next-coarser level, $l-1$, is given by Equation (14.2) for l greater than zero:

$$\epsilon\,(l-1) = 2\epsilon\,(l)\,. \tag{14.2}$$

In other words, the geometric error in each level halves while going from coarser to finer levels. The geometric error for the finest level is chosen at the time the chunk tree is generated. A value of 0.5 means that the heights in the finest-detail chunks are allowed to deviate from the original mesh by up to 0.5 height units. Using a nonzero value for ϵ at the finest clipmap level allows the most detailed mesh to be simplified as well in flat or nearly flat areas, saving memory and bus bandwidth. This is an advantage of chunked LOD over geometry clipmapping, which requires uniform tessellation even where it does not provide any benefit.

In order to achieve this very specific level of error, Ulrich implemented an algorithm that is based on work by Lindstrom et al. and Duchaineau et al., and has similarities to the view-independent progressive mesh technique described by Bloom [18, 41, 102].

The algorithm consists of three parts: *updating*, *propagation*, and *meshing*.

14.5.1 Updating

The first part of the chunk-generation algorithm updates each vertex in the input height map with an *activation level*, which is the number of the coarsest level that must include the vertex in order to meet the geometric error requirements.

The input height map, which must have $2^n + 1$ heights on each side, is treated as an implicit binary triangle tree, or *bintree*. At the root, the bintree has two triangles formed by four posts from the height map, as shown in Figure 14.7. Each of these triangles has two child triangles formed by bisecting the triangle's longest edge. Subdivision continues recursively until all height-map posts are covered by triangle vertices. Due to the carefully chosen dimensions of the input height map, the bisecting vertex always falls on a post in the height map.

Updating proceeds recursively over the implicit triangle bintree, starting with the two triangles that form the quad that is the extent of the entire terrain region. This recursive update function is shown in Listing 14.3. The vertices of each triangle are labeled as shown in Figure 14.8.

Computing the activation level for the base vertex of a triangle is straightforward. First, we compute an estimate of the height at the base

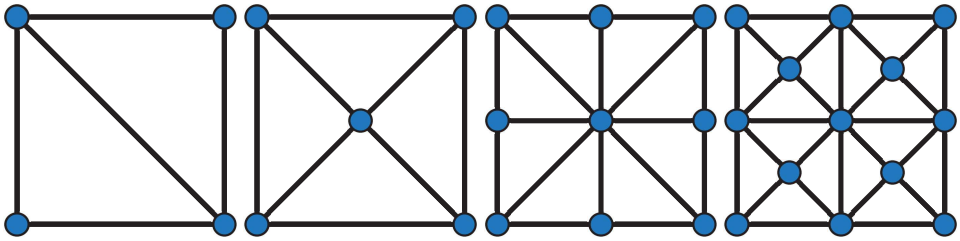


Figure 14.7. The input height map forms an implicit binary triangle tree (bintree). The root consists of two triangles formed by the posts at the four corners of the height map. Each triangle has two children formed by bisecting the triangle's longest edge. The first four levels of the implicit bintree are shown.

vertex by averaging the heights at the left and right vertices of the triangle. The difference between this estimated height and the actual height at the base vertex is the geometric error.

```
void Update(HeightMap heightMap,
            double baseMaxError,
            int baseLevel,
            int apexX, int apexY,
            int rightX, int rightY,
            int leftX, int leftY)
{
  // The base vertex is midway between left and right.
  int dx = leftX - rightX;
  int dy = leftY - rightY;
  if (Math.Abs(dx) <= 1 && Math.Abs(dy) <= 1)
  {
    return;
  }

  int baseX = rightX + (dx / 2);
  int baseY = rightY + (dy / 2);

  // Sample the heights of left, right, and base.
  // Also estimate the height at base by averaging the
  // left and right heights.
  short leftHeight = heightMap.GetHeight(leftX, leftY);
  short rightHeight = heightMap.GetHeight(rightX, rightY);
  short baseHeight = heightMap.GetHeight(baseX, baseY);
  double estimatedHeight = (leftHeight + rightHeight) / 2.0;

  // Compute the difference between the actual and estimated
  // heights at the base.  This is the geometric error.
  double geometricError = Math.Abs(baseHeight - estimatedHeight);

  // If this error is larger than the error allowed at the
  // finest detail level, compute the coarsest detail level
  // that still must include this vertex and update the vertex
  // with this information.
  if (error >= baseMaxError)
  {
    int activationLevel =
```

```
        baseLevel = (int)Math.Round(Math.Log(geometricError /
                                             baseMaxError, 2.0));
    heightMap.Activate(baseX, baseY, activationLevel);
  }

  // Recurse to the triangle formed by base, apex, right.
  Update(heightMap, baseMaxError, baseLevel, baseX, baseY,
         apexX, apexY, rightX, rightY);

  // Recurse to the triangle formed by base, left, apex.
  Update(heightMap, baseMaxError, baseLevel, baseX, baseY,
         leftX, leftY, apexX, apexY);
}
```

Listing 14.3. This recursive function assigns an activation level to each vertex in a height map. The activation level is the coarsest level that must include the vertex in order to meet geometric error requirements.

At first glance, it may seem strange to compute an error metric from one line segment. After all, we're choosing whether or not to include a vertex in a particular LOD, and removing a vertex actually affects all the triangles that share that vertex. Depending on the triangulation of the height map, a single vertex is included in up to eight triangles!

In reality, we're computing the error for a very specific situation. Given an existing mesh, we're computing how much error is introduced if we remove a single edge separating two triangles and fuse the triangles. Thus, the computed error is relative to the previous, more complicated mesh, not relative to the original, highest-detail mesh.

Lindstrom et al. found that this approach resulted in less than 5% of the displacements exceeding the target geometric error when compared against the original mesh [102]. Furthermore, the average geometric error was well below the target geometric error. If this is inadequate for a particular application, a more precise simplification algorithm can be used, such as

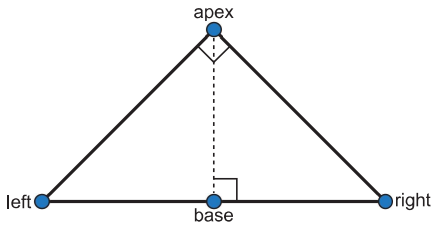


Figure 14.8. In the implicit bintree, triangle vertices are labeled *apex*, *left*, and *right*. The angle at the apex is always 90°, and the angles at *left* and *right* are equal. A fourth vertex, called *base*, bisects the line segment between the *left* and *right* vertices.

one of the algorithms mentioned in Section 12.2.3, at the cost of increased preprocessing time.

Using our rule that each successive level is allowed twice the geometric error of the prior level, we compute the number of levels past the finest level that need to also include this vertex. Finally, we subtract that from the level number of the finest level to find the actual number of the coarsest level that must include this vertex.

14.5.2 Propagation

The second part in the chunk-generation process, shown in Listing 14.4, propagates activation levels between vertices so that the simplified mesh at each level remains watertight.

```
void Propagate(HeightMap heightMap,
               int levels,
               int size)
{
  int increment = 2;
  for (int level = 0; level < levels; ++level)
  {
    for (int chunkCenterY = increment / 2;
         chunkCenterY < size;
         chunkCenterY += increment)
    {
      for (int chunkCenterX = increment / 2;
           chunkCenterX < size;
           chunkCenterX += increment)
      {
        PropagateInChunk(heightMap, level,
                         chunkCenterX, chunkCenterY);
      }
    }
    increment *= 2;
  }
}

void PropagateInChunk(HeightMap heightMap, int level,
                      int centerX, int centerY)
{
  int halfSize = 1 << level;
  int quarterSize = halfSize / 2;

  if (level > 0)
  {
    // Propagate child vertices to edge vertices
    int activationLevel;

    // Northeast child
    activationLevel =
        heightMap.GetActivationLevel(centerX + quarterSize,
                                     centerY - quarterSize);
    heightMap.Activate(centerX + halfSize,
                       centerY, activationLevel);
    heightMap.Activate(centerX,
                       centerY - halfSize, activationLevel);
```

```
    // Northwest child
    activationLevel =
        heightMap.GetActivationLevel(centerX - quarterSize,
                                     centerY - quarterSize);
    heightMap.Activate(centerX,
                       centerY - halfSize, activationLevel);
    heightMap.Activate(centerX - halfSize,
                       centerY, activationLevel);

    // Southwest child
    activationLevel =
        heightMap.GetActivationLevel(centerX - quarterSize,
                                     centerY + quarterSize);
    heightMap.Activate(centerX - halfSize,
                       centerY, activationLevel);
    heightMap.Activate(centerX,
                       centerY + halfSize, activationLevel);

    // Southeast child
    activationLevel =
        heightMap.GetActivationLevel(centerX + quarterSize,
                                     centerY + quarterSize);
    heightMap.Activate(centerX,
                       centerY + halfSize, activationLevel);
    heightMap.Activate(centerX + halfSize,
                       centerY, activationLevel);
  }

  // Propagate edge vertices to center.
  heightMap.Activate(
      centerX, centerY,
      hf.GetActivationLevel(centerX + halfSize, centerY));
  heightMap.Activate(
      centerX, centerY,
      hf.GetActivationLevel(centerX, centerY - halfSize));
  heightMap.Activate(
      centerX, centerY,
      hf.GetActivationLevel(centerX, centerY + halfSize));
  heightMap.Activate(
      centerX, centerY,
      hf.GetActivationLevel(centerX - halfSize, centerY));
}
```

Listing 14.4. This function propagates the activation levels computed by `Update` between vertices in order to maintain a watertight mesh.

