

A Streaming Viewer for Triangular Meshes

Sandeep Sarat
sarat@cs.jhu.edu

*Department of Computer Science
Johns Hopkins University*

Abstract

Increasing size of surface models has led to a great deal of research in progressive multi-resolution representation of these models. Network streaming visualization of large models is particularly interesting. Quite a few existing techniques use a point based rendering system. This approach accelerates rendering but suffers from a degradation in the image quality. To mitigate image degradation, we present a hybrid point and triangle primitive based streaming viewer for triangular meshes. The advantage of using such a mixture of rendering primitives is that, while points can accelerate object display rendering, rendering selective objects using triangles can compensate for the degradation in image quality caused due to the point based rendering.

We have based our system on an existing point based renderer, QSplat, and extended it to incorporate triangle based rendering. The extension achieves a vast improvement in the quality of the rendered image as compared to QSplat. Like QSplat, the system is meant to be deployed in environments with moderate network bandwidths (e.g. LAN).

1 Introduction

Sophistication in scanning techniques has enabled scanning at high resolutions, thereby resulting in a tremendous increase in the size of the generated surface models [LPC⁺00] (typically millions of triangles). Downloading such large models even over presently available network bandwidths is time-consuming. It would be infeasible to use these models if they had to be downloaded entirely before they could be viewed. Therefore, streaming the model data is essential to permit interaction with partially downloaded models.

Many techniques [Hop96, PR00] etc. achieve this goal by employing progressive compression, wherein a coarse approximation of the model is transmitted initially followed by a series of finer corrections. This process allows the user to get an early grasp of the geometry of the model and interact with it. However, a majority of these algorithms are impractical for large meshes since a lot of effort is spent on encoding/decoding connectivity per vertex. Creating and rendering the streamable versions of these models is time-consuming.

Many systems convert the input polygon mesh into a pure point based model [RL00, PZvBG00] and take advantage of this simplicity to speedup rendering. While the point based approach accelerates rendering, image quality suffers due to the decrease in resolution. The lack of connectivity in pure point based systems implies there is a loss of information needed for display interpolation between the points. The degradation of image quality is evident while observing objects closely and is especially severe

for smooth, flat polygon-like objects, as over-sized points prove to be a poor substitute for polygons. The image therefore, appears blocky in either of the two above described scenarios.

We explore the feasibility of using triangles along with points as rendering primitives to alleviate image degradation. We have extended an existing point based renderer, QSplat [RL00], to incorporate triangle based rendering. Streaming QSplat [RL01] is a networked point based viewer which uses the QSplat implementation for representing and rendering large models. QSplat avoids triangles as they are more expensive to render while compared to points. As a result it pays a significant penalty on image quality. In our system, points or triangles are chosen during rendering time based on their screen space projection size. Using this approach, we can improve image quality, besides retaining many desirable properties of the QSplat technique. Although, using triangles as rendering primitives leads to an increase in the size of the model to be streamed, our approach works fairly well over moderate speed networks with hundreds of Kbps bandwidth (e.g LAN) where our system is meant to be deployed.

2 Previous Work

Several schemes have been proposed for representing, rendering and streaming of large surface models.

A majority of the schemes transmit low resolution data first followed by high resolution detail progressively. Eck et.al [EDD⁺95] encode corrections to a base mesh us-

ing wavelets. The model may then be transmitted by sending the base mesh and streaming the wavelet coefficients in order of magnitude [KSS00]. The advantage of this approach is that color and geometry wavelets can be streamed independently of each other.

Hoppe [Hop96] introduced an algorithm for progressive transmission starting from a coarse base mesh and then progressively transmitting a series of refinements of the mesh based on a vertex split primitive. The granularity obtained is optimal (each vertex collapse corresponds to a mesh refinement). The disadvantage is that the number of refinements required is $O(n)$ for a mesh with n vertices and therefore expensive. To reduce the number of refinements in the decimation, Pajarola et.al [PR00] group vertex splits into batches thus amortizing the cost of transmission per refinement.

