Streaming QSplat: A Viewer for Networked Visualization of Large, Dense Models

Szymon Rusinkiewicz
Marc Levoy
Stanford University [†]

Abstract

Steady growth in the speeds of network links and graphics accelerator cards has brought increasing interest in streaming transmission of three-dimensional data sets. We demonstrate how streaming visualization can be made practical for data sets containing hundreds of millions of samples. Our system is based on QSplat, a multiresolution rendering system for dense polygon meshes that employs a bounding sphere hierarchy data structure and splat rendering. We show how to incorporate view-dependent progressive transmission into QSplat, by having the client request visible portions of the model in order from coarse to fine resolution. In addition, we investigate interaction techniques for improving the effectiveness of streaming data visualization. In particular, we explore color-coding streamed data by resolution, examine the order in which data should be transmitted in order to minimize visual distraction, and propose tools for giving the user fine control over download order.

Categories and Subject Descriptors: I.3.2 [Computer Graphics]: Graphics Systems – Distributed / Network Graphics; I.3.3 [Computer Graphics]: Picture/Image Generation – Viewing Algorithms; I.3.6 [Computer Graphics]: Methodology and Techniques – Interaction techniques.

Keywords: Rendering systems, Level of detail algorithms, Streaming, Progressive transmission.

1 Introduction

In the past, interactive 3D content has not had a large presence on the World Wide Web. Despite the availability of standards such as VRML, 3D models have been constrained to specialized niches because of long download times and poor interactive performance. Today, however, the availability of low-cost, high-performance graphics cards and the introduction of high-speed residential Internet connectivity are making it practical to include 3D models as important components of web sites.

Given the presently available network bandwidths, however, it would not be feasible to use large 3D models if those models had to be downloaded entirely before they could be viewed. The size of currently attainable models, however, is increasing rapidly because of the availability of devices and algorithms for scanning large objects at high resolution; meshes of several hundred million polygons can now be produced [Levoy 00]. For models of this size, the only practical way of allowing remote download and visualization is to stream the data as it is needed, and to permit the viewer to look at and interact with partially-downloaded models.

The currently dominant strategy for streaming large polygon meshes is to simplify them and transmit them progressively. However, most of these algorithms are impractical for models larger than a few million polygons. QSplat is a system for representing and rendering large meshes that takes advantage of the fact that connectivity information may be dropped for large, densely and regularly sampled meshes, and points can be used as the rendering primitive [Rusinkiewicz 00]. QSplat combines this splat-based renderer with a multiresolution representation based on a bounding sphere hierarchy to allow large 3D models to be displayed at interactive rates.

In this paper we introduce a streaming version of QSplat, allowing large models to be progressively streamed across a network of limited bandwidth. The system retains the advantages of QSplat, such as low preprocessing costs and high rendering performance, but adds view-dependent network streaming of geometry. The extension to streaming is based on the fact that we can terminate the recursion of our data structure at any time during rendering if we find that portions of the hierarchy are not yet present on the client; a low-resolution model is rendered, and the missing nodes are requested from the server. Thus, portions of the model are downloaded as the user looks at them.

Though progressive transmission of 3D data has been explored before, most of the effort has focused on either high-speed streaming from a local disk or low-speed streaming across a slow connection. In the former case, the available bandwidth is often adequate to mask the presence of streaming, and research has concentrated on techniques such as prefetching that attempt to hide the fact that data is being read progressively at all. With low-speed links, attention has mostly focused on achieving good approximations to the final model while transmitting as little data as possible, regardless of the required CPU time.

In contrast to the high- and low-bandwidth extremes, comparatively little effort has been devoted to the user interaction issues that become relevant at intermediate speeds (e.g. a few hundred kbps, which is becoming an increasingly common rate for residential Internet connectivity). These speeds are high enough that it is often not worthwhile to implement expensive compression and optimization techniques, but are sufficiently low that there is little hope of concealing the presence of streaming for large models. Thus, we ac-

[†] Stanford Computer Graphics Lab
Gates Building 3B
Stanford University
Stanford, CA 94305
{smr,levoy}@graphics.stanford.edu

cept that the streaming process will be visible to the user, and focus on designing a user interface that lets the user know how much data is present, minimizes the visual distraction due to streaming, and gives the user fine control over the streaming process.

We first examine some previous systems that have been used for 3D streaming and large data visualization. Next, we review the basics of the QSplat data structure and rendering algorithm, and describe the extensions that must be made to support 3D streaming. In Section 4, we discuss interaction issues that influence the design of the streaming QSplat user interface, focusing on how to aid the user in interpreting the data and understanding and controlling the streaming process. Finally, we present future work that could be done to broaden the applicability of streaming QSplat.

2 Previous Work

Several schemes have been proposed for transmitting 3D data across a network. The simplest ones rely on transmitting a full polygonal model (either directly [VRML 97] or in a compressed format [Taubin 98]), and therefore require the entire model to be transmitted before the user can look at it.