To that end, propagation begins on the finest, most detailed level and proceeds up the hierarchy of levels from there. Propagation is completed for all chunks in a given level before proceeding to the next level.

Within a level, propagation proceeds chunk by chunk. The propagation process for a single chunk within a level is shown in Figure 14.9.

First, in all levels except the finest, the activation level of the vertex at the center of each of the four child quads is propagated to the adjacent edges. Then, in all levels including the finest, the activation levels of the four edge vertices are propagated to the center vertex.

When an activation level is propagated to a vertex, the vertex adopts the new activation level only if it is lower than its current activation level.

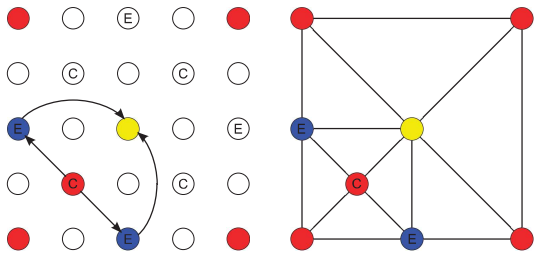


Figure 14.9. (a) Propagation for a 5×5 chunk. Child vertices are marked with C and edge vertices are marked with E. After the update step, five vertices (red) are activated. In order to create a watertight mesh at this level, the activation of the red child vertex is propagated to the blue edges. Then, the edge activation is propagated to the center (yellow). (b) The resulting tessellation.

In other words, propagation can cause a vertex to be included in coarser-detail levels but can never cause it to be removed from levels it already inhabits.

14.5.3 Meshing

In the final part of the chunk-generation process, we create a mesh from the activated vertices in each chunk.

Ulrich uses Lindstrom et al.'s algorithm to tessellate each chunk [102]. This algorithm produces one long triangle strip for the entire chunk but uses a large number of degenerate triangles to do so.

Another way to approach the problem is to tessellate the chunk using a straightforward indexed triangle list instead of a triangle strip. Then, the mesh can be handed off to an algorithm that optimizes it for the vertex cache. Forsyth and Sander et al. describe fast and effective ways to optimize arbitrary meshes for the vertex cache [50, 148].

Question ○○◯◯

This chunk-generation process is by far the most complicated part of a chunked LOD implementation. The preprocessing requirement also imposes limits on the user-supplied data that can be rendered. Is it necessary? Or are today's GPUs fast enough that we could simply use a tiled mipmap of the height map as a quadtree? What are the advantages and disadvantages of such an approach?

14.6 Shading

How can we apply textures to a chunked LOD terrain?

As a first cut, textures are chunked similar to geometry, and we simply associate a static texture with each chunk and map it to the chunk vertices based on their coordinates in the (x, y) plane. In fact, all of the shading techniques discussed in Section 11.4 are applicable to terrain rendered with chunked LOD. An obvious question, though, when applying many of these techniques is, what resolution texture should we use for any given chunk?

In much the same way that we determine which LOD to use for a chunk based on chunk geometric errors projected into screen space, we can also directly determine the texture resolution necessary to achieve a specific relationship between texels and pixels on the screen.

Recall that the screen-space geometric error of a chunk was computed according to Equation (12.1). Similarly, the size, t_s, of a texel projected onto the screen satisfies the inequality in Equation (14.3):

$$t_s \leq \frac{t_g x}{2d_{\min} \tan \frac{\theta}{2}}, \tag{14.3}$$

where t_g is the geometric size of a texel.

By substituting τ for ρ in Equation (12.1) and rearranging it, we can determine the minimum distance, $d_{\min}$, at which the chunk will be displayed. This is Equation (14.4):

$$d_{\min} = \frac{\epsilon x}{2\tau \tan \frac{\theta}{2}}. \tag{14.4}$$

If the distance from the viewer to the chunk is even slightly smaller than this value, the chunk's four children will be rendered instead.

Finally, we substitute $d_{\min}$ into Equation (14.3) and solve for T_g, yielding Equation (14.5):

$$t_g \leq \frac{t_s \epsilon}{\tau}. \tag{14.5}$$

Using this equation, we directly compute the minimum texel size, t_g, necessary to achieve a given screen-space texel error, t_s, for a chunk with geometric error ϵ and maximum screen-space height error τ.

Conveniently, because ϵ doubles with each coarser quadtree level, the required texture resolution halves at each level.

This equation does not, however, take into account the effect of texture stretching on steeply sloped terrain features. The chunk-generation process could take this effect into account as well, if desired, at the cost of a more time-consuming preprocessing step. There is also the possibility that it could lead to an unreasonably large texture being selected for an entire chunk due to a small area of steep geometry. Texture stretching is discussed in more detail in Section 11.4.3.

Kevin Says ○○◯◯

It is, of course, quite reasonable for the maximum screen-space geometric error to be different from the maximum screen-space texel error. In STK and Insight3D, we use a default maximum screen-space geometric error of two pixels and a default maximum screen-space texel error of one pixel. We found that higher texel error resulted in more obvious imagery popping. However, both quantities can be configured by the user.

14.7 Out-of-Core Rendering

With a large terrain, it is not possible to fit all of the chunks into memory at once. Thus, an out-of-core algorithm is required in order to bring chunks into memory as they're needed.

We follow the outline given in Section 12.3, in which a loading thread is responsible for loading chunks that are requested by the rendering thread. This is shown in Listing 14.5.

As before, we render the first chunk that meets the screen-space error requirements, but now we also request residency for the child chunks of each rendered chunk. This acts as a form of prefetching because it increases the likelihood that the child chunks will already be resident when the viewer moves closer to the child chunks. If a chunk is already loading or resident, `RequestResidency` increases its loading priority or reduces its replacement priority, respectively (see Section 12.3.2).

In addition, we fall back on rendering the current chunk, regardless of screen-space error, anytime the current chunk's children are not loaded. Thus, on a cache miss, we render the best data that we have available for the region.

The ChunkNode instances in Listing 14.5 do not necessarily contain the chunk's vertex and index buffers. When `RequestResidency` is called by the rendering thread, the request is posted to the request queue. The loading thread selects chunks to load based on the load-ordering policy and populates ChunkNode instances with the loaded data.

A ChunkNode instance without vertex or index data is called a *skeleton node*. It contains information about the world-space extent and geometric error of the chunk, as well as the parent–child relationships between chunks, but does not contain the geometry for the chunk. The loading thread turns skeleton nodes into *loaded nodes*. It can also turn a loaded node back into a skeleton node if the node is selected for replacement according to the replacement policy (see Section 12.3.3).

```
public void Render(ChunkNode node, SceneState sceneState)
{
  RequestResidency(node);

  if (!node.AllChildChunksResident ||
      ScreenSpaceError(node, sceneState) <= tau)
  {
    RenderChunk(node, sceneState);
    foreach (ChunkNode child in node.Children)
    {
      RequestResidency(child);
    }
  }
  else
  {
    foreach (ChunkNode child in node.Children)
    {
      Render(child, sceneState);
    }
  }
}
```

Listing 14.5. The chunked LOD rendering algorithm is extended to support out-of-core rendering.

For a large terrain, there can easily be millions of nodes, so keeping all of them in memory, even just as skeletons, is prohibitive. Instead, nodes should only be kept in memory if they were recently rendered, if they are queued for loading, or if they are ancestors of loading or rendered nodes. In a garbage-collected language like C#, each node stores a WeakReference to each of its child nodes. A WeakReference does not prevent the object from being garbage collected and is automatically set to null when that happens.

Loaded and loading nodes are prevented from being garbage collected because the replacement and request queues hold normal (nonweak) references to them. In addition, nodes hold normal references to their parent nodes so that the parents are not garbage collected if the children are loading or loaded. Thus, skeleton nodes are garbage collected eventually if they don't have loaded or loading children. Since skeleton nodes, unlike loaded nodes, do not have any unmanaged resources, it is not necessary to explicitly dispose them.

In a language with explicit memory management, like C++, normal references are maintained by reference counting. To simulate the weak references from parent to child nodes, an uncounted pointer is used that is set to null by the child node's destructor.

Most chunked LOD implementations enforce the rule that if a chunk is loaded, all ancestor chunks all the way up to the root have their geometry resident as well. This is important when the viewer moves away from a chunk, causing its parent to be rendered instead of it and its three

neighbors, and it is the reason for the call to `RequestResidency` at the top of Listing 14.5.

If the parent is not loaded, we have two options, neither of them very good. We can show the first ancestor that *is* loaded, resulting in changing the LOD by two or more levels at once and certainly creating objectionable popping. Or, we can continue to show the higher-detail child chunks until the parent loads. This second option leads to a potentially substantial increase in rendering load and can cause aliasing artifacts as well if the higher LOD has multiple triangles per pixel on the screen in the new view.

14.8 Chunked LOD on a Globe

Perhaps surprisingly, the chunked LOD algorithm presented so far can be used to render terrain extruded from an ellipsoid with few modifications. If the vertices in each chunk are already mapped to the ellipsoid, and the bounding volumes and geometric errors of each chunk reflect this, the core rendering algorithm is unchanged from what we've presented so far.

In fact, the only change we need to make is in morphing between levels of detail. When extruding from a plane, we simply maintain a morph delta per vertex, which is the difference in height between the coarser and the current LOD. When extruding from an ellipsoid, however, the up direction is not so easy to determine. Instead, we provide the corresponding position of the vertex in the next-coarser LOD and smoothly blend with that, as shown in Listing 14.6.

