UNSUPERVISED CONVERSION OF 3D MODELS FOR INTERACTIVE METAVERSES

Jeff Terrace[1], Ewen Cheslack-Postava[2], Philip Levis[2], and Michael J. Freedman[1]

[1]Princeton University [2]Stanford University

ABSTRACT

A virtual-world environment becomes a truly engaging platform when users have the ability to insert 3D content into the world. However, arbitrary 3D content is often not optimized for real-time rendering, limiting the ability of clients to display large scenes consisting of hundreds or thousands of objects. We present the design and implementation of an automatic, unsupervised conversion process that transforms 3D content into a format suitable for real-time rendering while minimizing loss of quality. The resulting progressive format includes a base mesh, allowing clients to quickly display the model, and a progressive portion for streaming additional detail as desired. Sirikata, an open virtual world platform, has processed over 700 models using this method.

Index Terms— 3D Models, Content Conditioning, Texture Mapping, 3D Meshes, Metaverses, Virtual Worlds

1. INTRODUCTION

Virtual worlds are three-dimensional graphical environments where people can interact, engage in games, and collaborate. One class of virtual worlds is the "metaverse", such as Second Life, where anyone can add new objects to the world by creating 3D models and writing scripts to control them.

Creating a truly engaging application in a metaverse requires filling it with 3D content. That content can come from a wide variety of sources: modeling packages like Blender or Maya, content repositories such as Google's 3D Warehouse or NASA's 3D Resources, or even less traditional sources such as 3D scanning. Due to their disparate sources, these models have a wide array of characteristics, from million-triangle and hundred-megabyte scanned models to detailed architectural models referencing hundreds of materials and textures. A metaverse platform that accepts arbitrary 3D content must transform any model into a form to be rendered in real-time as just one of thousands of models in a scene.

Metaverses cannot employ the same techniques that other applications, such as games, use to condition and prepare content. Whereas game developers can work closely with artists and filter content through a conditioning pipeline to ensure real-time frame rates, metaverses must handle arbitrary user-provided content. Additionally, game content is shipped to a client before the application runs, but metaverse users expect to drop in a new 3D model at any time (uploading it to virtual world servers "in the cloud") and use it in the world immediately. The metaverse could reject unoptimized content, but such narrow constraints drastically decrease the quantity of readily available content and diminish the usability of the system. Users should not need to care about the intricate, technical details of 3D content.

To realize this vision for interactive metaverses, this paper proposes an *unsupervised* content conditioning pipeline for 3D content. Users can upload to a repository, which automatically transforms the content into a format that ensures good performance in a real-time rendering environment. The model is transformed to use a single material (permitting efficient rendering), simplified to a level of detail appropriate for single model in a scene, and converted to a progressive format to allow clients to quickly display a low resolution representation, with additional detail streamed as desired. This unsupervised transformation significantly lowers the bar for user-generated virtual worlds. Users can contribute arbitrary content, and the system ensures the content's feasibility for other, heterogeneous clients.

This work started primarily as an *engineering* task to build the content conditioning and encoding pipeline needed for large-scale, 3D, interactive metaverses. Our conversion process leverages several known techniques, but in providing a complete, robust, and *unsupervised* system for dynamic virtual worlds, we have solved several problems that arose with previous techniques. Our contributions include several algorithms and novel heuristics for:

- A stopping point for existing supervised algorithms, chosen to work well for a large collection of models;
- Apportioning constrained texture space to areas of a 3D model, with the goal of minimizing loss in quality;
- A new progressive encoding for meshes and textures that balances the trade-offs between efficient transmission and efficient display; and
- A complete, robust conversion framework.

In doing so, we present a framework for the unsupervised conversion of 3D content for use in user-generated virtual worlds. The conversion process produces models in a consistent format, increasing the number of models that can be rendered at real-time frame rates and decreasing the amount of data that needs to be downloaded to first display a model.

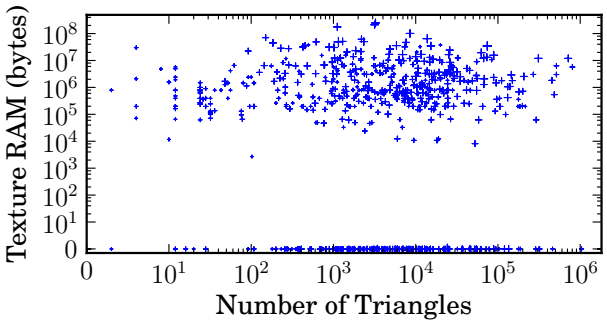


Fig. 1. Number of triangles (x-axis), texture RAM (y-axis), and number of draw calls (marker size) for 748 test models.

2. MOTIVATION AND GOALS

Beyond the scope of this paper, our goal is to build a platform and architecture for scalable metaverses. To aid with the platform's design and development, a set of 15 users were asked to create a set of sample applications. As part of this process, they uploaded 3D models to a content distribution server for use with the platform. Most content came from external sources, while a small percentage were created by the users themselves. Unfortunately, we quickly ran into problems.

Modern consumer graphics cards are only efficient for models with specific properties. As a GPU can render a few million triangles at interactive frame rates, an individual model with hundreds of thousands of triangles does not leave room for complex scenes. Excessively large textures are similarly limiting. Further, a GPU is only efficient when geometry is submitted in a batch, sharing the same set of vertices, textures, and material properties. Modern GPUs only support a few thousand draw calls at real-time rates.