More sophisticated systems transmit low-resolution data first, so the user can begin to interact with the model, then progressively stream higher-resolution data, time permitting [Gueziec 99]. The progressive mesh framework [Hoppe 96] represents a mesh as a simple "base mesh" plus a series of refinements to the mesh based on a vertex split primitive. Progressive meshes, therefore, are well-suited to streaming [Prince 00], especially with the addition of compression [Pajarola 00].

Corrections to a base mesh may also be encoded using wavelets, as was first proposed in multiresolution analysis [Eck 95]. The model may then be transmitted by sending the base mesh and streaming the wavelet coefficients in order of magnitude [Khodakovsky 00]. One advantage of this approach, explored by Certain et. al., is that color and geometry wavelets may be streamed independently [Certain 96].

Commercial systems incorporating some of these algorithms are beginning to appear. MetaStream's MTS products, for example, represent geometry as a base mesh together with a series of vertex split operations [Abadjev 99], similar in spirit to progressive meshes. Other products are available that stream polygonal models (e.g. [RealityWave]) or voxelized volumetric data (e.g. [Octree]).

Many systems for architectural walkthrough and terrain flythrough are designed to work with scenes larger than the available memory [Funkhouser 92, Funkhouser 96, Aliaga 99]. In order to achieve high-quality renderings, they explicitly manage the way data is transferred between memory and disk. These systems typically employ the notion of a potentially-visible set (PVS) of data, comprising both currently-visible data and data that may come into view in the near future, given some assumptions about where the user is likely to move and look next. These systems then perform prefetching to ensure that off-screen data is loaded into memory before the user looks at it. Network streaming of potentially-visible sets for such applications has been explored by Cohen-Or and Zadicario [Cohen-Or 98].

Compared with most of the above systems (both research and commercial), our streaming QSplat implementation has higher rendering performance (both because it uses simpler rendering primitives and because it does not require CPU time to be devoted to decompression), requires less preprocessing time, and uses a standard HTTP server rather than a custom streaming server. As we discuss in Section 3.6, this makes QSplat well-suited for streaming large models across networks of moderate bandwidths. Our system, however, is not as bandwidth-efficient as some systems that incorporate more sophisticated geometric compression. In addition, since it uses splats as the rendering primitive, it will have lower visual quality than polygon-based systems for certain kinds of scenes.

3 Streaming QSplat

Our system for network streaming of large 3D meshes is based on QSplat, a multiresolution point rendering system. We first review the QSplat data structure and rendering algorithm, then describe the additions needed to support view-dependent transmission. We also consider the advantages and disadvantages of basing a streaming system on QSplat.

3.1 Data Structure and Rendering Algorithm

QSplat uses a hierarchical bounding sphere data structure for visibility culling, level-of-detail control, and rendering. Each node in this tree contains:

- The sphere's position and radius, quantized relative to the position and radius of the node's parent.
- A per-vertex normal, used for lighting calculations.
- The width of a normal cone, used together with the normal for hierarchical backface culling.
- Optionally, a per-vertex color.

Each node is 4 bytes without color, 6 bytes with per-vertex color. The layout of each node is shown in Figure 1.

