Point Based Rendering for Massive Data Sets: A Case Study

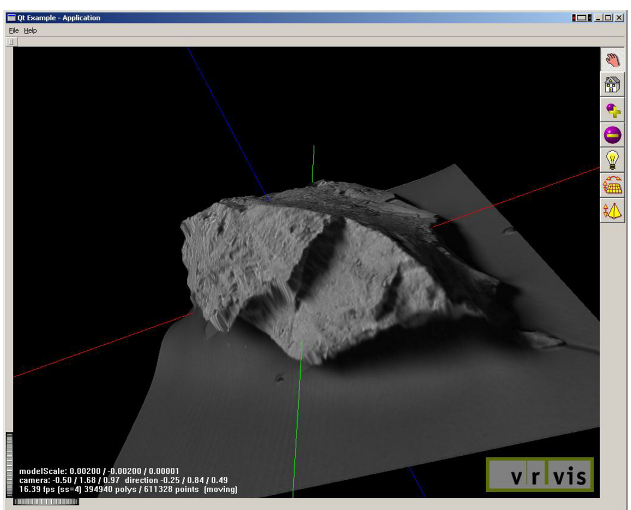
Stephan Mantler
mantler@vrvis.at

Anton L. Fuhrmann
fuhrmann@vrvis.at

VRVis Center for Virtual Reality and Visualization
Vienna, Austria

Abstract

In this case study, we describe our experiences with implementing a hybrid polygonal and point based rendering sytem for displaying large data sets. Our reasons for choosing a point based sytems are explained in detail, together with some notes about the implementation and how it was integrated into a larger environment. We conclude the report with a discussion of what worked well and what didn't, and the lessons learned.



1. Introduction

We were approached by one of our partner companies with the necessity of rendering massive microscopy data sets in real time for visual analysis and extraction of various parameters. Produced by a number of methods (stereo microscopy, shape from focus, REM and others), the input data was specified to be 2.5D data arranged in regular grids of about 2000x2000 sample points, with future sizes up to 10k x 10k planned. In addition to *z* values, 24-bit color data was acquired or generated for each sample. Due to the limited precision of the reconstruction process, the z values - and thus the coordinates of each sample in an array up to 16k x 16k - could be sufficiently represented by (scaled) 16-bit integers.

The data could also contain "invalid" points where no valid information was available, such as areas where the reconstruction failed; these areas should be omitted from rendering and show up as holes in the dataset.

The existing system reduced the regular grid in a preprocessing step and rendered the resulting mesh with a typical polygonal rendering engine. Since this preprocessing step did reduce the available information, it also adversely affected the investigation of small details of the analysed surface.

Since the host application did already exist, our work resulted in a new module that could be used as a drop-in replacement for the existing renderer. While this allowed us to leave the user interaction routines intact and to focus on the rendering aspect itself, it also lead to a few problems that will be discussed later on.

1.1. Related Work

Given that the target application is fairly close to terrain rendering in many respects, many algorithms from this area are equally applicable or could be adapted. For example, our system borrows ideas from Ulrich's Chunked LODs [11] as well as de Boer's Geomipmapping algorithm [4].

Many systems for terrain rendering are based on adaptive simplification schemes of polygonal meshes [1, 6, 7]. Although they achieve very good runtime performance, they require a fairly slow preprocessing step. This delay was one of the reasons for considering a reimplementation, as long wait times do severely impact the workflow.

An interesting approach are Sim Dietrich's Elevation Maps [5]. In this case, height information is stored as a texture and rendered directly through clever use of alpha testing. The result is in fact quite similar to Schaufler's Layered

Impostors [10]. In addition, the rendering quality degrades noticeably when the object is viewed at a near horizontal angle (such that polygons are viewed close to edge-on), especially in the near field.

Point based rendering was highlighted by several highly interesting papers, including the QSplat rendering system presented by Rusinkiewicz and Levoy [9], and the Surfels rendering primitives developed by Pfister et.al. [8], both in 2000. Any point based system is certainly inspired by these works, although the special needs of our system (such as fast retexturing and extreme close-up views) necessitated a somewhat different approach.

Hybrid point and polygon based systems were the logical next step; the Hybrid Simplification algorithm by Cohen et.al. [3] and the POP system by Chen and Nguyen [2], both presented at the Visualization 2001 conference, are of note in this area.