Progressive Forest Split (PFS) of Taubin et.al [TGHL98] is a compact representation for any manifold mesh. PFS decomposes a given mesh by a low resolution of detail and a sequence of forest split operations. The forest split operation is specified by a forest in the graph of vertices and edges of the mesh, a sequence of simple polygons, and a sequence of vertex displacements. In order to express each forest split operation, forest edges, sequence of simple polygons, and vertex displacements are coded into compressed data streams.

In Bajaj et.al [BPZ99] progressive connectivity transmission is based on the construction of multi-resolution surfaces of triangular meshes. For an arbitrary given mesh, its geometry and connectivity information is first organized by a layering structure. In the light of this structure, vertices are further grouped into contours while triangles are merged into strips or fans. These layers constitute a multi-resolution representation of the model.

Alliez et.al [AD01] introduced a valence-driven decimating approach. Patch tiling and retriangulation were carried out after every decimating conquest (vertex removal) to maintain uniform valency. The advantage of maintaining a regular valency is that it can be used to provide efficient encoding of connectivity.

Traditional methods such as the ones described in Hoppe, Bajaj, Pajarola spend a lot of effort on encoding/decoding geometry and other properties (color, texture etc.) of the mesh. As a result they are not practical for large models. Accordingly, there have been a lot of efforts devoted towards speeding up rendering.

Culling away non-contributive primitives is an effective method to speed up the rendering of the mesh. Kumar et.al

[KM96] present a hierarchical backface culling algorithm based on cones of normals. Other widely used visibility culling techniques are frustum culling to discard primitives not in the viewing frustum, and occlusion culling [ZI97] to discard primitives that are not visible as they are blocked by closer primitives.

Architectural walkthrough [ACW⁺99] systems employ a potentially visible set and a currently visible set of data, to perform fast prefetching. They explicitly manage the way data is transferred between disk and memory. Thus currently offscreen data is prefetched into memory before it is displayed on the screen.

Points are faster to render while compared to polygons. This notion has been exploited by a quite a few systems e.g QSplat [RL00, RL01], Surfel [PZvBG00]. In QSplat connectivity information is ignored on the assumption that large mesh models are dense enough to compensate for the loss of information, thereby treating the vertices as the input mesh. It generates a quad-tree structure from these points building up a hierarchy of nodes. Nodes correspond to a bounding sphere around a point defined by the co-ordinates of the node with a radius specified in the node. Color, normal and the width of the normal cone are also recorded within a node. This preprocessed hierarchical node tree version of the model is stored and used to render the model directly without the need for decoding. During rendering, QSplat recursively traverses the hierarchy in depth first order and splatting non-culled nodes upto a certain threshold screen size, thereby ultimately descending down to the leaves of the tree.

3 Design and Implementation

Our system is based on the QSplat point rendering system. In this section, we describe the phases that are involved in processing and streaming an input triangular data model across the network.

3.1 Preprocessing

The input model to the preprocessing algorithm is any triangular mesh. The main difference in preprocessing while compared to QSplat, is in the construction of leaf nodes in the hierarchical tree representation of the model. QSplat uses the mesh vertices directly for the leaf nodes. We use the triangles of the mesh instead. A bounding sphere is computed for this triangle and is stored in the leaf node along with the normal and color of that triangle (Figure 1). In QSplat, the bounding sphere at the leaf node is

centered around a mesh vertex and its radius computed so that all the bounding spheres of the neighboring vertices overlap (to avoid holes). In our case the bounding sphere is the circumcircle of the input triangle. Therefore, the bounding sphere encloses the triangle and holes are guaranteed to be absent. Computing the circumcircle is expensive, so we do not calculate the circumcircle in all cases (QSplat uses a similar trick). When the triangle is obtuse-angled or right-angled, the diameter of the circle is longest side. Only for acute-angled triangles, we compute the circumcircle. The normal is computed by taking the cross product between two sides of the triangle. The color represents the color of the triangle. Now that we have all the leaf nodes, we use the usual QSplat algorithm to build up the rest of the tree hierarchy. The algorithm builds up the tree by splitting the set of vertices along the longest axis of its bounding box, recursively computing the two subtrees, and finding the bounding sphere of the two children spheres. As the tree is built recursively, per-vertex properties (such as normal and color) at interior nodes are set to the average of these properties in the subtrees. For a more detailed description of the tree building algorithm see QSplat [RL00].