How do we create ellipsoid-mapped chunks during the chunk-generation process? The answer depends on the source terrain data, but in most cases, it's fairly straightforward.

```
in vec3 position;
in vec3 coarserPosition;
// ...

uniform float u_morph;
// ...

void main()
{
  // ...

  vec3 morphedPosition = mix(coarserPosition, position, u_morph);

  // ...
}
```

Listing 14.6. Chunks on an ellipsoid are smoothly morphed from one LOD to another given the corresponding position of the vertex in the coarser LOD mesh.

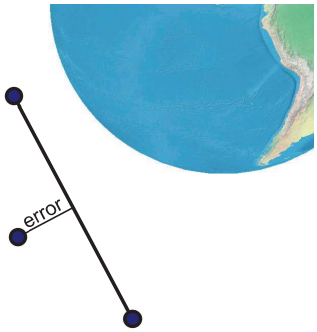


Figure 14.10. Two vertices have equal height but are far apart on the globe. Adding a vertex in the middle vastly improves the terrain's conformance to the globe, so this vertex should be active. If ellipsoid mapping is not considered when computing error, this vertex instead appears to have little impact on the error and is not activated.

For the common case where the source data is a height map in the geographic projection, the $(longitude, latitude, height)$ of each post is transformed to WGS84 Cartesian coordinates, as shown in Section 2.3. The transformed position is used for determining vertex activation during chunk generation so that large regions of equal height, such as oceans, maintain their curved appearance, as shown in Figure 14.10. In addition, the transformed coordinates are used to compute the bounding volume for the chunk.

Because a geographic projection is oversampled at the poles, chunks created in this way will be oversampled there as well. The mesh-simplification process during chunk generation goes a long way toward mitigating this, however. While a region of the input height map might have hundreds of coincident points at the poles, mesh simplification will eliminate most of these.

Care must be taken to ensure that sufficient numerical precision is preserved when mapping posts to the ellipsoid. Storing WGS84 Cartesian vertex positions as single-precision floating-point numbers will likely lead to jittering, as explained in Chapter 5. Any of the solutions explained there will address the precision problem for chunked LOD, but rendering relative to center is usually the best solution.

14.9 Chunked LOD Compared to Geometry Clipmapping

Chunked LOD and geometry clipmapping take two very different approaches to terrain LOD. The strengths and weaknesses of the two approaches

	Geometry Clipmapping	Chunked LOD
Preprocessing	Minimal. A mipmapped height map is all that is required.	Extensive. The terrain mesh is simplified to create chunks, and a bounding volume and error metric must be computed for each chunk.
Mesh Flexibility	None. The mesh must be a regular grid, and therefore, vertical and overhanging features are not possible.	Good. Chunks are irregular meshes and the topology is not important as long as a bounding volume and error metric can be determined.
Triangle Count	High. The regular grid structure burns a lot of triangles, including in areas where they provide no benefit. The triangles are pushed to the GPU very efficiently, however.	Lower. Mesh simplification eliminates unnecessary triangles.
Ellipsoid Mapping	Challenging. Solutions that preserve all of the advantages of the algorithm probably require 64-bit GPUs to be practical. Other solutions lose some of the advantages.	Straightforward. Vertices are mapped to the ellipsoid during chunk generation, and precision problems are managed using RTC or RTE techniques.
Error Control	Poor. The horizontal size of triangles can be controlled by changing the clipmap size, but extreme vertical features can make triangles arbitrarily large.	Excellent. Chunks are selected to achieve a maximum screen-space error, in pixels.
Frame-Rate Consistency	Excellent. Scenes have the same number of triangles regardless of the terrain features, so the frame rate is steady.	Poor. Because the screen-space error guides rendering, bumpy terrain will cause more high-detail chunks to be rendered and the frame rate will be lower than when rendering flatter terrain.
Mesh Continuity	Excellent. Blending between levels creates a smooth and watertight mesh.	Poor. Cracks and T-junctions are hidden rather than eliminated.
Terrain Data Size	Small. Only heights need to be stored, and the regular structure allows for impressive levels of compression. Some techniques for mapping the terrain to an ellipsoid will require that more data be stored, however.	Large. Full xyz-coordinates need to be stored, plus morph targets.
Legacy Hardware Support	Poor. GPU must be able to do efficient vertex texture fetch.	Good. Triangle throughput is important, but even the fixed-function pipeline can be used to render chunks.

Table 14.1. Comparing geometry clipmapping and chunked LOD.

are summarized in Table 14.1. In most cases, a virtual globe developer will not choose one of these algorithms and run with it, but will instead use these algorithms as starting points for developing their own, perhaps incorporating aspects of each.

14.10 Resources

Ulrich's original paper on chunked LOD is a good place to go for more information [170]. His website[2] also has prechunked test data and full source code to a public domain chunk-generation and rendering implementation.

[2] http://tulrich.com/geekstuff/chunklod.html

Implementing a Message Queue

Section 10.3.1 introduced `OpenGlobe.Core.MessageQueue` for communication between threads. We treated it as a black box that we posted messages to on one thread and usually processed in another thread. This appendix opens the black box and looks at the message queue's implementation. In particular, it looks at implementing the following public members:

```
public class MessageQueue : IDisposable
{
  public event EventHandler<MessageQueueEventArgs>
      MessageReceived;
  public void StartInAnotherThread();
  public void Post(object message);
  public void Post(Action<object> callback, object message);
  public void ProcessQueue();
  public void Terminate();
  // ...
}
```

These members serve the following purposes:

- `MessageReceived`. The event raised to process a message. It is raised in the thread that is processing the queue.
- `StartInAnotherThread`. Starts a dedicated thread to process messages in the queue.
- `Post`. Asynchronously adds a message. In one overload, just a message of type `object` is passed. In the other overload, a callback is also provided that is called instead of `MessageReceived` when the message is

processed. This allows different messages to be processed in different ways. We will see its usefulness when implementing `Terminate`.

- `ProcessQueue`. Synchronously processes all messages in the queue in the calling thread.
- `Terminate`. Asynchronously signals the message queue to stop processing messages. The call returns immediately without waiting for `MessageQueue` processing to terminate. Any messages posted prior to calling this method will be processed prior to message queue termination.

The implementation of MessageQueue requires two private data members: a queue for messages and an indicator of the current state, as shown in Listing A.1.

The queue holds messages that have been posted but have not yet been processed. Conceptually, messages are added to the end and removed from the beginning. The message queue can be in one of three states:

- *Stopped.* The message queue is not processing messages. If new messages are posted, they will be added to the queue but not processed until later.
- *Running.* The message queue is running; posted messages are processed in the order they are received.
- *Stopping.* The message queue has been signaled to stop by a call to `Terminate`, but it has not yet stopped.

```
private struct MessageInfo
{
  public MessageInfo(Action<object> callback, object message)
  {
    Callback = callback;
    Message = message;
  }

  public Action<object> Callback;
  public object Message;
}

private enum State
{
  Stopped,
  Running,
  Stopping
}

private List<MessageInfo> _queue = new List<MessageInfo>();
private State _state;
```

Listing A.1. MessageQueue private data members

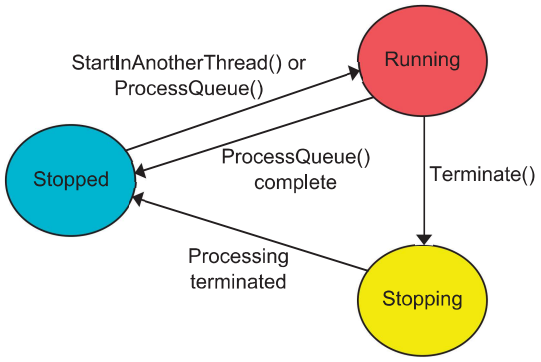


Figure A.1. Transitions between message queue states.

The transitions between these states are shown in Figure A.1. A message queue usually spends most of its time in the `Running` state, either waiting for messages or processing them.

Access to the two data members is protected by a mutex.[1] In C#, any reference type can be used as a mutex with the lock keyword, so we use `_queue` as the lock object for convenience. We are careful to hold the lock for the minimum amount of time necessary to achieve the necessary mutual exclusion.

The implementation for `Post` is shown in Listing A.2. The overload taking just a message is implemented in terms of the overload taking a callback and a message. The queue stores both the callback and the message using an instance of MessageInfo introduced in Listing A.1.

```
public void Post(object message)
{
  Post(null, message);
}

public void Post(Action<object> callback, object message)
{
  lock (_queue)
  {
    _queue.Add(new MessageInfo(callback, message));
    Monitor.Pulse(_queue);
  }
}
```

Listing A.2. MessageQueue.`Post` implementation.

[1]For best performance, a nonblocking message queue can be implemented using a singly linked list and atomic compare-and-swap operations. We use a mutex to keep our implementation simple; writing lock-free code can be error prone [162].

The most important part of the `Post` implementation is the thread synchronization. Since `Post` can be called from multiple threads at the same time, and the processing thread could also be accessing `_queue`, `_queue` is protected with a lock so only one thread at a time can access it. After obtaining the lock, `Post` adds an item to the queue and uses Monitor.`Pulse` to signal the potentially waiting processing thread that a message is now available for processing.[2] The object Monitor.`Pulse` only wakes up a thread that is already waiting on the same lock object. It will have no effect on a call to Monitor.`Wait` that happens after the pulse. For that reason, it is important that the call to Monitor.`Pulse` be inside the lock. Otherwise, the pulse could occur after the processing thread has decided to call Monitor.`Wait`, but before it actually does so.

Question ○○◯◯

An alternative to the asynchronous `Post` is the synchronous `Send`, which blocks until the message is processed in the processing thread. How would you implement `Send`? When is deadlock possible? We've included an implementation of `Send` in MessageQueue. How does it compare to your implementation?