1.2. Why Point Based Rendering?

Considering the availability of other rendering algorithms and acceleration data structures for vaguely similiar data (e.g. terrain data), our choice of point based rendering was ultimately driven by a number of factors:

Full resolution - Current graphics hardware (“next generation” at the time of decision) was estimated to be sufficiently fast to render one sample per pixel, therefore possibly displaying the data at full resolution. Even if data reduction is necessary to view the entire dataset interactively, our goal was to provide the best possible accuracy when examining data very closely.

Speed - Given the above goals, the rendering of individual points rather than very small triangles promised a good speed-up due to less bandwith usage. Points can be rendered with a single vertex, but (disregarding triangle strips or fans) each triangle requires three times as much.

Feature Set - Some requested features, such as rendering invalid data as holes, seemed to be easier to implement correctly with a point based approach than to include them in a polygonal simplification scheme.

Faster Preprocessing - The computationally expensive data reduction step could be omitted or replaced by a faster method suitable for the point based engine.

2. Technical Description

As has been mentioned above, our system was encapsulated as a module that could be used as a drop-in replacement for the existing renderer. The final system could then use both the old and new renderer simultaneously, allowing for a good side-by-side comparison. The final integration was done by our partner company; after the module interface had been agreed upon the integration of new updates was extremely efficient.

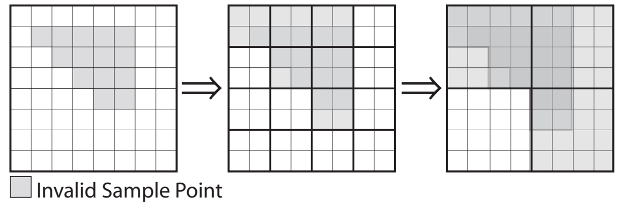


Figure 1. Growing invalid areas in chunk LODs.

Our renderer relied on the OpenGL state to be set up for rendering, ie. camera position, lighting and clip planes to be already defined correctly. As some of these values are required within the renderer for error estimation and other purposes, appropriate interfaces to the relevant objects had to be defined and implemented.

2.1. A Hybrid Rendering System

To facilitate processing and assure high performance, the data set is partitioned into square “chunks” similar to the system described by [11]. Each chunk is processed completely independently; neighboring chunks overlap by one sample row to avoid holes during rendering. The chunk size is determined at runtime, depending on the overall dimensions of the data set.

In a preprocessing step, a hierarchy of lower resolution grids is constructed. Invalid data points, normal and texture information are handled appropriately: if all data points are valid, their average values are used for one lower resolution data point in their center. If at least one of them is marked as invalid, the lower resolution data point is also invalid (see Figure 1)[1]. To optimize polygonal rendering for the near field, triangle strips of the full resolution data are also precalculated as well.

During rendering, the level of detail of each chunk is calculated through estimating the maximum screen space distance betwen two adjacent point samples. Note that the maximum world space distance can be calculated during preprocessing, leaving only the projection to screen space

1 This leads to “growing” invalid areas in the far field, but we think that this preserves a meaningful visual cue about invalid data in this area, and is therefore preferable to the alternative “shrinking”. A truly faithful rendering would have to display such data with semitransparent primitives, adding an additional overhead during rendering.

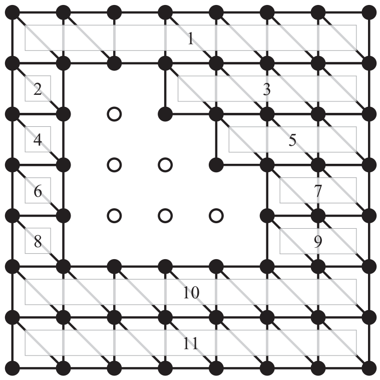


Figure 2. Creating triangle strips. Filled circles represent valid data; empty circles depict invalid sample points. Numbers show how triangle strips are arranged for rendering.

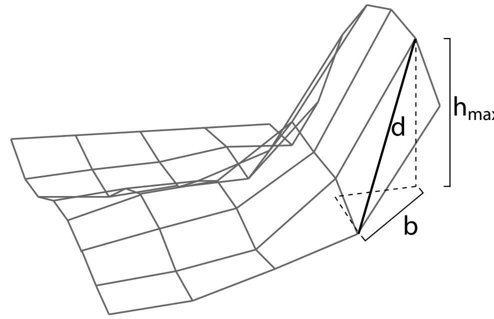


Figure 3. Estimating the maximum point size required to render a chunk. d can be easily computed in world space coordinates from h_{max} and the base diagonal b; projecting a sphere with radius d from the closest point of the chunk to the image plane then yields the maximum distance between two samples.

for runtime calculations (see Figure 3). By comparing this screen space estimate to the current OpenGL point size, the appropriate level of detail can be found efficiently. By choosing a level of detail where this distance is just smaller than the OpenGL point size, the rendering is guaranteed to cover the screen without holes, and automatically uses the lowest possible resolution. Finally, if the screen size is too large to be covered with OpenGL points even at the highest resolution, this near field chunk is rendered polygonally.