Figure 1: Bounding Sphere logic. For obtuse-angled and right-angled, the diameter of the sphere is the longest side. For an acute-angled, it is the circumcircle.

The number of leaf nodes in the resulting hierarchy tree is the number of triangles present in the model (roughly $2 \cdot n$ for an n vertex mesh). Therefore, a conscious attempt is made to increase the average branching factor (roughly 4) to reduce the number of interior nodes, thereby reducing the storage requirements for the tree model.

While constructing the tree all of the properties at each node are quantized to save space. The color is quantized to 16 bits, the normal to 14 bits and the width of the normal cone to 2 bits. The normal cone is used for back face culling. The position and radius of each sphere is encoded relative to its parent in the tree hierarchy. At leaf nodes we need to write out the co-ordinates of the vertices of the triangle also. Each co-ordinate is quantized to 16 bits.

Position and Radius	Tree Structure	Normal	Width of Normal Cone	Optional Color	Quantized Vertices
13	3	14	2	16	16,16,16 16,16,16 16,16,16

Figure 2: Leaf Node Layout (not to Scale). This contains the extra space for the quantized vertices also.

Position and Radius	Tree Structure	Normal	Width of Normal Cone	Optional Color
13	3	14	2	16

Figure 3: Interior node Layout (not to Scale).

The bits in the tree structure field are used to differentiate the type of node layout.

Once the whole tree is constructed, it is written and stored on disk at the server side.

3.2 Network Streaming

Akin to Streaming QSplat, our system is meant for use in a local area environment where network bandwidths of the order of a few hundreds of Kbps (e.g LAN) are not uncommon. The file containing the hierarchical tree representation of the model is stored on the server. The server has no knowledge about the format of the file. The server fulfills requests for byte ranges from the file using an application level protocol such as HTTP/1.1. Hence, we can use any standard web server (e.g. Apache) for the streaming server. At present, we use a simple streaming server which listens for a request range from a file and transmits the appropriate bytes. As the server is ignorant about the file format, the onus is on the client to request the appropriate byte ranges.

The key to network streaming is the observation that during rendering we can terminate recursive descent of the hierarchy at any time. In place of a missing node, we display a splat corresponding to the parent node in the hierarchy. We stop the recursion if the node has not yet been totally transmitted.

The property of QSplat that is of great use in achieving inexpensive 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. Thus a view dependent streaming is extremely ef-

efficient. 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.

To aid network streaming we use three components.

1. An availability bitmask to indicate the regions of the model that are presently available locally on the client. While maximum flexibility can be achieved by using a bit for every single node, the size of the bitmask can be huge. To save the memory spent on the bitmask, we use a larger granularity. We use two bits to represent the availability of a 4K byte block (if present locally or not and if already requested from the server). We intentionally made the granularity coarser than that of the streaming QSplat implementation because in our case we have a higher number of leaf nodes in the hierarchy tree as we use triangles instead of vertices as leaf nodes and therefore larger tree hierarchy files.
2. A separate thread on the client that makes requests to a server, listens for responses, and updates the hierarchy tree and the availability mask on receiving data. We use a memory mapped file on the client side to store the model hierarchy. This places the burden of working set management on the operating system and simplifies the implementation.
3. A request queue containing the set of regions of the model that the client would like to receive. As we traverse the hierarchy while rendering and encounter chunks that are not present on the client, we push requests onto the request queue. The request queue is shared between the renderer and the requestor thread (described above). The request queue has to be flushed before every rendered frame. This ensures limiting the number of requests and that outdated chunks which are now out of view are not requested. If the request queue is empty after rendering a frame we download other parts of the model which are not yet present locally. This information can be obtained by scanning through the availability bitmask.

3.3 Rendering

We recursively traverse the locally present hierarchical tree and display nodes. Recursion is stopped if a node is not presently available or if a node is smaller than a threshold or if a node is a leaf.

During recursion, we cull away nodes that are outside the viewing frustum. This is performed by testing a node's bounding sphere against the view frustum. If the sphere lies outside the view frustum, recursion is not pursued on that node. If the sphere is entirely inside the view frustum, no further frustum culling is performed on its children. Otherwise, we recurse traversing down the hierarchy tree. Backface culling is also performed to test if a node is back facing. If the normal cone is completely away from the viewer, the node and its subtrees are discarded.