As described previously, messages are processed when `ProcessQueue` is called explicitly or continuously by a dedicated thread that is started with a call to `StartInAnotherThread`. The implementation of `ProcessQueue` is shown in Listing A.3.

We must first lock in order to avoid a race condition if `Post` is called in another thread, or if two different threads call `ProcessQueue` at the same time. Once we've established that no other thread is currently processing queued messages, we set the `_state` to `Running`, pull all of the currently queued messages into a local list, clear `_queue`, and release the lock. Notably, we release the lock before we actually process the messages. This maximizes parallelism by allowing other threads to `Post` messages even while previously posted messages are being processed.

Also note how we move the entire list of messages from `_queue` to `current` all at once, rather than removing one message and processing it before removing the next message. This has two advantages. First, it allows us to use List<MessageInfo>, a simple array-like data structure, rather

[2] In C++, the .NET lock keyword and Monitor class can be replaced with `boost::mutex` and `boost::condition_variable`, respectively. In Java, they can be replaced with the `synchronized` keyword and the `notify` method on `java.lang.Object`.

```
public void ProcessQueue()
{
  List<MessageInfo> current = null;

  lock (_queue)
  {
    if (_state != State.Stopped)
    {
      throw new InvalidOperationException(
          "The MessageQueue is already running.");
    }

    if (_queue.Count > 0)
    {
      _state = State.Running;
      current = new List<MessageInfo>(_queue);
      _queue.Clear();
    }
  }

  if (current != null)
  {
    ProcessCurrentQueue(current);
    lock (_queue)
    {
      _state = State.Stopped;
    }
  }
}
```

Listing A.3. MessageQueue.`ProcessQueue` implementation.

than a more complicated data structure that allows items to be efficiently removed from the beginning. Second, and more importantly, it minimizes locking overhead.

To avoid the lock overhead of `Post`, the rendering thread can add requests to a local queue and only lock once per frame when adding all requests. What are the trade-offs in doing so? How could this negatively affect the parallelism between the rendering thread and the worker thread?

○○○○ Question

Once the lock is released, the `current` queue is processed by `ProcessCurrentQueue`, shown in Listing A.4. Finally, the `_state` is transitioned back to `Stopped`.

We simply loop over the messages in the queue and process each one. A message is processed in one of two ways, depending on which overload of `Post` was used to post the message. If the message has a callback, the callback is invoked and passed the message data. Otherwise,

```
private void ProcessCurrentQueue(List<MessageInfo> currentQueue)
{
  for (int i = 0; i < currentQueue.Count; ++i)
  {
    if (_state == State.Stopping)
    {
      // Push the remainder of 'current' back into '_queue'.
      lock (_queue)
      {
        currentQueue.RemoveRange(0, i);
        _queue.InsertRange(0, currentQueue);
      }
      break;
    }

    MessageInfo message = currentQueue[i];
    if (message.Callback != null)
    {
      message.Callback(message.Message);
    }
    else
    {
      EventHandler<MessageQueueEventArgs> e = MessageReceived;
      if (e != null)
      {
        e(this, new MessageQueueEventArgs(message.Message));
      }
    }
  }
}
```

Listing A.4. MessageQueue.`ProcessCurrentQueue` implementation.

the `MessageReceived` event is raised with the message data in the event arguments.

Question ○○◯◯

What happens if an exception is thrown while processing a message? How can the MessageQueue handle this possibility more robustly?

Processing the message can change the state to `Stopping`, in which case the unprocessed portion of the `current` queue is copied back to the `_queue` so that those messages will be available if and when processing is restarted and the for loop exits. Changing the state to `Stopping` is exactly how `Terminate` is implemented, as shown in Listing A.5.

It is not necessary to lock before updating `_state` in `StopQueue` because this method is posted to the queue as a message and executed by the thread that is processing messages. Since no other thread cares about this state transition, no thread synchronization is necessary.

```
public void Terminate()
{
  Post(StopQueue, null);
}

private void StopQueue(object userData)
{
  _state = State.Stopping;
}
```

Listing A.5. Implementing MessageQueue.`Terminate` with `Post` and a callback method.

Given your knowledge of `Terminate` and `Send`, how would you implement the synchronous `TerminateAndWait`?

○○○○ Question

So far, we have a working MessageQueue that allows us to call `ProcessQueue` to synchronously process the messages posted to it. Listing A.6 shows `StartInAnotherThread`, which spins up a thread dedicated to processing messages posted to the queue.

Much like `ProcessQueue`, `StartInAnotherThread` takes the lock and then verifies that no other thread is already processing messages. It then transitions the `_state` to `Running` and starts a new thread. The thread is configured as a background thread, which means it will not stop the overall process in which it is running from terminating. The entry point for the new thread is a method called `Run`, shown in Listing A.7.

```
public void StartInAnotherThread()
{
  lock (_queue)
  {
    if (_state != State.Stopped)
    {
      throw new InvalidOperationException(
          "The MessageQueue is already running.");
    }

    _state = State.Running;
  }

  Thread thread = new Thread(Run);
  thread.IsBackground = true;
  thread.Start();
}

private void Run() { /* ... */ }
```

Listing A.6. MessageQueue.`StartInAnotherThread` implementation.

```
private void Run()
{
  List<MessageInfo> current = new List<MessageInfo>();

  do
  {
    lock (_queue)
    {
      if (_queue.Count > 0)
      {
        current.AddRange(_queue);
        _queue.Clear();
      }
      else
      {
        Monitor.Wait(_queue);

        current.AddRange(_queue);
        _queue.Clear();
      }
    }

    ProcessCurrentQueue(current);
    current.Clear();
  } while (_state == State.Running);

  lock (_queue)
  {
    _state = State.Stopped;
  }
}
```

Listing A.7. Processing messages in the queue with MessageQueue.`Run`.

The implementation of this method is in many ways similar to the implementation of `ProcessQueue` shown in Listing A.3. We still take the lock, copy the contents of the `_queue` into a local `current` queue, release the lock, and process the `current` queue. The difference, though, is that the whole process is now wrapped in a do ... while loop that repeats the process until the queue is no longer in the `Running` state. That will happen when the message posted by `Terminate` is processed and transitions the state to `Stopping`. In other words, it keeps processing messages in a loop until terminated.

Another significant difference is the call to Monitor.`Wait` when the queue has no messages. Without that call, the MessageQueue would still work! It is very important to include it, though, because it keeps the do ... while loop from spinning at full speed, busy waiting, and using lots of CPU time without doing any useful work. The alert reader may have noticed that, until now, we have not justified the existence of the call to Monitor.`Pulse` in `Post`. The call to Monitor.`Wait` in this method is the reason that it exists. The object Monitor.`Wait` causes the thread to wait, *without using any CPU time*, until another thread calls Monitor.`Pulse`.

MessageQueue is unbounded; it can grow indefinitely if messages are posted faster than they are consumed. Add a `MaximumLength` property so `Post` blocks if adding a message to the queue will exceed this property's value.

◯◯◯◦ Try This

MessageQueue allows multiple threads to post to the same queue but only a single thread to process messages. To emulate multiple processing threads, we suggested multiple message queues, scheduled using round-robin in Section 10.3.2. Modify the message queue to support multiple processing threads. Start with round-robin scheduling, then modify the design to allow user-defined scheduling algorithms.

◯◯◯◦ Try This

Bibliography

[1] Kurt Akeley and Jonathan Su. "Minimum Triangle Separation for Correct z-Buffer Occlusion." In *Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*. New York: ACM Press, 2006. Available at http://research.microsoft.com/pubs/79213/GH%202006%20p027%20Akeley%20Su.pdf.

[2] Kurt Akeley. "The Hidden Charms of the z-Buffer." In *IRIS Universe 11*, pp. 31–37, 1990.

[3] Tomas Akenine-Moller, Eric Haines, and Naty Hoffman. *Real-Time Rendering*, Third edition. Wellesley, MA: A K Peters, Ltd., 2008.

[4] Johan Andersson and Natalya Tatarchuk. "Frostbite: Rendering Architecture and Real-Time Procedural Shading & Texturing Techniques." In *Game Developers Conference*, 2007. Available at http://developer.amd.com/assets/Andersson-Tatarchuk-FrostbiteRenderingArchitecture%28GDC07_AMD_Session%29.pdf.

[5] Johan Andersson. "Terrain Rendering in Frostbite Using Procedural Shader Splatting." In *Proceedings of SIGGRAPH 07: ACM SIGGRAPH 2007 Courses*, pp. 38–58. New York: ACM Press, 2007. Available at http://developer.amd.com/media/gpu_assets/Andersson-TerrainRendering(Siggraph07).pdf.

[6] Johan Andersson. "Parallel Graphics in Frostbite—Current & Future." In *SIGGRAPH*, 2009. Available at http://s09.idav.ucdavis.edu/talks/04-JAndersson-ParallelFrostbite-Siggraph09.pdf.

[7] Edward Angel. *Interactive Computer Graphics: A Top-Down Approach Using OpenGL*, Fourth edition. Reading, MA: Addison Wesley, 2005.

[8] Apple. "Enabling Multi-Threaded Execution of the OpenGL Framework." Technical report, Apple, 2006. Available at http://developer.apple.com/library/mac/#technotes/tn2006/tn2085.html.

[9] Arul Asirvatham and Hugues Hoppe. "Terrain Rendering Using GPU-Based Geometry Clipmaps." In *GPU Gems 2*, edited by Matt Pharr, pp. 27–45. Reading, MA: Addison-Wesley, 2005. Available at http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter02.html.