In general, this mixed display of polygonal and point based rendering works very well. Since the z-Buffer is set correctly, interpenetration always leads to the expected (correct) results. This is in stark contrast to impostor rendering, where correct values of the z-Buffer are difficult to achieve.

To provide additional feedback, other geometry can be rendered as well. Through a number of stencil buffer operations, height lines can be simulated, and (line based) locators can be used to place markers on points of interest. The simulation of height lines is performed with two additional rendering passes. The first pass only updates the stencil buffer and renders boxes with +1 / -1 stencil operations for each height line, with z-buffer testing (but no updates) from the previously rendered data. The second pass simply displays a screen aligned quad, using a stencil test for not equal zero (see Figure 4).

Optionally, the standard photographic texture can be replaced by an elevation color code; this is implemented as a 1D texture for polygons with automatic generation of the S coordinate. For point data, texture lookup turned out to be surprisingly expensive, so the elevation color code is rendered through an OpenGL color array in this case, just like the photographic texture.

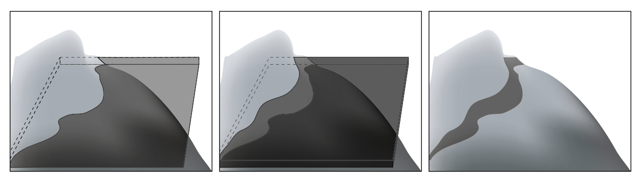


Figure 4. Drawing height lines with the stencil buffer. The shaded area represents previously rendered height field. (left) Back facing layers are rendered with an operation of -1. (middle) Forward facing layers are rendered with an operation of +1. (right) a fullscreen quad is rendered with a stencil test for nonzero values, leaving only the height line.

However, we did run into a number of unexpected issues: Two-sided lighting does not work as expected for point based systems; while the normal vectors of back facing polygons are flipped as expected, point samples remain unlit and noticeably darker. Although this is not in error with the OpenGL specifications (which does not mention point primitives in the context of two-sided lighting at all), it is an unfortunate situation and leads to unacceptable visual discrepancies (see 5).

Also, using texture coordinates, or even automatic texture coordinate generation, for point samples caused a sig-

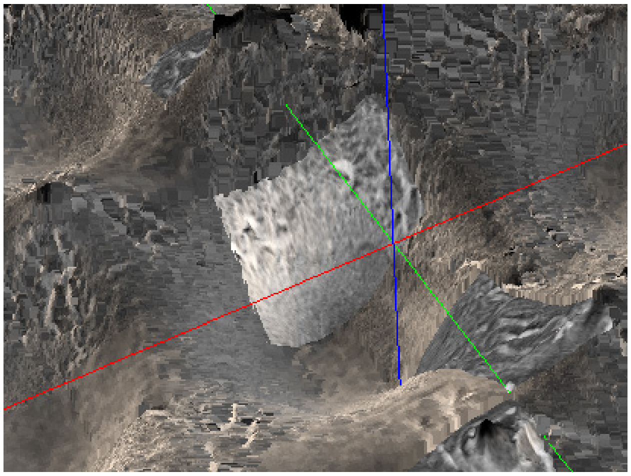


Figure 5. Unlit backside despite two-sided lighting. The majority of the surface is rendered as points; only the center chunk is displayed as polygons. It is clearly brighter.

nificant slowdown. This forced us to precalculate vertex color information for the use of elevation color codes. In the case of polygons that span a relatively large section in Z, the use of textures for elevation color is imperative for correct display, as otherwise the intermediate color ramp would not be displayed correctly. As a result, our application toggles frequently between enabling and disabling textures when rendering elevation color codes.

2.2. Interaction

To support interaction with the data set, picking has been implemented via the OpenGL Feedback Buffer mechanism. Only a small subset of the scene (a box of 5x5 pixels centered on the current mouse coordinates) is rendered, and the feeback buffer returns exact vertex coordinates of any primitives that have been displayed. This information can then be easily converted to object space coordinates, which in turn results in the actual data points.