After successfully passing through the culling phase, the screen space projection size (splat size) of the node is evaluated. If the splat size is larger than a threshold, we recurse otherwise we display a splat on the screen. If a node is a leaf, we draw the triangle contained in the leaf node. If a node that has not been culled away and in view is not present locally, a request for the node is pushed into the request queue.

4 Results

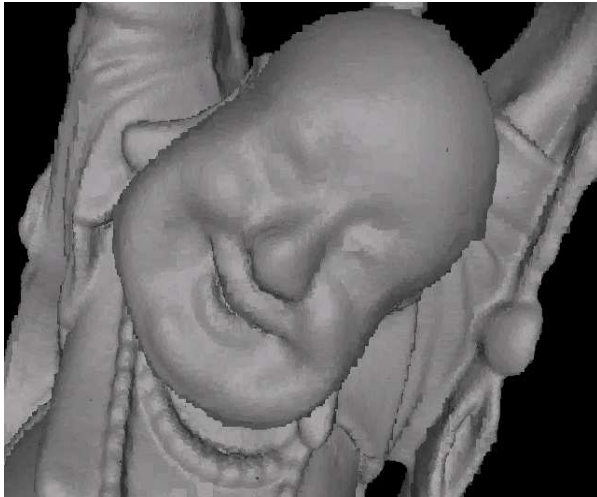
We have implemented our system on a 2.4 GHz Pentium IV with 384 MB memory and an nVIDIA GeForce2 GTS graphic card. We have obtained the QSplat implementation from the authors and have used it extensively in our code. This also helps in provide an apples to apples comparison of our approach with QSplat.

We present the results for two large models - Happy Buddha (1,087,716 triangles, 543,652 vertices) and Hand (654,666 triangles, 327,323 vertices). The average pre-processing times are 20.9 seconds for Buddha and 11.6 seconds for Hand (orders of magnitude faster than most contemporary mesh decimation algorithms). This is a direct consequence of using QSplat for the point rendering system.

4.1 Discussion

As shown in figure 4 we achieve superior image quality while compared to QSplat. This is because QSplat works best for large, dense models containing relatively regular, uniformly-spaced data points. At high zoom, where the spacing of samples is large compared to pixel size, points prove to be a poor substitute for polygons especially near sharp edges and corners. Accordingly, QSplat images become blocky.

Presently, in the leaf node layout, quantized triangle vertex coordinates w.r.t the bounding sphere center are used



QSplat



Our System

Figure 4: A comparison of the images produced by our technique with QSplat. QSplat has 543652 spheres and we have 1087716 spheres. Note that the number of spheres is nearly double, as the number of triangles is roughly twice the number of vertices.

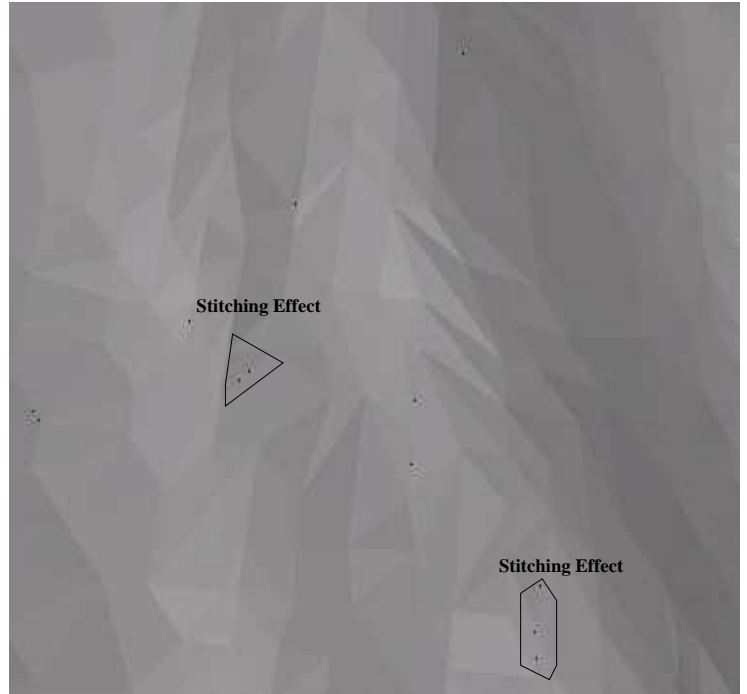


Figure 5: Stitching Effect: At very high zoom - Black dots appear on the boundary between adjacent triangles. This is because the triangle coordinates are quantized w.r.t the leaf node center. So while rendering, adjacent triangles are drawn such that the shared vertices have slightly different coordinates in the two triangles.