[10] Andreas Baerentzen, Steen Lund Nielsen, Mikkel Gjael, Bent D. Larsen, and Niels Jaergen Christensen. "Single-Pass Wireframe Rendering." In *Proceedings of SIGGRAPH 06: ACM SIGGRAPH 2006 Sketches*. New York: ACM Press, 2006. Available at http://www2.imm.dtu.dk/pubdb/views/publication_details.php?id=4884.

[11] Steve Baker. "Learning to Love Your z-Buffer." Available at http://www.sjbaker.org/steve/omniv/love_your_z_buffer.html, 1999.

[12] Avi Bar-Zeev. "How Google Earth Really Works." Available at http://www.realityprime.com/articles/how-google-earth-really-works, 2007.

[13] Sean Barrett. "Enumeration of Polygon Edges from Vertex Windings." Available at http://nothings.org/computer/edgeenum.html, 2008.

[14] Louis Bavoil. "Efficient Multifragment Effects on Graphics Processing Units." Master's thesis, University of Utah, 2007. Available at http://www.sci.utah.edu/~csilva/papers/thesis/louis-bavoil-ms-thesis.pdf.

[15] Bill Bilodeau and Mike Songy. "Real Time Shadows." In *Creativity '99, Creative Labs Inc. Sponsored Game Developer Conferences*, 1999.

[16] Ruzinoor bin Che Mat and Dr. Mahes Visvalingam. "Effectiveness of Silhouette Rendering Algorithms in Terrain Visualisation." In *Proceedings of the National Conference on Computer Graphics and Multimedia (CoGRAMM)*, 2002. Available at http://staf.uum.edu.my/ruzinoor/COGRAMM.pdf.

[17] Charles Bloom. "Terrain Texture Compositing by Blending in the Frame-Buffer." Available at http://www.cbloom.com/3d/techdocs/splatting.txt, 2000.

[18] Charles Bloom. "View Independent Progressive Meshes (VIPM)." Available at http://www.cbloom.com/3d/techdocs/vipm.txt, 2000.

[19] Jeff Bolz. "ARB_ES2_compatibility." Available at http://www.opengl.org/registry/specs/ARB/ES2_compatibility.txt, 2010.

[20] Jeff Bolz. "ARB_shader_subroutine." Available at http://www.opengl.org/registry/specs/ARB/shader_subroutine.txt, 2010.

[21] Flavien Brebion. "Tip of the Day: Logarithmic z-Buffer Artifacts Fix." Available at http://www.gamedev.net/blog/73/entry-2006307-tip-of-the-day-logarithmic-zbuffer-artifacts-fix/, 2009.

[22] Pat Brown. "EXT_texture_array." Available at http://www.opengl.org/registry/specs/EXT/texture_array.txt, 2008.

[23] Bruce M. Bush. "The Perils of Floating Point." Available at http://www.lahey.com/float.htm, 1996.

[24] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. "Bigtable: A Distributed Storage System for Structured Data." In *Proceedings of the 7th Conference on USENIX Symposium on Operating Systems Design and Implementation*, 7, 7, pp. 205–218, 2006. Available at http://labs.google.com/papers/bigtable-osdi06.pdf.

[25] Bernard Chazelle. "Triangulating a Simple Polygon in Linear Time." *Discrete Comput. Geom.* 6:5 (1991), 485–524.

[26] M. Christen. "The Future of Virtual Globes—The Interactive Ray-Traced Digital Earth." In *ISPRS Congress Beijing 2008, Proceedings of Commission II, ThS 4*, 2008. Available at http://www.3dgi.ch/publications/chm/virtualglobesfuture.pdf.

[27] Malte Clasen and Hans-Christian Hege. "Terrain Rendering Using Spherical Clipmaps." In *Eurographics/IEEE-VGTC Symposium on Visualisation*, pp. 91–98, 2006. Available at http://www.zib.de/clasen/?page_id=6.

[28] Kate Compton, James Grieve, Ed Goldman, Ocean Quigley, Christian Stratton, Eric Todd, and Andrew Willmott. "Creating Spherical Worlds." In *Proceedings of SIGGRAPH 07: ACM SIGGRAPH 2007 Sketches*, p. 82. New York: ACM Press, 2007. Available at http://www.andrewwillmott.com/s2007.

[29] Wagner T. Correa, James T. Klosowski, and Claudio T. Silva. "Visibility-Based Prefetching for Interactive Out-of-Core Rendering." In *Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics*. Los Alamitos, CA: IEEE Computer Society, 2003. Available at http://www.evl.uic.edu/cavern/rg/20040525_renambot/Viz/parallel_volviz/prefetch_outofcore_viz_pvg03.pdf.

[30] Patrick Cozzi and Frank Stoner. "GPU Ray Casting of Virtual Globes." In *Proceedings of SIGGRAPH 10: ACM SIGGRAPH 2010 Posters*, p. 1. New York: ACM Press, 2010. Available at http://www.agi.com/gpuraycastingofvirtualglobes.

[31] Patrick Cozzi. "A Framework for GLSL Engine Uniforms." In *Game Engine Gems 2*, edited by Eric Lengyel. Natick, MA: A K Peters, Ltd., 2011. Available at http://www.gameenginegems.net/.

[32] Patrick Cozzi. "Delaying OpenGL Calls." In *Game Engine Gems 2*, edited by Eric Lengyel. Natick, MA: A K Peters, Ltd., 2011. Available at http://www.gameenginegems.net/.

[33] Matt Craighead, Mark Kilgard, and Pat Brown. "ARB_point_sprite." Available at http://www.opengl.org/registry/specs/ARB/point_sprite.txt, 2003.

[34] Matt Craighead. "NV_primitive_restart." Available at http://www.opengl.org/registry/specs/NV/primitive_restart.txt, 2002.

[35] Cyril Crassin, Fabrice Neyret, Sylvain Lefebvre, and Elmar Eisemann. "GigaVoxels: Ray-Guided Streaming for Efficient and Detailed Voxel Rendering." In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D)*. New York: ACM Press, 2009. Available at http://artis.imag.fr/Publications/2009/CNLE09.

[36] Chenguang Dai, Yongsheng Zhang, and Jingyu Yang. "Rendering 3D Vector Data Using the Theory of Stencil Shadow Volumes." In *ISPRS Congress Beijing 2008, Proceedings of Commission II, WG II/5*, 2008. Available at http://www.isprs.org/proceedings/XXXVII/congress/2_pdf/5_WG-II-5/06.pdf.

[37] Christian Dick, Jens Krüger, and Rüdiger Westermann. "GPU Ray-Casting for Scalable Terrain Rendering." In *Proceedings of Eurographics 2009—Areas Papers*, pp. 43–50, 2009. Available at http://wwwcg.in.tum.de/Research/Publications/TerrainRayCasting.

[38] Christian Dick. "Interactive Methods in Scientific Visualization: Terrain Rendering." In *IEEE/VGTC Visualization Symposium*, 2009. Available at http://wwwcg.in.tum.de/Tutorials/PacificVis09/Terrain.pdf.

[39] Alan Neil Ditchfield. "Honeycomb Spherical Figure." Available at http://www.neubert.net/Download/global-grid.doc, 2001.

[40] David Douglas and Thomas Peucker. "Algorithms for the Reduction of the Number of Points Required to Represent a Digitized Line or its Caricature." *The Canadian Cartographer* 10:2 (1973), 112–122.

[41] Mark Duchaineau, Murray Wolinsky, David E. Sigeti, Mark C. Miller, Charles Aldrich, and Mark B. Mineev-Weinstein. "ROAMing Terrain: Real-Time Optimally Adapting Meshes." In *Proceedings of the 8th Conference on Visualization '97*, pp. 81–88. Los Alamitos, CA: IEEE Computer Society, 1997.

[42] Jonathan Dummer. "Cone Step Mapping: An Iterative Ray-Heightfield Intersection Algorithm." Available at http://www.lonesock.net/files/ConeStepMapping.pdf, 2006.

[43] David Eberly. *3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics*, Second edition. San Francisco: Morgan Kaufmann, 2006.

[44] David Eberly. "Triangulation by Ear Clipping." Available at http://www.geometrictools.com/Documentation/TriangulationByEarClipping.pdf, 2008.

[45] David Eberly. "Wild Magic 5 Overview." Available at http://www.geometrictools.com/WildMagic5Overview.pdf, 2010.

[46] Christer Ericson. "Physics for Games Programmers: Numerical Robustness (for Geometric Calculations)." Available at http://realtimecollisiondetection.net/pubs/, 2007.

[47] Christer Ericson. "Order Your Graphics Draw Calls Around!" Available at http://realtimecollisiondetection.net/blog/?p=86, 2008.

[48] 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 Press, 2001.

[49] ESRI. "ESRI Shapefile Technical Description." Available at http://www.esri.com/library/whitepapers/pdfs/shapefile.pdf, 1998.

[50] Tom Forsyth. "Linear-Speed Vertex Cache Optimisation." Available at http://home.comcast.net/~tom_forsyth/papers/fast_vert_cache_opt.html, 2006.

[51] Tom Forsyth. "A Matter of Precision." Available at http://home.comcast.net/~tom_forsyth/blog.wiki.html#[[A%20matter%20of%20precision]], 2006.

[52] Tom Forsyth. "Renderstate Change Costs." Available at http://home.comcast.net/~tom_forsyth/blog.wiki.html#[[Renderstate%20change%20costs]], 2008.

[53] Anton Frühstück. "GPU Based Clipmaps." Master's thesis, Vienna University of Technology, 2008. Available at http://www.cg.tuwien.ac.at/research/publications/2008/fruehstueck-2008-gpu/.

[54] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York: W. H. Freeman, 1979.