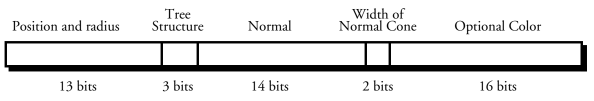
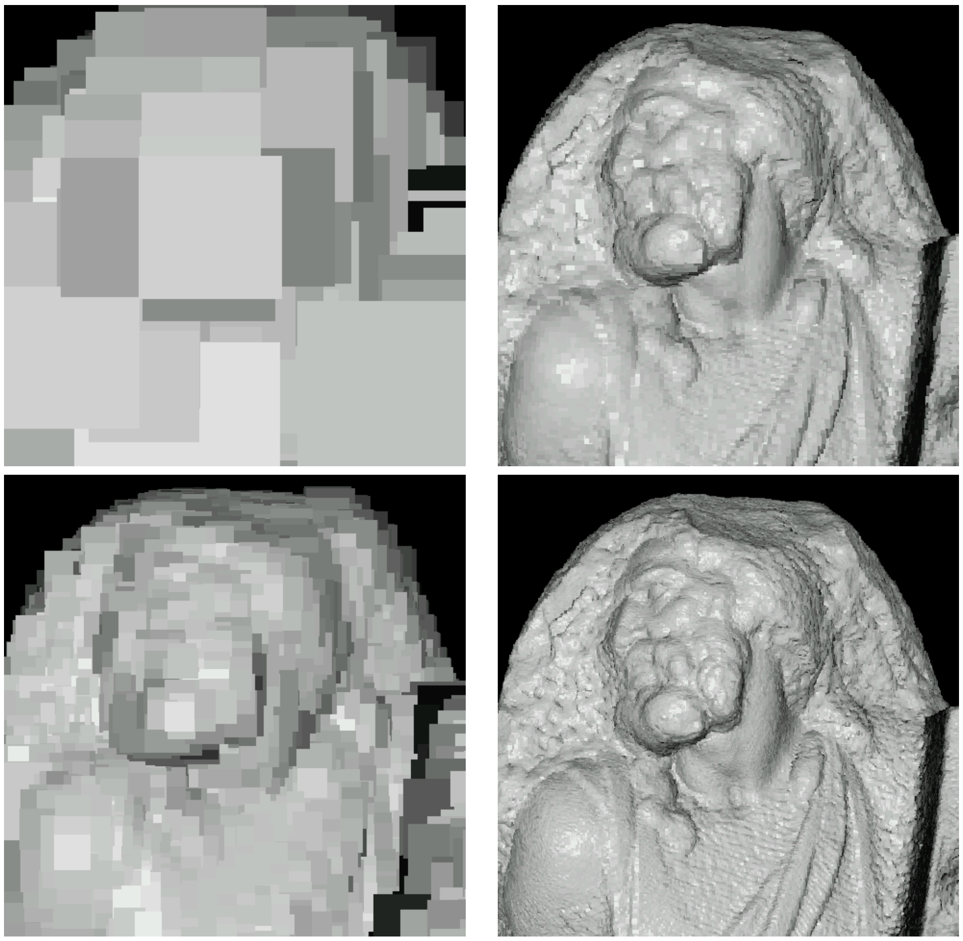


Figure 1: QSplat node layout.

The hierarchy is constructed as a preprocess from an input polygonal mesh, with each leaf node of the tree corresponding to a vertex of the original mesh. The connectivity of the original mesh is discarded, but in order to guarantee hole-free renderings we are careful to make each leaf sphere large enough to touch its neighbors.

During rendering, we recursively traverse the hierarchy in depth-first order. At each non-leaf node, we first determine whether the sphere is entirely off-screen (by projecting the sphere onto the viewing plane) or backfacing (by testing whether a cone defined by the per-vertex normal and cone width faces away from the viewer). If so, we can ignore the sphere and its children, thus performing visibility culling. If the subtree is at least partially visible, we compare the projected screen size of the sphere to a cutoff value. If the sphere is larger than our threshold, we recurse. If the sphere is smaller than the threshold, or if we reach a leaf node, we draw a splat on the screen with position and size determined by the location and radius of the sphere. The recursion cutoff is adjusted in a feedback loop (based on the time to render the previous frame) in order to maintain a user-selected frame rate. Once the user stops moving the mouse, we redraw the model with progressively smaller thresholds, until we either descend to the leaves of the tree or reach a splat size of one pixel.

QSplat directly uses the compressed representation during rendering, thus requiring no extra time or memory for decompression. In addition, this means that the on-disk and in-memory representations of a model are identical, so we can memory map the file from disk. This places the burden of working set management on the operating system, which simplifies the implementation. Because we do not have explicit control over when data is loaded, however, QSplat may experience glitches in the frame rate when new sections of the model are



a) Appearance of the model immediately after the start of streaming.

b) 1 second after (a).

c) 10 seconds after (a).

d) 60 seconds after (a).

Figure 2: View-dependent streaming data transmission of a 130 million sample model over a network limited to 384 kbps. See also Figure 4 (color plate).

seen for the first time. These glitches, moreover, become more pronounced when the model is being loaded not from a local disk but over a networked filesystem. Though the performance is acceptable given a local server and a fast network, performance becomes much worse as network bandwidth drops and latency increases. Thus, as we will see later, a network streaming implementation of QSplat must take control of its data management, and make explicit requests for the data it needs. Though the changes required to support this data management prove modest, the result is a system that effectively allows remote visualization of large data sets, and is flexible enough to permit an exploration of some of the user-interface issues surrounding streaming 3D data visualization.

3.2 Network Streaming

The key to network streaming of QSplat models is the observation that during rendering we can terminate recursive decent of our hierarchical representation at any time. In place of missing geometry, QSplat displays a splat corresponding to the parent node in the hierarchy. In the system of [Rusinkiewicz 00], recursion is terminated under two conditions: if the children of a given node are smaller than a threshold, or if we reach a leaf node. To accommodate streaming, we need to add one additional condition: we stop recursion if the children of a given node have not yet been transmitted from the server to the client. Thus, with low run-time cost we transparently accommodate the presence or absence on the client of various portions of the hierarchy, including the possibility of having different resolutions of data present throughout the model. Note that we still perform the usual feedback-driven frame rate control when the user is dragging the mouse, so although the frame rate may be higher than the user setting if not enough data is present, it will not drop lower than requested. Figure 2 illustrates the results of streaming transmission.