to display the triangle. Therefore, a vertex shared by adjacent triangles will have slightly different coordinates when rendering the adjacent triangles. This is due to the floating point error introduced during quantization. This results in a “stitching effect” in the rendered mesh (Figure 6) at very high zoom factors. But for most practical purposes, it is as good as the original model. This effect can be countered by numbering the vertices of the mesh and passing the vertex numbers and co-ordinates separately.



Figure 6: View Dependent Streaming: In (a) we zoomed into Buddha’s Face. In (b) when we zoom out, information corresponding to the face has been streamed, but the lower portions of the model - base, legs etc. are yet to be streamed.

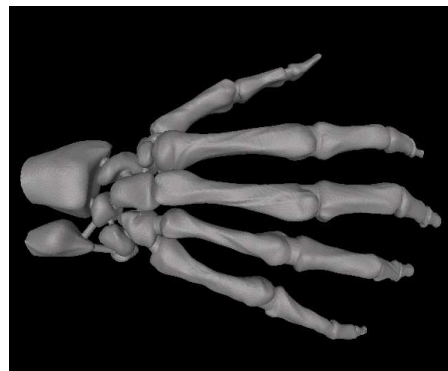
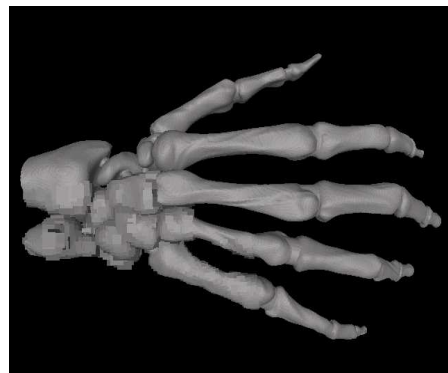
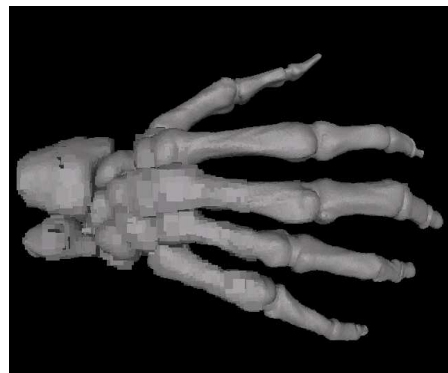
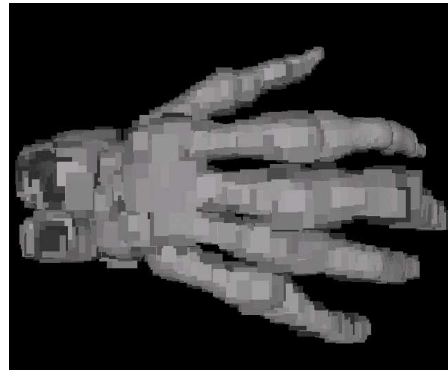


Figure 7: The Hand Model: At various stages of streaming

5 Acknowledgments

Our system is based on the QSplat. We have extensively used code from the QSplat implementation in our code. We would like to thank Budi Purnomo of GLAB, Johns Hopkins University, for providing the ply viewer code and a few ply models.

6 Conclusion and Future Work

We have presented a hybrid point and triangle based rendering system which has been designed for network streaming over moderate network speeds. We take advantage of the simplicity of a point based approach to speed up rendering and the correctness of a triangle based approach to improve image quality during rendering.

At present, during streaming we push the required nodes onto a request queue. There is no specific priority function assigned to a node. As a result, some parts of the model may have a higher resolution while compared to the others. Priority functions based on the node's screen size can be used to provide uniform resolution throughout the model.

