3D Engine Design for Virtual Globes

Patrick Cozzi and Kevin Ring

Editorial, Sales, and Customer Service Office

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

Library of Congress Cataloging-in-Publication Data

To be determined

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

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.

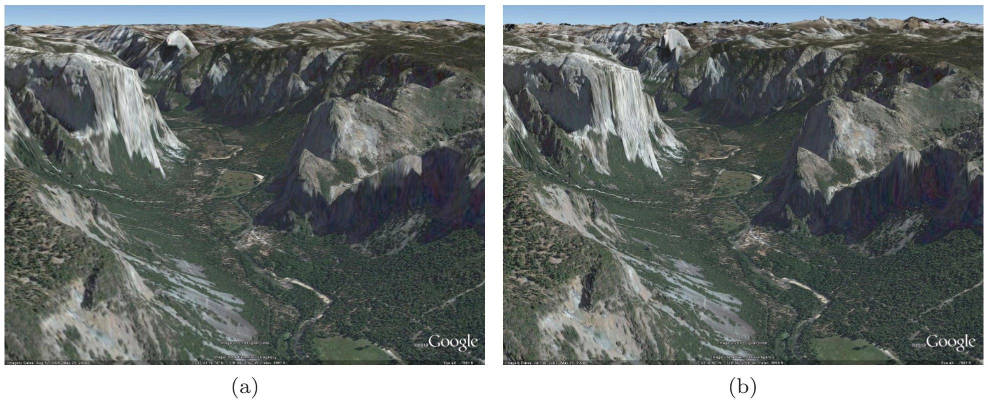


Figure 12.1. The same view of Yosemite Valley, El Capitan, and Half Dome at (a) low detail and (b) high detail. The differences are most noticeable in the shapes of the peaks in the distance. Image USDA Farm Service Agency, Image (C) 2010 DigitalGlobe. (Figures taken using Google Earth.)

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

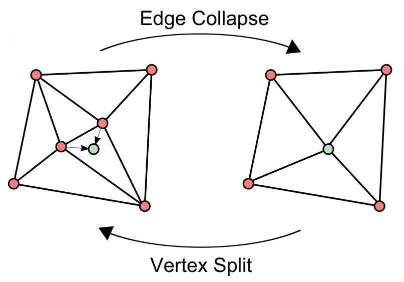


Figure 12.2. For an edge interior to a mesh, edge collapse removes two triangles and vertex split creates two triangles.

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

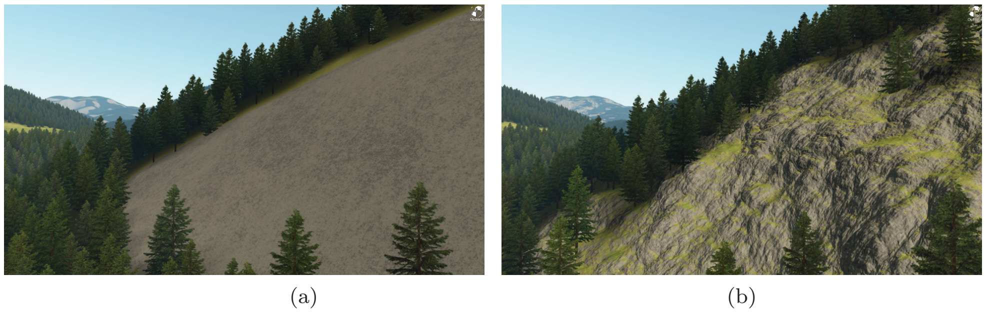


Figure 12.3. Fractal detail can turn basic terrain into a beautiful landscape. (a) Original 76 m terrain data. (b) With fractal detail. (Images courtesy of Brano Kemen, Outerra.)

12.1.3 Hierarchical Level of Detail

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

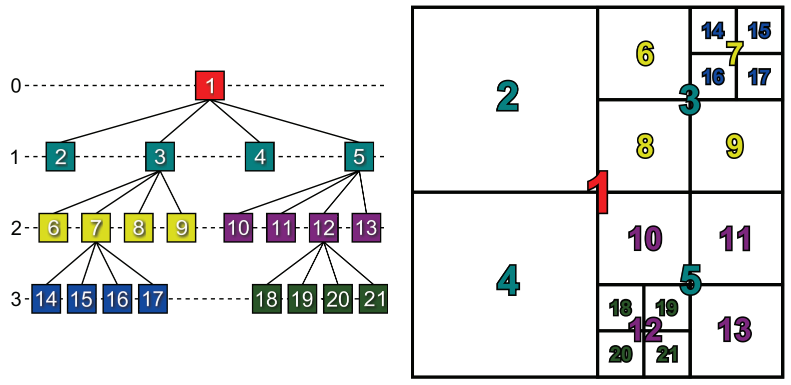
- Reducing the amount of processing by the CPU.
- Reducing the quantity of data sent over the system bus to the GPU.

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

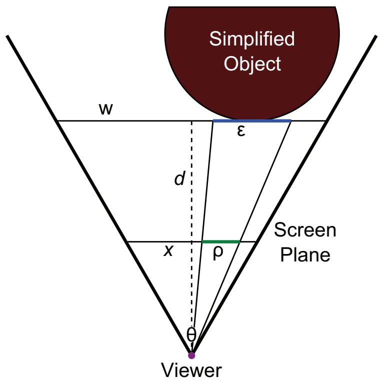


Figure 12.5. The screen-space error, ρ, of an object is estimated from the distance between the object and the viewer, the parameters of the view, and the geometric error, ϵ, of the object.

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

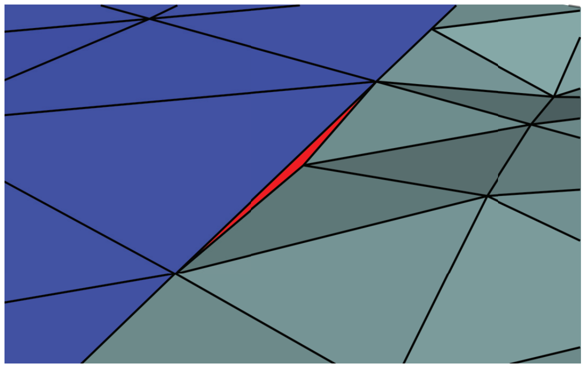


Figure 12.6. Cracking occurs when an edge shared by two adjacent LODs is divided by an additional vertex in one LOD but not the other.

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.

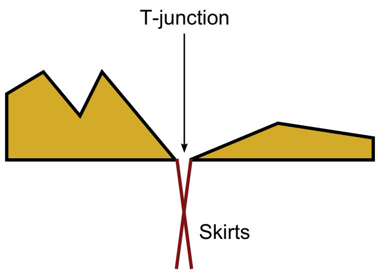


Figure 12.7. A side view of an exaggerated T-junction between two adjacent LODs. Skirts can hide T-junctions if they are angled slightly outward.

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 $\times$ 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 $\times$ 1,000 “skipped” post heights to produce one