To allow for network streaming, therefore, we need three components:

- A bitmask indicating which regions of the model are present on the client.
- A prioritized request queue containing a set of regions of the model that the client would like to receive, given the current camera position.
- A separate thread on the client that makes requests to a server, listens for responses, and updates the tree data structure and availability mask as data is received.

Given these, the rendering algorithm is shown in Figure 3.

3.3 Availability Mask

In order to perform rendering correctly, we must have a data structure that maintains information about which portions of the model have been received from the server. For maximum flexibility, we would like to have the mask as fine-grained as possible – ideally, we would store availability information at the granularity of a single node of the tree – so that we can download precisely the areas of the model in which we are interested. For efficiency of download and to minimize the memory spent on the mask, however, we must use a larger granularity.

```
TraverseHierarchy(node)
{
  if (node not visible)
    skip this branch of the tree
  else if (node is a leaf node)
    draw a splat
  else if (benefit of recursing further is too low)
    draw a splat
  else if (any child is not present)
    draw a splat
    RequestQueue.insert(children(node), priority)
  else
    for each child in children(node)
      TraverseHierarchy(child)
}

DrawFrame(model)
{
  RequestQueue.clear
  TraverseHierarchy(model.root)
  if (not RequestQueue.empty)
    n ← (estimated net bandwidth) / (frame rate)
    for i ← 1 .. n
      SendRequest(RequestQueue.top)
      RequestQueue.pop()
}
```

Figure 3: Streaming QSplat algorithm for rendering and progressive download.

In our system, we represent availability at the granularity of fixed-size (typically 1 kilobyte) blocks. In addition, to simplify the bookkeeping, we increase the storage per chunk to two bits, so that we can represent four states for each block: not present, desired (i.e., present in the request queue), requested from the server, and present.

3.4 Request Queue

As we traverse the hierarchy during rendering and encounter chunks that are not present on the client, we push requests for these blocks onto a priority queue (implemented as a max-heap). As we will see in Section 4.2, the priority for a node is determined from the projected screen size and position of that node's parent (which triggered the request for the node). The priority of a chunk is the highest priority of all nodes within that chunk.

The request queue is cleared before every rendered frame. This ensures that:

- The request queue never gets too large, since its size will be proportional to the number of rendered nodes, rather than the total number of nodes in the model.
- A chunk that moves out of the field of view will be dropped from the request queue, preventing the system from wasting time on downloading sections of the model that are no longer relevant to the user's viewpoint.

If the request queue is ever empty after rendering a frame, meaning that all currently-visible data was already present on the client, we first download data in the vicinity of the viewpoint, then revert to downloading any remaining parts of the model in order from the root of the tree to the leaves.

3.5 Network Communication

The streaming QSplat client uses a separate thread to make requests from the server and listen for responses. The number of requests to make per frame is based on an estimate of the network bandwidth, so that there are never too many outstanding requests. The data requested from the server consists of ranges of the original file; thus, the server need not have any special knowledge of the QSplat file format. In our implementation of the streaming QSplat client we have chosen to use the HTTP/1.1 protocol (including the byte-range and persistent connection features [Fielding 97]) to issue requests, so we may stream models from any standard web server (e.g. Apache); a separate streaming server is not required.

3.6 Discussion

Let us now examine some of the advantages and disadvantages involved in using QSplat, as compared to traditional polygonal representations, as the basis of a network streaming system.

Suitability of QSplat for Network Streaming: Streaming QSplat retains most of the advantages and disadvantages of QSplat in its suitability for representing various classes of geometric models. In particular, streaming QSplat will work best for large, dense models containing relatively regular, uniformly-spaced data points (e.g. as produced by VRIP [Curless 96] and marching cubes [Cline 88]) and high geometric detail at fine scales. In contrast, a QSplat representation of a model with large flat regions, subtle curves, or sharp corners will not look as good as a polygonal or spline model of equal size. Moreover, a low-resolution version of any model, when rendered with splats, will contain visible artifacts of the splat shape.

A second property of QSplat that becomes useful for streaming is the fact that parts of a model may be transmitted in any order, subject only to the constraint that parent nodes must be transmitted before children. This makes it easy to incorporate various strategies for choosing the order in which parts of the model are transmitted. By contrast, some geometric compression techniques require that the model be transmitted in a particular order, since they represent vertex positions and connectivity by encoding deltas along a particular path through the vertices of the model.

Compressed Data Size: One difference between QSplat and most other geometric compression techniques is that QSplat uses the same data representation on disk and in memory, thus not requiring extra time or space for decompression. In designing streaming QSplat, we have chosen to use this same data representation for network transmission as well. By eliminating the need to encode and decode a compressed format, we simplify the requirements for the network server, and we minimize run-time overhead in the client when using moderate- or high-speed links.