As has been noted above, standard navigation through the data set was already supported by the host environment and used an 'examiner viewer' type of scheme. However, since our system needs the current camera position and other data from the viewer configuration, minor changes had to be made in the host environment to expose this information. For example, our renderer tracked the current frame rate during interaction to achieve sufficient performance, and optionally produced a full-resolution high quality rendering when the system was idle.

2.3. Project Lifecycle

After several update - integrate - review cycles, most open issues in the requirements document had been resolved, but new issues were consistently being brought up. Some where due to the way we had implemented a certain feature; others had simply been overlooked in the original specifications.

One of the key points in this process was that some of the test data was supplied to us only at a very late point in the development. We had received a few data sets to begin with, but they were relatively benign. Only later did we learn that some of the samples had very high frequency textures (requiring appropriate pre-filtering for downsampling) and other properties that we had not anticipated. This caused a significant redesign of some code segments that could have easily been avoided if a full set of worst-case test data had been available at an earlier time.

This development cycle ended when our partner had to begin preparations (ie. code freeze) for the upcoming new release of their product. At this point, the decision was made to keep the current (previous) system on the grounds that our new renderer still had unresolved issues that would have been show stoppers in the final product.

3. Conclusion

At the end of the development cycle, we had an application that fully met a number of the original (and 'evolved') targets; others were partially successful and some aspects were either unresolved or had to be removed again.

With our final application, the following main goals were met successfully:

Fast Preprocessing. Our preprocessing method is significantly faster than before, even though it is still too slow for interactive work - loading and preprocessing a 2048*2048 dataset takes about one minute. Most of this time is spent performing I/O operations, so this would definitely benefit from fast disks (e.g. RAID storage systems).

Full Resolution. Even for large datasets, the full resolution can be retained and examined in close-up views. Our representation is quite compact in memory, and could be expanded to supporting out-of-core data for even larger datasets. Each sample point requires 16 bytes of storage, and including triangle strips a data set of $m*n$ samples requires about 80*m*n bytes. Therefore, a typical 2048x2048 data set requires about 320MB.

Hybrid Point/Polygonal Rendering. In most cases (except for lighting, see below), switching between point based and polygonal representations is almost seamless and hardly noticeable. We actually had to implement a 'color code' version to confirm that switching between the polygonal and point based representations did take place as intended.

However, there are also some points that did not work out as well as expected. Some of them were fundamental drawbacks of the point based approach that we only discovered during development; others were solutions that sounded fine at first but turned out to be counterproductive.

Performance. We did not achieve a significant performance improvement in comparison to the previous (purely polygonal) representation. We suspect that the point rendering path, especially when used with vertex arrays and OpenGL extensions, is not as optimized as the polygonal rendering paths. Therefore, overall performance improvements of "next-generation" graphics cards do not guarantee that point based rendering has indeed become faster on the new GPUs.

Lighting. One example of things that unexpectedly turned out to be a problem is two-sided lighting. Point based rendering is not a truly drop-in replacement in this case, as the texturing and lighting functions are not fully compatible. Two sided lighting turned out not to work at all (it turns out that the OpenGL specifications did not state that it should), and using texture coordinates with OpenGL point primitives was much slower than using the glColor() functions.

Asynchronous Preprocessing. To compensate for the delay introduced by the preprocessing step, our implementation supported an asynchronous mode, where a part of the model could already be examined while the rest of the system was still being read in. However, our partner reported this as too confusing regarding the true extents of the data set, and this feature was subsequently disabled in favor of a more traditional progress bar display during preprocessing.

Given that new generations of graphics cards are becoming available every few months, some of these issues can now probably be adressed differently. The problem of two sided lighting for point samples could probably be handled with a vertex shader that flips the normal vector accordingly, and elevation color codes might be extremely fast and flexible with fragment shaders.

Even the elevation data itself could be streamlined by using vertex shaders for displacement, drastically reducing the application's memory footprint since a simple 2-byte $m * n$ texture would suffice compared to the currently required 6 bytes per sample.

However, these features were not available in the original target hardware, and therefore not considered during the project lifetime. This shows that it is actually reasonable to reach higher, and develop at least for the current top-end hardware, as its performance and features will very likely have become standard or even been surpassed once the project is ready to be delivered.