[55] Michael Garland and Paul S. Heckbert. "Surface Simplification Using Quadric Error Metrics." In *Proceedings of SIGGRAPH 97, Computer Graphics Proceedings, Annual Conference Series*, edited by Turner Whitted, pp. 209–216. Reading, MA: Addison Wesley, 1997. Available at http://mgarland.org/files/papers/quadrics.pdf.

[56] Samuel Gateau. "Solid Wireframe." In *White Paper WP-03014-001 v01*. NVIDIA Corporation, 2007. Available at http://developer.download.nvidia.com/SDK/10.5/direct3d/Source/SolidWireframe/Doc/SolidWireframe.pdf.

[57] Nick Gebbie and Mike Bailey. "Fast Realistic Rendering of Global Worlds Using Programmable Graphics Hardware." *Journal of Game Development* 1:4 (2006), 5–28. Available at http://web.engr.oregonstate.edu/~mjb/WebMjb/Papers/globalworlds.pdf.

[58] Ryan Geiss and Michael Thompson. "NVIDIA Demo Team Secrets: Cascades." In *Game Developers Conference*, 2007. Available at http://www.slideshare.net/icastano/cascades-demo-secrets.

[59] Ryan Geiss. "Generating Complex Procedural Terrains Using the GPU." In *GPU Gems 3*, edited by Hubert Nguyen, pp. 7–37. Reading, MA: Addison Wesley, 2007. Available at http://http.developer.nvidia.com/GPUGems3/gpugems3_ch01.html.

[60] Philipp Gerasimov, Randima Fernando, and Simon Green. "Shader Model 3.0: Using Vertex Textures." Available at ftp://download.nvidia.com/developer/Papers/2004/Vertex_Textures/Vertex_Textures.pdf, 2004.

[61] Thomas Gerstner. "Multiresolution Visualization and Compression of Global Topographic Data." *GeoInformatica* 7:1 (2003), 7–32. Available at http://wissrech.iam.uni-bonn.de/research/pub/gerstner/globe.pdf.

[62] Benno Giesecke. "Space Vehicles in Virtual Globes: Recommendations for the Visualization Of Objects in Space." Available at http://www.agi.com/downloads/support/productSupport/literature/pdfs/whitePapers/2007-05-24_SpaceVehiclesinVirtualGlobes.pdf, 2007.

[63] Enrico Gobbetti, Dave Kasik, and Sung eui Yoon. "Technical Strategies for Massive Model Visualization." In *Proceedings of the 2008 ACM Symposium on Solid and Physical Modeling*, pp. 405–415. New York: ACM Press, 2008.

[64] Google. "KML Reference." Available at http://code.google.com/apis/kml/documentation/kmlreference.html, 2010.

[65] K. M. Gorski, E. Hivon, A. J. Banday, B. D. Wandelt, F. K. Hansen, M. Reinecke, and M. Bartelmann. "HEALPix: A Framework for High-Resolution Discretization and Fast Analysis of Data Distributed on the Sphere." *The Astrophysical Journal* 622:2 (2005), 759–771. Available at http://stacks.iop.org/0004-637X/622/759.

[66] Evan Hart. "OpenGL SDK Guide." Available at http://developer.download.nvidia.com/SDK/10.5/opengl/OpenGL_SDK_Guide.pdf, 2008.

[67] Chris Hecker. "Let's Get to the (Floating) Point." *Game Developer Magazine* February (1996), 19–24. Available at http://chrishecker.com/Miscellaneous_Technical_Articles#Floating_Point.

[68] Tim Heidmann. "Real Shadows, Real Time." In *IRIS Universe*, 18, 18, pp. 23–31. Fremont, CA: Silicon Graphics Inc., 1991.

[69] Martin Held. "FIST: Fast Industrial-Strength Triangulation of Polygons." Technical report, Algorithmica, 2000. Available at http://cgm.cs.mcgill.ca/~godfried/publications/triangulation.held.ps.gz.

[70] Martin Held. "FIST: Fast Industrial-Strength Triangulation of Polygons." Available at http://www.cosy.sbg.ac.at/~held/projects/triang/triang.html, 2008.

[71] Martin Held. "Algorithmische Geometrie: Triangulation." Available at http://www.cosy.sbg.ac.at/~held/teaching/compgeo/slides/triang_slides.pdf, 2010.

[72] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*, Fourth edition. San Francisco: Morgan Kaufmann, 2006.

[73] John Hershberger and Jack Snoeyink. "Speeding Up the Douglas-Peucker Line-Simplification Algorithm." In *Proc. 5th Intl. Symp. on Spatial Data Handling*, pp. 134–143, 1992. Available at http://www.bowdoin.edu/~ltoma/teaching/cs350/spring06/Lecture-Handouts/hershberger92speeding.pdf.

[74] Hugues Hoppe. "Smooth View-Dependent Level-of-Detail Control and Its Application to Terrain Rendering." In *Proceedings of the Conference on Visualization '98*, pp. 35–42. Los Alamitos, CA: IEEE Computer Society, 1998.

[75] Takeo Igarashi and Dennis Cosgrove. "Adaptive Unwrapping for Interactive Texture Painting." In *ACM Symposium on Interactive 3D Graphics*, pp. 209–216. New York: ACM Press, 2001. Available at http://www-ui.is.s.u-tokyo.ac.jp/~takeo/papers/i3dg2001.pdf.

[76] Ivan-Assen Ivanov. "Practical Texture Atlases." In *Gamasutra*, 2006. Available at http://www.gamasutra.com/features/20060126/ivanov_01.shtml.

[77] Tim Jenks. "Terrain Texture Blending on a Programmable GPU." Available at http://www.jenkz.org/articles/terraintexture.htm, 2005.

[78] Jukka Jylänki. "A Thousand Ways to Pack the Bin—A Practical Approach to Two-Dimensional Rectangle Bin Packing." Available at http://clb.demon.fi/files/RectangleBinPack.pdf, 2010.

[79] Jukka Jylänki. "Even More Rectangle Bin Packing." Available at http://clb.demon.fi/projects/even-more-rectangle-bin-packing, 2010.

[80] Anu Kalra and J.M.P. van Waveren. "Threading Game Engines: QUAKE 4 & Enemy Territory QUAKE Wars." In *Game Developer Conference*, 2008. Available at http://mrelusive.com/publications/presentations/2008_gdc/GDC%2008%20Threading%20QUAKE%204%20and%20ETQW%20Final.pdf.

[81] Brano Kemen. "Floating Point Depth Buffer." Available at http://outerra.blogspot.com/2009/12/floating-point-depth-buffer.html, 2009.

[82] Brano Kemen. "Logarithmic Depth Buffer." Available at http://outerra.blogspot.com/2009/08/logarithmic-z-buffer.html, 2009.

[83] R. Keys. "Cubic Convolution Interpolation for Digital Image Processing." *IEEE Transactions on Acoustics, Speech and Signal Processing* ASSP-29 (1981), 1153–1160.

[84] Mark Kilgard and Daniel Koch. "ARB_fragment_coord_conventions." Available at http://www.opengl.org/registry/specs/ARB/fragment_coord_conventions.txt, 2009.

[85] Mark Kilgard and Daniel Koch. "ARB_provoking_vertex." Available at http://www.opengl.org/registry/specs/ARB/provoking_vertex.txt, 2009.

[86] Mark Kilgard and Daniel Koch. "ARB_vertex_array_bgra." Available at http://www.opengl.org/registry/specs/ARB/vertex_array_bgra.txt, 2009.

[87] Mark Kilgard, Greg Roth, and Pat Brown. "GL_ARB_separate_shader_objects." Available at http://www.opengl.org/registry/specs/ARB/separate_shader_objects.txt, 2010.

[88] Mark Kilgard. "Avoiding 16 Common OpenGL Pitfalls." Available at http://www.opengl.org/resources/features/KilgardTechniques/oglpitfall/, 2000.

[89] Mark Kilgard. "More Advanced Hardware Rendering Techniques." In *Game Developers Conference*, 2001. Available at http://developer.download.nvidia.com/assets/gamedev/docs/GDC01_md2shader_PDF.zip.

[90] Mark Kilgard. "OpenGL 3: Revolution through Evolution." In *Proceedings of SIGGRAPH Asia*, 2008. Available at http://www.khronos.org/developers/library/2008_siggraph_asia/OpenGL%20Overview%20SIGGRAPH%20Asia%20Dec08%20.pdf.

[91] Mark Kilgard. "EXT_direct_state_access." Available at http://www.opengl.org/registry/specs/EXT/direct_state_access.txt, 2010.

[92] Donald E. Knuth. *Seminumerical Algorithms*, Second edition. The Art of Computer Programming, vol. 2, Reading, MA: Addison-Wesley, 1997.

[93] Jaakko Konttinen. "ARB_debug_output." Available at http://www.opengl.org/registry/specs/ARB/debug_output.txt, 2010.

[94] Eugene Lapidous and Guofang Jiao. "Optimal Depth Buffer for Low-Cost Graphics Hardware." In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, pp. 67–73, 1999. Available at http://www.graphicshardware.org/previous/www_1999/presentations/d-buffer/.

[95] Cedric Laugerotte. "Tessellation of Sphere by a Recursive Method." Available at http://student.ulb.ac.be/~claugero/sphere/, 2001.

[96] Jon Leech. "ARB_sync." Available at http://www.opengl.org/registry/specs/ARB/sync.txt, 2009.

[97] Aaron Lefohn and Mike Houston. "Beyond Programmable Shading." In *ACM SIGGRAPH 2008 Courses*, 2008. Available at http://s08.idav.ucdavis.edu/.