The tradeoff is that QSplat may not be as bandwidth-efficient as algorithms that incorporate more sophisticated geometric compression. As an example, QSplat requires per-vertex normals to be stored and transmitted explicitly. Although QSplat's representation of normals is reasonably efficient (14 bits per node), normals could instead be computed by the client from transmitted polygon geometry, thereby saving network bandwidth. (QSplat could not use this approach, since it does not use polygons.)

4 Interaction Techniques for 3D Streaming

As mentioned earlier, we have chosen to focus on the user interaction techniques that become relevant to streaming at moderate network bandwidths (e.g. a few hundred kbps), rather than on the low- or high-bandwidth extremes. Our motivation for this is the observation that, after remaining static for many years, typical network speeds appear to be rising, especially in residential settings. These speeds, however, are still not sufficiently high that streaming becomes invisible. Therefore, since the user will be able to observe the streaming, we explore color coding to communicate the relative resolution of data present at various points. In addition, we investigate several options for the order in which to stream data, including a user-controlled "magnifying glass" tool that directly controls download order. Finally, we examine the role of prefetching at these speeds.

4.1 Color-coding by Resolution

In a view-dependent streaming system such as ours, the model may, because of previous camera movements, have different sections available at different resolutions. Similar situations arise in other multiresolution rendering systems, such as the hierarchical splatting of Laur and Hanrahan [Laur 91]. When looking at such models, it is possible to mistake low-resolution splats for plausible object geometry. Thus, users need visual cues that allow them to distinguish a transition between areas of different resolutions from an actual feature of the object. To accomplish this, streaming QSplat provides an optional user-selected color coding of the downloaded data, so that areas of different resolutions appear in different colors. This provides visual feedback for the user about the resolution at which various areas of the model are being rendered, and which areas are still being downloaded. The color coding is used in the figures in the color plate.

4.2 Streaming Order

When the camera is positioned to look at some portion of the model that has not been seen before, we must choose the order in which to stream the nodes within the view frustum. This reduces to defining a priority function for a given node, since the position of nodes within the request queue determines the order in which they will be downloaded. There are several possibilities for this ordering:

1. We may base the priority function on the level of a node within the bounding sphere hierarchy. This will have the effect of downloading (a portion of) the tree in order from root to leaves, such that all nodes at any given level of the tree will be downloaded before we start on the next level within the tree. This has the benefit of being simple to compute, but has the drawback that it may assign the same weight to differently-sized pieces of the model. As a result, nodes downloaded at the same time may have different sizes.
2. We may prioritize nodes by size (i.e. sphere radius) in object space. This is also simple to compute, but has a drawback similar to option 1 because it may assign the same weight to equally-sized pieces of the model regardless of their distance to the viewer. Thus, far-away nodes occupying a relatively small area on the screen may be assigned the same priority as close-by nodes that appear larger on the screen.
3. To remedy the above problem, we may assign priorities based on a node's projected screen size. With this priority function, nodes that appear the same size for a given camera position will be downloaded at roughly the same time. This exposes a second problem, however: the (x, y) screen location at which data is being streamed will be constantly varying in a seemingly-random fashion. This proves to be somewhat distracting for the user, since data appears to be changing at unpredictable locations on the screen.
4. A potential fix for the above problem is to stream based on the screen-space y coordinate. This refines the model in a single pass from the top of the screen to the bottom, which appears more ordered and thus less objectionable for the user. This approach, however, has the drawback that the single pass over the screen is slow, since the user must wait for full-resolution data to be downloaded at each y location.
5. The advantages of approaches 3 and 4 can be combined by prioritizing the nodes such that we perform a number of top-to-bottom sweeps over the data. Each of these passes has its own screen-space cutoff for node size, and we download only the nodes larger than this cutoff. A similar strategy has been used for progressive download of images, e.g. progressive GIFs. A priority function that implements this behavior is

$$\begin{aligned} \mathit{Priority}(n) \;=\;& \lceil \log_k \mathit{Splatsize}(n.\mathit{radius}) \rceil \cdot 1000 \;+ \\ & \mathit{Project}(n.\mathit{center}).y \end{aligned}$$

   This bases the priority on a (logarithmically) quantized version of the node's screen size, with a secondary ordering based on the screen y coordinate. The base of the logarithm, k, determines how much data is downloaded per pass. We have experimentally determined that using $k = \sqrt{2}$ produces acceptable results, roughly doubling the number of downloaded nodes on each pass.

We have chosen to use the algorithm described in option 5 in our implementation. The effect of using this priority function is demonstrated in Figure 5 (color plate) and in the accompanying video. We believe that it offers a good compromise of downloading the most relevant data as soon as possible while minimizing visual distraction.