3.1. Summary

In summary, this project presented an informative learning opportunity for working with "sidestream" (not as widely used) OpenGL features, and on how to handle cooperation with partners that are a long distance away. Some of the key lessons learned are:

- Point based and polygonal rendering can be used very easily in a mixed environment. Unlike impostor rendering, point based structures produce correct values in the z-Buffer and interpenetration works exactly as expected.
- However, support for point based rendering seems to be much less optimized in current graphics hardware and/or drivers than polygonal rendering. This is hardly surprising, but an (unrelated) test showed that line drawing performed at exactly the same speeds on several generations of graphics cards. So, performance advances in graphics hardware are typically focused on popular features, whereas other data paths are much less optimized (or do not gain any speedup at all). Also, the OpenGL standard leaves some room for interpretation.
- Polygonal performance is also much closer to point based rendering for continuous surfaces, since the renderer can make extensive use of triangle strips, which brings down the bandwidth needed to render n triangles from $3n$ vertices to $2 + n$. There is still some internal overhead for rendering triangles that can be avoided when rendering points (fragment generation, for example), but for this type of data memory and bus bandwidth usage is actually similar for point based and polygonal rendering.
- Close and precise communications is very important, especially with partners that are met only infrequently. Some of the problems we ran into were purely due to misunderstandings or delays in communication. Also, assumptions are bad – always confirm them before making decisions. This also includes a thorough understanding of the data sets involved, including its "worst case" parameters.
- Priorities and competences should be assigned as soon as possible. It is imperative to understand what is really important for the partners, and also to reliably confirm which parts of the system can be adapted freely and what has to be kept strictly as is. This includes user interface (GUI and interaction handling) design issues as

well as module interfaces and semantics, process timings and file formats.

- Whenever possible, goals should be quantifiable. Setting precise (numerical) targets, such as “between 10 and 15 frames per second”, is a much more reliable and consistent goal than “interactive performance” (which is a much more subjective rating). Obviously some features (such as visual quality) will always remain subjective, but the more goals can be numerically verified to be met, the better.

Acknowledgements

This work has been done in the scope of the basic research on virtual environments (`http://www.VRVis.at/vr/`) at the VRVis Research Center in Vienna, Austria (`http://www.VRVis.at/`), which is funded by the Austrian governmental research program K plus.

References

[1] Charles Bloom. View independent progressive meshes VIPM. web page. http://www.cbloom.com/3d/techdocs/vipm.txt.

[2] Baoquan Chen and Xuan Nguyen. POP: A hybrid point and polygon rendering system for large data. In Thomas Ertl, Ken Joy, and Amitabh Varshney, editors, *Proceedings Visualization 2001*, pages 45–52. IEEE Computer Society Technical Committee on Visualization and Graphics Executive Committee, 2001.

[3] Jonathan D. Cohen, Daniel G. Aliaga, and Weiqiang Zhang. Hybrid simplification: Combining multi-resolution polygon and point rendering. In Thomas Ertl, Ken Joy, and Amitabh Varshney, editors, *Proceedings Visualization 2001*, pages 37–44. IEEE Computer Society Technical Committee on Visualization and Graphics Executive Committee, 2001.

[4] Willem H. de Boer. Fast terrain rendering using geometrical mipmapping, October 2000. flipCode.com Tutorial.

[5] Sim Dietrich. Elevation maps. NVIDIA white papert, 2000. http://developer.nvidia.com/view.asp?IO=Elevation_Maps_Paper.

[6] Mark A. Duchaineau, Murray Wolinsky, David E. Sigeti, Mark C. Miller, Charles Aldrich, and Mark B. Mineev-Weinstein. ROAMing terrain: Real-time optimally adapting meshes. In *IEEE Visualization ’97*, October 1997.

[7] Hugues Hoppe. View-dependent refinement of progressive meshes. In Turner Whitted, editor, *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 189–198. ACM SIGGRAPH, Addison Wesley, August 1997. ISBN 0-89791-896-7.

[8] Hanspeter Pfister, Matthias Zwicker, Jeroen van Baar, and Markus Gross. Surfels: Surface elements as rendering primitives. In Kurt Akeley, editor, *Siggraph 2000, Computer Graphics Proceedings*, Annual Conference Series, pages 335–342. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.

[9] Szymon Rusinkiewicz and Marc Levoy. QSplat: A multiresolution point rendering system for large meshes. In Kurt Akeley, editor, *Siggraph 2000, Computer Graphics Proceedings*, Annual Conference Series, pages 343–352. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.

[10] Gernot Schaufler. Per-object image warping with layered impostors. In George Drettakis and Nelson Max, editors, *Rendering Techniques ’98*, Eurographics, pages 145–156. Springer-Verlag Wien New York, 1998.

[11] Thatcher Ulrich. Rendering massive terrains using chunked level of detail control. In *SIGGRAPH 2002 Course Notes CD-ROM*. Association for Computing Machinery, ACM SIGGRAPH, August 2002. Course 35.