The per-node storage requirement in our system is high compared to other traditional compression techniques. We can reduce this by using huffman coding to reduce the storage requirements for the normal from the present 14 bits to perhaps 3-5 bits. Similar huffman coding could be used to reduce the storage requirements of the color and vertex coordinates. Adding extra compression, however, would slow down the rendering performance as CPU time is required for decompression.

As discussed earlier, quantizing the coordinates of the triangle vertices w.r.t the center of the leaf bounding sphere leads to a stitching effect at very high zoom. This can be avoided by using an index to a linear list of quantized vertices [Dee95] in the leaf node.

References

[ACW⁺99] Daniel G. Aliaga, Jon Cohen, Andrew Wilson, Eric Baker, Hansong Zhang, Carl Erikson, Kenneth E. Hoff III, Tom Hudson, Wolfgang Sturzlinger, Rui Bastos, Mary C. Whitton, Frederick P. Brooks Jr., and Dinesh Manocha. MMR: an interactive massive model rendering system using geometric and image-based acceleration. In *Symposium*

on Interactive 3D Graphics, pages 199–206, 1999.

[AD01] Pierre Alliez and Mathieu Desbrun. Progressive compression for lossless transmission of triangle meshes. In Eugene Fiume, editor, *SIGGRAPH 2001, Computer Graphics Proceedings*, pages 195–202. ACM Press / ACM SIGGRAPH, 2001.

[BPZ99] Chandrajit L. Bajaj, Valerio Pascucci, and Guozhong Zhuang. Progressive compression and transmission of arbitrary triangular meshes. In David Ebert, Markus Gross, and Bernd Hamann, editors, *IEEE Visualization '99*, pages 307–316, San Francisco, 1999.

[Dee95] M. Deering. Geometry compression. In *Proc. of SIGGRAPH*, pages 13–20, 1995.

[EDD⁺95] Matthias Eck, Tony DeRose, Tom Duchamp, Hugues Hoppe, Michael Lounsbery, and Werner Stuetzle. Multiresolution analysis of arbitrary meshes. *Computer Graphics*, 29(Annual Conference Series):173–182, 1995.

[Hop96] Hugues Hoppe. Progressive meshes. *Computer Graphics*, 30(Annual Conference Series):99–108, 1996.

[KM96] Subodh Kumar and Dinesh Manocha. Hierarchical back-face culling. Technical Report TR96-014, 15, 1996.

[KSS00] Andrei Khodakovsky, Peter Schröder, and Wim Sweldens. Progressive geometry compression. In Kurt Akeley, editor, *Siggraph 2000, Computer Graphics Proceedings*, pages 271–278. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.

[LPC⁺00] Marc Levoy, Kari Pulli, Brian Curless, Szymon Rusinkiewicz, David Koller, Lucas Pereira, Matt Ginzton, Sean Anderson, James Davis, Jeremy Ginsberg, Jonathan Shade, and Duane Fulk. The digital michelangelo project: 3D scanning of large statues. In Kurt Akeley, editor, *Siggraph 2000, Computer Graphics Proceedings*, pages 131–144. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.

[PR00] R. Pajarola and J. Rossignac. Compressed progressive meshes. *IEEE Transactions*

on Visualization and Computer Graphics,
6(1):79–93, /2000.

- [PZvBG00] 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*, pages 335–342. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.
- [RL00] Szymon Rusinkiewicz and Marc Levoy. QSplat: A multiresolution point rendering system for large meshes. In Kurt Akeley, editor, *Siggraph 2000, Computer Graphics Proceedings*, pages 343–352. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.
- [RL01] Szymon Rusinkiewicz and Marc Levoy. Streaming QSplat: a viewer for networked visualization of large, dense models. In *Symposium on Interactive 3D Graphics*, pages 63–68, 2001.
- [TGHL98] Gabriel Taubin, André Gueziec, William Horn, and Francis Lazarus. Progressive forest split compression. *Computer Graphics*, 32(Annual Conference Series):123–132, 1998.
- [ZI97] Hansong Zhang and Kenneth E. Hoff III. Fast backface culling using normal masks. In *Symposium on Interactive 3D Graphics*, pages 103–106, 189, 1997.