4.3 Magnifying Glass

For certain model inspection tasks it is desirable to have finer-grained control over download order than the above algorithm provides. For example, in a large, complex model there may be a feature of interest that a user wishes to examine at the highest possible level of detail. Given only the above algorithm, the only way to accomplish this quickly (i.e., without waiting for the entire screen to be refined to the desired resolution) would be to zoom in on the given feature. Sometimes, however, it is desirable to see the feature of interest in the context of the surrounding geometry, for which lower resolutions are often sufficient. Under such circumstances, we can introduce tools that allow the user to boost the priority of certain points on the model or regions of the screen. As an example, we have implemented a "magnifying glass" tool that temporarily increases the priority of a region of the screen (the magnifying glass metaphor in user interfaces has been explored before, e.g. in the work on "Magic Lenses" by Bier et. al. [Bier 93]). The magnifying glass may be dragged around to permit the user to focus on any locations on the screen. The effect is illustrated in Figure 6 (color plate). Note that color coding is especially useful in this case to illustrate what sections of the model are present at what resolution.

4.4 Prefetching

Architectural walkthrough and terrain rendering systems often use prefetching to improve the quality of renderings and to avoid latencies in the availability of high-resolution data when the user moves to new parts of the model. We have implemented a prefetching algorithm for streaming QSplat that places nodes slightly outside the view frustum onto the request queue with a low priority. After some experimentation, however, we have found that using prefetching does not improve the quality of interaction with QSplat to the extent it does with architectural walkthrough and terrain rendering systems. The chief causes of this are:

- In contrast with walkthrough systems, QSplat is best suited to visualizing objects, not environments. Because of this, and because of the trackball interface used by QSplat (as compared to a "flythrough" interface), the camera movements during interaction with QSplat tend to be less predictable than in walkthrough systems. This results in larger potentially-visible sets, so resources devoted to prefetching are spread out over a larger area of the model.

- Systems in which prefetch is most effective stream data from disk, which can be done at a sufficiently high rate that they successfully create the illusion that high-resolution data is always available. In contrast, we assume a network link with significantly lower bandwidth. Coupled with the fact that QSplat draws more primitives per frame than most comparable polygon-based systems, we can not hope to maintain the illusion that highest-resolution data is always available.
- It is difficult to determine an acceptable value for the relative priority to be assigned to on-screen and off-screen data. If not enough weight is given to on-screen data, the refinement rate of the visible portion of the model slows down to an undesirable degree. If the off-screen data is not weighted enough, there is little visible difference compared to not performing any prefetching. This is because the off-screen data is downloaded at a slow rate compared to the speed at which it will be downloaded as soon as it comes into view.

Because of the above factors, it is difficult to find circumstances under which it is clearly useful to perform prefetching in QSplat. In fact, after some experimentation we have decided to abandon prefetching entirely, and only fetch off-screen data once the entire viewport is fully refined (i.e., to a node size of one pixel), at which point the system is idle and might as well spend its time prefetching.

5 Conclusions and Future Work

We have demonstrated a system for view-dependent network streaming and interactive display of large, complex 3D models. The implementation works with a standard web server, incurs low run-time overhead on the client, and takes advantage of the low preprocessing costs, compact storage, and real-time rendering capabilities of QSplat.

As mentioned earlier, the per-node storage requirements of QSplat are higher than those achievable by some other geometric compression algorithms, largely because QSplat must store per-vertex normals. Although it would not be practical to eliminate QSplat's per-vertex normals completely, their storage cost could be considerably reduced in cases in which low per-primitive cost is critical (e.g. low-speed modem links). By combining incremental encoding of normals (i.e., encoding the normal of each node as a displacement relative to the normal of its parent node) with an entropy coding technique (e.g. Huffman coding [Huffman 52]), we could reduce the storage requirements for a normal from the present 14 bits to perhaps 3-5 bits per node. In addition, using Huffman coding for vertex position, sphere radius, and color could further reduce the per-node storage requirements of QSplat, to be competitive with state-of-the-art polygonal compression techniques. Adding this extra compression, however, would require devoting CPU time to decompressing the network-streamed data before it could be rendered, thus decreasing rendering performance (especially on a single-CPU machine) and increasing the latency with which newly-downloaded blocks could be used in rendering.

A second improvement would be to eliminate the need for temporary storage on the client. Because the present implementation is based closely on QSplat, the client requires a local temporary file equal in size to the size of the model. This file is memory mapped, and blocks are written to the file as they are received. For widest applicability, such as a web browser plugin, the client machine should not be required to have this much free disk space (which for a model of hundreds of millions of samples may approach a gigabyte). The temporary file could be eliminated by adding an additional level of indirection to the mapping from the logical position of a section of a model to physical location in memory. This extra pointer would also permit sections of the model to be discarded in an LRU fashion, to limit total memory usage. For certain systems, the virual memory system can provide the same capabilities.

References

[**Abadjev 99**] Abadjev, V., del Rosario, M., Lebedev, A., Migdal, A., and Paskhaver, V. "MetaStream," *Proc. VRML*, 1999.