[98] Aaron Lefohn and Mike Houston. "Beyond Programmable Shading." In *ACM SIGGRAPH 2009 Courses*, 2009. Available at http://s09.idav.ucdavis.edu/.

[99] Aaron Lefohn and Mike Houston. "Beyond Programmable Shading." In *ACM SIGGRAPH 2010 Courses*, 2010. Available at http://bps10.idav.ucdavis.edu/.

[100] Ian Lewis. "Getting More From Multicore." In *Game Developer Conference*, 2008. Available at http://www.microsoft.com/downloads/en/details.aspx?FamilyId=A36FE736-5FE7-4E08-84CF-ACCF801538EB&displaylang=en.

[101] Peter Lindstrom and Jonathan D. Cohen. "On-the-Fly Decompression and Rendering of Multiresolution Terrain." In *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pp. 65–73. New York: ACM Press, 2010. Available at https://e-reports-ext.llnl.gov/pdf/371781.pdf.

[102] Peter Lindstrom, David Koller, William Ribarsky, Larry F. Hodges, Nick Faust, and Gregory A. Turner. "Real-Time, Continuous Level of Detail Rendering of Height Fields." In *Proceedings of SIGGRAPH 96, Computer Graphics Proceedings, Annual Conference Series*, edited by Holly Rushmeier, pp. 109–118. Reading, MA: Addison Wesley, 1996.

[103] Benj Lipchak, Greg Roth, and Piers Daniell. "ARB_get_program_binary." Available at http://www.opengl.org/registry/specs/ARB/get_program_binary.txt, 2010.

[104] William E. Lorensen and Harvey E. Cline. "Marching Cubes: A High Resolution 3D Surface Construction Algorithm." *SIGGRAPH Computer Graphics* 21:4 (1987), 163–169.

[105] Frank Losasso and Hugues Hoppe. "Geometry Clipmaps: Terrain Rendering Using Nested Regular Grids." In *Proceedings of SIGGRAPH 2004 Papers*, pp. 769–776. New York: ACM Press, 2004. Available at http://research.microsoft.com/en-us/um/people/hoppe/proj/geomclipmap/.

[106] Kok-Lim Low and Tiow-Seng Tan. "Model Simplification Using Vertex-Clustering." In *I3D 97: Proceedings of the 1997 Symposium on Interactive 3D Graphics*, pp. 75–ff. New York: ACM Press, 1997.

[107] David Luebke, Martin Reddy, Jonathan Cohen, Amitabh Varshney, Benjamin Watson, and Robert Huebner. *Level of Detail for 3D Graphics*. San Francisco: Morgan Kaufmann, 2002. Available at http://lodbook.com/.

[108] Henrique S. Malvar. "Fast Progressive Image Coding without Wavelets." In *Proceedings of the Conference on Data Compression*. Washington, DC: IEEE Computer Society, 2000. Available at http://research.microsoft.com/apps/pubs/?id=101991.

[109] CCP Mannapi. "Awesome Looking Planets." Available at http://www.eveonline.com/devblog.asp?a=blog&bid=724, 2010.

[110] Paul Martz. "OpenGL FAQ: The Depth Buffer." Available at http://www.opengl.org/resources/faq/technical/depthbuffer.htm, 2000.

[111] Grégory Massal. "Depth Buffer—The Gritty Details." Available at http://www.codermind.com/articles/Depth-buffer-tutorial.html, 2006.

[112] Morgan McGuire and Kyle Whitson. "Indirection Mapping for Quasi-Conformal Relief Mapping." In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D '08)*, 2008. Available at http://graphics.cs.williams.edu/papers/IndirectionI3D08/.

[113] Tom McReynolds and David Blythe. *Advanced Graphics Programming Using OpenGL, The Morgan Kaufmann Series in Computer Graphics*. San Francisco: Morgan Kaufmann, 2005.

[114] "Direct3D 11 MultiThreading." Available at http://msdn.microsoft.com/en-us/library/ff476884.aspx, 2010.

[115] Microsoft. "DirectX Software Development Kit—RaycastTerrain Sample." Available at http://msdn.microsoft.com/en-us/library/ee416425(v=VS.85).aspx, 2008.

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

[117] NASA Solar System Exploration. "Mars: Moons: Phobos." Available at http://solarsystem.nasa.gov/planets/profile.cfm?Object=Mar_Phobos, 2003.

[118] National Imagery and Mapping Agency. *Department of Defense World Geodetic System 1984: Its Definition and Relationships with Local Geodetic Systems*, Third edition. National Imagery and Mapping Agency, 2000. Available at http://earth-info.nga.mil/GandG/publications/tr8350.2/wgs84fin.pdf.

[119] Kris Nicholson. "GPU Based Algorithms for Terrain Texturing." Master's thesis, University of Canterbury, 2008. Available at http://www.cosc.canterbury.ac.nz/research/reports/HonsReps/2008/hons_0801.pdf.

[120] NVIDIA. "Using Vertex Buffer Objects." Available at http://www.nvidia.com/object/using_VBOs.html, 2003.

[121] NVIDIA. "Improve Batching Using Texture Atlases." Available at http://developer.download.nvidia.com/SDK/9.5/Samples/DEMOS/Direct3D9/src/BatchingViaTextureAtlases/AtlasCreationTool/Docs/Batching_Via_Texture_Atlases.pdf, 2004.

[122] NVIDIA. "NVIDIA CUDA Compute Unified Device Architecture Programming Guide." Available at http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf, 2008.

[123] NVIDIA. "NVIDIA GPU Programming Guide." Available at http://www.nvidia.com/object/gpu_programming_guide.html, 2008.

[124] NVIDIA. "Bindless Graphics Tutorial." Available at http://www.nvidia.com/object/bindless_graphics.html, 2009.

[125] Deron Ohlarik. "Horizon Culling." Available at http://blogs.agi.com/insight3d/index.php/2008/04/18/horizon-culling/, 2008.

[126] Deron Ohlarik. "Precisions, Precisions." Available at http://blogs.agi.com/insight3d/index.php/2008/09/03/precisions-precisions/, 2008.

[127] Deron Ohlarik. "Triangulation Rhymes with Strangulation." Available at http://blogs.agi.com/insight3d/index.php/2008/03/20/triangulation-rhymes-with-strangulation/, 2008.

[128] Deron Ohlarik. "Horizon Culling 2." Available at http://blogs.agi.com/insight3d/index.php/2009/03/25/horizon-culling-2/, 2009.

[129] Jon Olick. "Next Generation Parallelism in Games." In *Proceedings of ACM SIGGRAPH 2008 Courses*. New York: ACM Press, 2008. Available at http://s08.idav.ucdavis.edu/olick-current-and-next-generation-parallelism-in-games.pdf.

[130] Sean O'Neil. "A Real-Time Procedural Universe, Part Three: Matters of Scale." Available at http://www.gamasutra.com/view/feature/2984/a_realtime_procedural_universe_.php, 2002.

[131] Open Geospatial Consortium Inc. "OSG KML." Available at http://portal.opengeospatial.org/files/?artifact_id=27810, 2008.

[132] Charles B. Owen. "CSE 872 Advanced Computer Graphics—Tutorial 4: Texture Mapping." Available at http://www.cse.msu.edu/~cse872/tutorial4.html, 2008.

[133] David Pangerl. "Practical Thread Rendering for DirectX 9." In *GPU Pro*, edited by Wolfgang Engel. Natick, MA: A K Peters, Ltd., 2010. Available at http://www.akpeters.com/gpupro/.

[134] Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. "OptiX: A General Purpose Ray Tracing Engine." *ACM Transactions on Graphics*. Available at http://graphics.cs.williams.edu/papers/OptiXSIGGRAPH10/.

[135] Emil Persson. "ATI Radeon HD 2000 Programming Guide." Available at http://developer.amd.com/media/gpu_assets/ATI_Radeon_HD_2000_programming_guide.pdf, 2007.

[136] Emil Persson. "Depth In-Depth." Available at http://developer.amd.com/media/gpu_assets/Depth_in-depth.pdf, 2007.

[137] Emil Persson. "A Couple of Notes about z." Available at http://www.humus.name/index.php?page=News&ID=255, 2009.

[138] Emil Persson. "New Particle Trimming Tool." Available at http://www.humus.name/index.php?page=News&ID=266, 2009.

[139] Emil Persson. "Triangulation." Available at http://www.humus.name/index.php?page=News&ID=228, 2009.

[140] Emil Persson. "Making it Large, Beautiful, Fast, and Consistent: Lessons Learned Developing Just Cause 2." In *GPU Pro*, edited by Wolfgang Engel, pp. 571–596. Natick, MA: A K Peters, Ltd., 2010. Available at http://www.akpeters.com/gpupro/.

[141] Matt Pettineo. "Attack of the Depth Buffer." Available at http://mynameismjp.wordpress.com/2010/03/22/attack-of-the-depth-buffer/, 2010.

[142] Frank Puig Placeres. "Fast Per-Pixel Lighting with Many Lights." In *Game Programming Gems 6*, edited by Michael Dickheiser, pp. 489–499. Hingham, MA: Charles River Media, 2006.

[143] John Ratcliff. "Texture Packing: A Code Snippet to Compute a Texture Atlas." Available at http://codesuppository.blogspot.com/2009/04/texture-packing-code-snippet-to-compute.html, 2009.

[144] Ashu Rege. "Shader Model 3.0." Available at ftp://download.nvidia.com/developer/presentations/2004/GPU_Jackpot/Shader_Model_3.pdf, 2004.

[145] Marek Rosa. "Destructible Volumetric Terrain." In *GPU Pro*, edited by Wolfgang Engel, pp. 597–608. Natick, MA: A K Peters, Ltd., 2010. Available at http://www.akpeters.com/gpupro/.