[**Aliaga 99**] Aliaga, D., Cohen, J., Wilson, A., Baker, E., Zhang, H., Erikson, C., Hoff, K., Hudson, T., Stuerzlinger, W., Bastos, R., Whitton, M., Brooks, F., and Manocha, D. "MMR: An Interactive Massive Model Rendering System Using Geometric and Image-Based Acceleration," *Proc. Symposium on Interactive 3D Graphics*, 1999.

[**Bier 93**] Bier, E., Stone, M., Pier, K., Buxton, W., and DeRose, T. "Toolglass and Magic Lenses: The See-Through Interface," *Proc. SIGGRAPH*, 1993.

[**Certain 96**] Certain, A., Popović, J, DeRose, T., Duchamp, T., Salesin, D., and Stuetzle, W. "Interactive Multiresolution Surface Viewing," *Proc. SIGGRAPH*, 1996.

[**Cline 88**] Cline, H. E., Lorensen, W. E., Ludke, S., Crawford, C. R., and Teeter, B. C. "Two Algorithms for the Three-Dimensional Reconstruction of Tomograms," *Medical Physics*, Vol. 15, No. 3, 1988.

[**Cohen-Or 98**] Cohen-Or, D. and Zadicario, E. "Visibility Streaming for Network-based Walkthroughs," *Proc. Graphics Interface*, 1998.

[**Curless 96**] Curless, B. and Levoy, M. "A Volumetric Method for Building Complex Models from Range Images," *Proc. SIGGRAPH*, 1996.

[**Eck 95**] Eck, M., DeRose, T., Duchamp, T., Hoppe, H., Lounsbery, M., and Stuetzle, W. "Multiresolution Analysis of Arbitrary Meshes," *Proc. SIGGRAPH*, 1995.

[**Fielding 97**] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Berners-Lee, T. "Hypertext Transfer Protocol – HTTP/1.1," *RFC 2068*, UC Irvine, DEC, MIT/LCS, 1997.

[**Funkhouser 92**] Funkhouser, T., Séquin, C., and Teller, S. "Management of Large Amounts of Data in Interactive Building Walkthroughs," *Proc. Symposium on Interactive 3D Graphics*, 1992.

[**Funkhouser 96**] Funkhouser, T. "Database Management for Interactive Display of Large Architectural Models," *Proc. Graphics Interface*, 1996.

[**Gueziec 99**] Gueziec, A., Taubin, G., Horn, B., and Lazarus, F. "A Framework for Streaming Geometry in VRML," *IEEE Computer Graphics & Applications*, Vol. 19, No. 2, 1999.

[**Hoppe 96**] Hoppe, H. "Progressive Meshes," *Proc. SIGGRAPH*, 1996.

[**Huffman 52**] Huffman, D. "A Method for the Construction of Minimum Redundancy Codes," *Proc. IRE*, Vol. 40, No. 9, 1952.

[**Khodakovsky 00**] Khodakovsky, A., Schröder, P., and Sweldens, W. "Progressive Geometry Compression," *Proc. SIGGRAPH*, 2000.

[**Laur 91**] Laur, D. and Hanrahan, P. "Hierarchical Splatting: A Progressive Refinement Algorithm for Volume Rendering," *Proc. SIGGRAPH*, 1991.

[**Levoy 00**] Levoy, M., Pulli, K., Curless, B., Rusinkiewicz, S., Koller, D., Pereira, L., Ginzton, M., Anderson, S., Davis, J., Ginsberg, J., Shade, J., and Fulk, D. "The Digital Michelangelo Project: 3D Scanning of Large Statues," *Proc. SIGGRAPH*, 2000.

[**Octree**] Octree Corporation, Inc., "Octree Graphics," Web page: http://www.octree.com/graphics.shtml

[**Pajarola 00**] Pajarola, R. and Rossignac, J. "Compressed Progressive Meshes," *IEEE Transactions on Visualization and Computer Graphics*, Vol. 6, No. 1, 2000.

[**Prince 00**] Prince, C. *Progressive Meshes for Large Models of Arbitrary Topology*, M. S. Dissertation, University of Washington, 2000.

[**RealityWave**] RealityWave, Inc., "VizStream Technology," Web page: http://www.realitywave.com/technology.asp

[**Rusinkiewicz 00**] Rusinkiewicz, S. and Levoy, M. "QSplat: A Multiresolution Point Rendering System for Large Meshes," *Proc. SIGGRAPH*, 2000.

[**Taubin 98**] Taubin, G. and Rossignac, J. "Geometric Compression Through Topological Surgery," *ACM Trans. on Graphics*, Vol. 17, No. 2, 1998.

[**VRML 97**] *Virtual Reality Modeling Language*, ISO/IEC Standard 14772-1:1997.

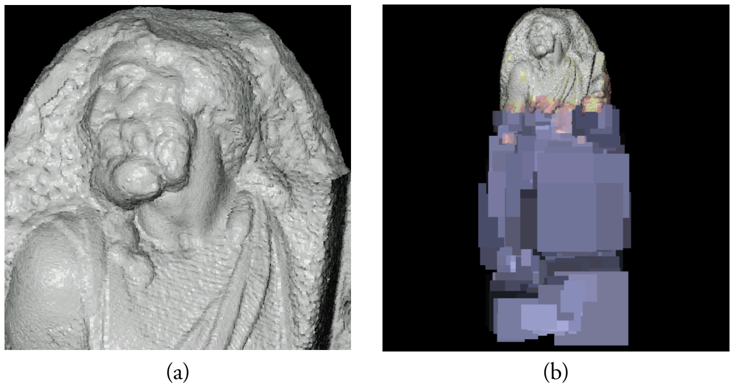


Figure 4: (a) Appearance of a region of the model after 60 seconds of streaming (same as Figure 2d). (b) Appearance of the model immediately after zooming out. Note that high-resolution data has been streamed only in the region on which we were zoomed in.

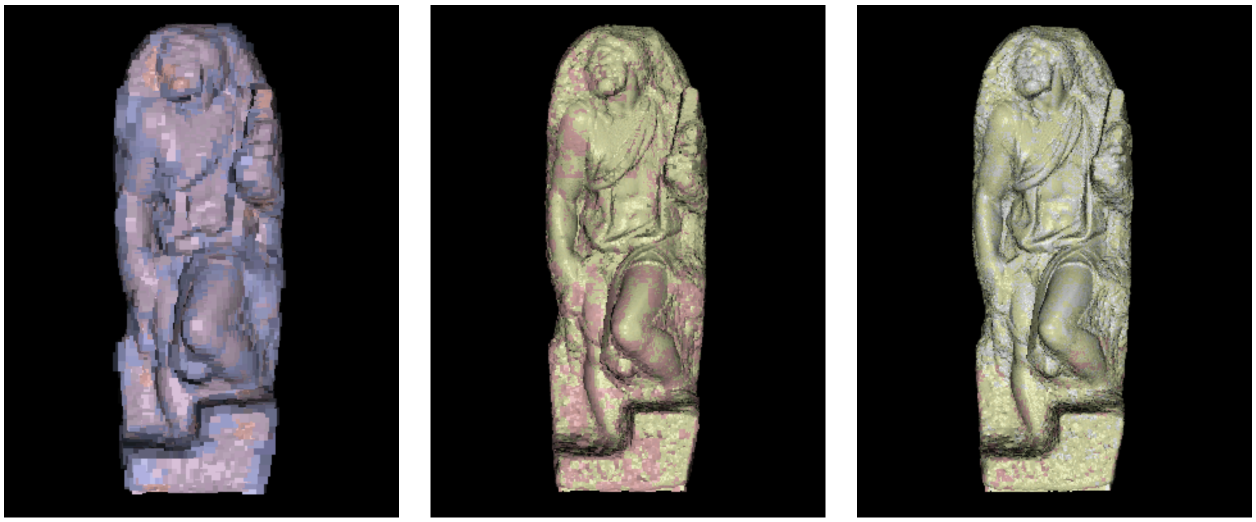


Figure 5: Streaming within a frame is performed in a series of top-to-bottom sweeps that each download all nodes larger than a certain screen-space tolerance. Here, we show the appearance of the model at three points during refinement. Because the refinement order is based on screen-space size, the splats present within a frame tend to be close to each other in size (as long as the viewpoint is not changed).

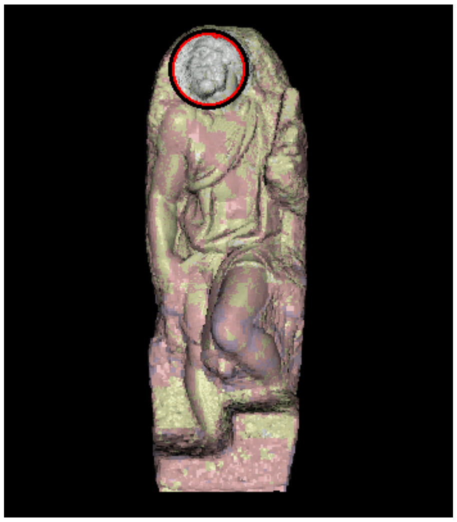


Figure 6: A "magnifying glass" tool is used to provide fine control over download order. As illustrated by the color coding, higher-resolution data has been streamed in the area of the face.

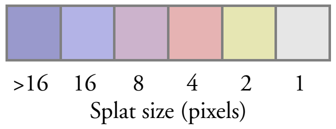


Figure 7: The color coding used by streaming QSplat.