[146] Jarek Rossignac and Paul Borrel. "Multi-Resolution 3D Approximations for Rendering Complex Scenes." In *Modeling in Computer Graphics: Methods and Applications*, edited by B. Falcidieno and T. Kunii, pp. 455–465. Berlin: Springer-Verlag, 1993. Available at http://www.cs.uu.nl/docs/vakken/ddm/slides/papers/rossignac.pdf.

[147] Randi J. Rost, Bill Licea-Kane, Dan Ginsburg, John M. Kessenich, Barthold Lichtenbelt, Hugh Malan, and Mike Weiblen. *OpenGL Shading Language*, Third edition. Reading, MA: Addison-Wesley, 2009.

[148] Pedro V. Sander, Diego Nehab, and Joshua Barczak. "Fast Triangle Reordering for Vertex Locality and Reduced Overdraw." *Proc. SIGGRAPH 07, Transactions on Graphics* 26:3 (2007), 1–9.

[149] Martin Schneider and Reinhard Klein. "Efficient and Accurate Rendering of Vector Data on Virtual Landscapes." *Journal of WSCG* 15:1–3 (2007), 59–64. Available at http://cg.cs.uni-bonn.de/de/publikationen/paper-details/schneider-2007-efficient/.

[150] William J. Schroeder, Jonathan A. Zarge, and William E. Lorensen. "Decimation of Triangle Meshes." *Proc. SIGGRAPH 92, Computer Graphics* 26:2 (1992), 65–70.

[151] Jim Scott. "Packing Lightmaps." Available at http://www.blackpawn.com/texts/lightmaps/, 2001.

[152] Mark Segal and Kurt Akeley. "The OpenGL Graphics System: A Specification (Version 3.3 Core Profile)." Available at http://www.opengl.org/registry/doc/glspec33.core.20100311.pdf, 2010.

[153] Dean Sekulic. "Efficient Occlusion Culling." In *GPU Gems*, edited by Randima Fernando. Reading, MA: Addison-Wesley, 2004. Available at http://http.developer.nvidia.com/GPUGems/gpugems_ch29.html.

[154] Jason Shankel. "Fast Heightfield Normal Calculation." In *Game Programming Gems 3*, edited by Dante Treglia. Hingham, MA: Charles River Media, 2002.

[155] Dave Shreiner and The Khronos OpenGL ARB Working Group. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1.*, Seventh edition. Reading, MA: Addison-Wesley, 2009.

[156] Irvin Sobel. "An Isotropic 3×3 Image Gradient Operator." In *Machine Vision for Three-Dimensional Scenes*, pp. 376–379. Orlando, FL: Academic Press, 1990.

[157] Wojciech Sterna. "Porting Code between Direct3D 9 and OpenGL 2.0." In *GPU Pro*, edited by Wolfgang Engel, pp. 529–540. Natick, MA: A K Peters, Ltd., 2010. Available at http://www.akpeters.com/gpupro/.

[158] David Stevenson. "A Report on the Proposed IEEE Floating Point Standard (IEEE Task p754)." *ACM SIGARCH Computer Architecture News* 8:5.

[159] Benjamin Supnik. "The Hacks of Life." Available at http://hacksoflife.blogspot.com/.

[160] Benjamin Supnik. "OpenGL and Threads: What's Wrong." Available at http://hacksoflife.blogspot.com/2008/01/opengl-and-threads-whats-wrong.html, 2008.

[161] Herb Sutter. "The Free Lunch Is Over: A Fundamental Turn toward Concurrency in Software." *Dr. Dobb's Journal* 30:3. Available at http://www.gotw.ca/publications/concurrency-ddj.htm.

[162] Herb Sutter. "Lock-Free Code: A False Sense of Security." *Dr. Dobb's Journal* 33:9. Available at http://www.drdobbs.com/cpp/210600279.

[163] Adam Szofran. "Global Terrain Technology for Flight Simulation." In *Game Developers Conference*, 2006. Available at http://www.microsoft.com/Products/Games/FSInsider/developers/Pages/GlobalTerrain.aspx.

[164] Christopher C. Tanner, Christopher J. Migdal, and Michael T. Jones. "The Clipmap: a Virtual Mipmap." In *Proceedings of SIGGRAPH 98, Computer Graphics Proceedings, Annual Conference Series*, edited by Michael Cohen, pp. 151–158. Reading, MA: Addison Wesley, 1998.

[165] Terathon Software. "C4 Engine Wiki: Editing Terrain." Available at http://www.terathon.com/wiki/index.php/Editing_Terrain, 2010.

[166] Art Tevs, Ivo Ihrke, and Hans-Peter Seidel. "Maximum Mipmaps for Fast, Accurate, and Scalable Dynamic Height Field Rendering." In *Symposium on Interactive 3D Graphics and Games (i3D'08)*, pp. 183–190, 2008. Available at http://www.tevs.eu/project_i3d08.html.

[167] Andrew Thall. "Extended-Precision Floating-Point Numbers for GPU Computation." Technical report, Alma College, 2007. Available at http://andrewthall.org/papers/.

[168] Nick Thibieroz. "Clever Shader Tricks." *Game Developers Conference*. Available at http://developer.amd.com/media/gpu_assets/04%20Clever%20Shader%20Tricks.pdf.

[169] Chris Thorne. "Origin-Centric Techniques for Optimising Scalability and the Fidelity of Motion, Interaction and Rendering." Ph.D. thesis, University of Western Australia, 2007. Available at http://www.floatingorigin.com/.

[170] Thatcher Ulrich. "Rendering Massive Terrains Using Chunked Level of Detail Control." In *SIGGRAPH 2002 Super-Size It! Scaling Up to Massive Virtual Worlds Course Notes*. New York: ACM Press, 2002. Available at http://tulrich.com/geekstuff/sig-notes.pdf.

[171] David A. Vallado and Wayne D. McClain. *Fundamentals of Astrodynamics and Applications*, Third edition. New York: Springer-Verlag, 2007. Available at http://celestrak.com/software/vallado-sw.asp.

[172] J.M.P. van Waveren. "Real-Time Texture Streaming & Decompression." Available at http://software.intel.com/en-us/articles/real-time-texture-streaming-decompression/, 2007.

[173] J.M.P. van Waveren. "Geospatial Texture Streaming from Slow Storage Devices." Available at http://software.intel.com/en-us/articles/geospatial-texture-streaming-from-slow-storage-devices/, 2008.

[174] J.M.P. van Waveren. "id Tech 5 Challenges: From Texture Virtualization to Massive Parallelization." In *SIGGRAPH*, 2009. Available at http://s09.idav.ucdavis.edu/talks/05-JP_id_Tech_5_Challenges.pdf.

[175] Gokul Varadhan and Dinesh Manocha. "Out-of-Core Rendering of Massive Geometric Environments." In *Proceedings of the Conference on Visualization '02*, pp. 69–76. Los Alamitos, CA: IEEE Computer Society, 2002.

[176] James M. Van Verth and Lars M. Bishop. *Essential Mathematics for Games and Interactive Applications*, Second edition. San Francisco: Morgan Kaufmann, 2008.

[177] Harald Vistnes. "GPU Terrain Rendering." In *Game Programming Gems 6*, edited by Michael Dickheiser, pp. 461–471. Hingham, MA: Charles River Media, 2006.

[178] Ingo Wald. "Realtime Ray Tracing and Interactive Global Illumination." Ph.D. thesis, Saarland University, 2004. Available at http://www.sci.utah.edu/~wald/PhD/.

[179] Bill Whitacre. "Spheres Through Triangle Tessellation." Available at http://musingsofninjarat.wordpress.com/spheres-through-triangle-tessellation/, 2008.

[180] David R. Williams. "Moon Fact Sheet." Available at http://nssdc.gsfc.nasa.gov/planetary/factsheet/moonfact.html, 2006.

[181] Michael Wimmer and Jiří Bittner. "Hardware Occlusion Queries Made Useful." In *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, edited by Matt Pharr and Randima Fernando. Reading, MA: Addison-Wesley, 2005. Available at http://www.cg.tuwien.ac.at/research/publications/2005/Wimmer-2005-HOQ/.

[182] Steven Wittens. "Making Worlds: 1—Of Spheres and Cubes." Available at http://acko.net/blog/making-worlds-part-1-of-spheres-and-cubes, 2009.

[183] Matthias Wloka. "Batch, Batch, Batch: What Does It Really Mean?" *Game Developers Conference*. Available at http://developer.nvidia.com/docs/IO/8230/BatchBatchBatch.pdf.

Index

About the Authors

Patrick Cozzi is a senior software developer on the 3D team at Analytical Graphics, Inc. (AGI) and a part-time lecturer in computer graphics at the University of Pennsylvania. He is a contributor to SIGGRAPH and the *Game Engine Gems* series. Before joining AGI in 2004, he worked on storage systems in IBM's Extreme Blue internship program at the Almaden Research Lab, interned with IBM's z/VM operating system team, and interned with the chipset validation group at Intel. Patrick has a master's degree in computer and information science from the University of Pennsylvania and a bachelor's degree in computer science from Penn State. His email address is pjcozzi@siggraph.org.

Kevin Ring first picked up a book on computer programming around the same time he learned to read, and he hasn't looked back since. In his software development career, he has worked on a wide range of software systems, from class libraries to web applications to 3D game engines to interplanetary spacecraft trajectory design systems, and most things in between. Kevin has a bachelor's degree in computer science from Rensselaer Polytechnic Institute and is currently the lead architect of AGI Components at Analytical Graphics, Inc. His email address is kevin@kotachrome.com.