Figure 1 shows a graph of the number of triangles, texture RAM (32 bits per pixel), and number of draw calls (marker size) for the 748 models uploaded over a period of three months. More than half the models uploaded by our users fail to satisfy at least one of these properties required for efficient rendering in a scene with thousands of models.

To enable users to add arbitrary content, we needed a way to convert the models into a format for real-time rendering The goals of the conversion process are as follows:

1. **Reducing Draw Calls:** A major bottleneck for large scenes is the maximum draw calls per second the graphics card supports. Our primary goal is to reduce the number of draw calls to a small, constant number.
2. **Simplifying Mesh:** Clients might want to load a complex mesh at lower resolution, *e.g.*, if the object is far in the distance or the client is running on a low-power mobile device.
3. **Reducing Texture Space:** Since graphics cards have a fixed amount of texture RAM (and for the same reason a simplified mesh is desired), a client might want to load a model's texture(s) at lower resolution.
4. **Progressive Transmission:** A progressive encoding allows a client to start rendering the model with only a subset of the data. This is especially desirable when connected via a low-bandwidth link or when a model covers only a small part of the user's field of view.

3. RELATED WORK

Early work on the simplification of polygonal models was focused on reducing the complexity of geometry alone [1, 2], while later work also considered additional attributes such as colors, normals, and texture coordinates [3, 4]. The first progressive encoding [5] allowed for a model's full resolution to be progressively reconstructed. However, textured models that are simplified with this method produce poor results, which led to simplification algorithms based on texture stretch [6, 7]. Our work closely follows Sander, *et al.* [7] with a few modifications (see Section 4), most importantly to allow the process to run unsupervised.

4. CONVERSION PROCESS

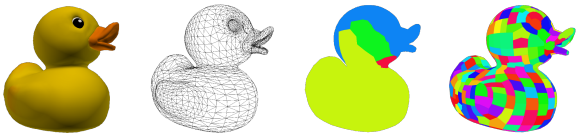
Our unsupervised conversion process turns any 3D model into an efficient, progressive encoding for use within a real-time rendering environment. The conversion process works by executing a series of steps:

1. Cleaning and normalizing the model (Section 4.1);
2. Breaking the model into charts, contiguous submeshes used to map the mesh into a texture (Section 4.2);
3. Fairly allocating texture space to charts (Section 4.3);
4. Packing charts into a texture atlas (Section 4.4);
5. Simplifying the model (Section 4.5); and
6. Encoding the result into a progressive, streamable format (Section 4.6).

4.1. Cleaning and Normalizing

Before the conversion process, the system normalizes the model by performing the following standard steps:

- Quads and polygons are converted to triangles. The mesh simplification algorithms require triangles, and clients would otherwise need to triangulate the model for rendering.
- Missing vertex normals are generated, enabling consistent client-side shading.
- Extraneous data is deleted, including unreferenced data and duplicate triangles.
- Complex scene hierarchies and instanced geometry is flattened to a single mesh. This can increase file size but makes simplification and charting easier, as well as simplifying client model parsing.
- Vertex data is scaled to a uniform size, in order to normalize error values in subsequent steps.



(a) Rendering (b) Wireframe (c) All Merges (d) Heuristic

Fig. 2. Example model of a duck, showing its original rendering (a), wireframe (b), result of merging its charts completely (c), and result of merging its charts when using the heuristic in Formula 1 for determining when to stop (d).

4.2. Creating Charts

A model requires multiple draw calls to render it primarily when it uses multiple materials and textures for submeshes. To reduce the draw calls required (Goal 1), the model's materials must be combined such that the mesh can be rendered in a single batch.[1] A naive approach would simply combine all textures into a texture atlas [8]. The problem with this approach is twofold. First, input models often waste texture space by using only small subsets of large textures, *e.g.*, a few leaves from a photograph of a tree. Second, models that use texture wrapping (*i.e.*, use texture coordinates beyond the dimensions of the texture that must be wrapped) require duplicating the texture many times so that these coordinates do not wind up in neighboring textures. This duplication is wasteful and can consume a large fraction of the texture budget.

Instead, the system copies only the referenced parts of textures into the atlas. To do so, the mesh is first partitioned into charts, or contiguous groups of triangles in the mesh [7]. The system creates a chart for each triangle and a greedy algorithm merges adjacent charts that create the least additional error using a priority queue. The process closely follows the algorithm from Sander *et al.* [7] and Garland *et al.* [9], with a few modifications. First, merging identical but opposite-facing triangles is disallowed, so that double-sided geometry does not result in charts that cannot be parameterized into texture space (see Section 4.3). Second, our algorithm only allows merging two charts if they are both textured or both contain the same color. This allows an entire color-based chart to be trivially paramaterized to a small fixed-size region.

While prior work [7] required an operator to manually choose when to stop the merge process, our algorithm must determine this stopping point automatically. Figure 2(a) demonstrates why selecting a stopping point is important. If the merge process is left to run to completion, only four charts remain (per Figure 2(c)). However, planarity is important when paramaterizing the charts into 2D texture space, because texture stretch increases when planarity decreases. This is the motivation behind using planarity and compactness in the cost assigned to merge operations [9].

[1]For simplicity, our implementation currently only considers the diffuse channel, but the same technique can be repeated for additional color channels, *e.g.*, normal maps, specular highlights, glow maps, etc.

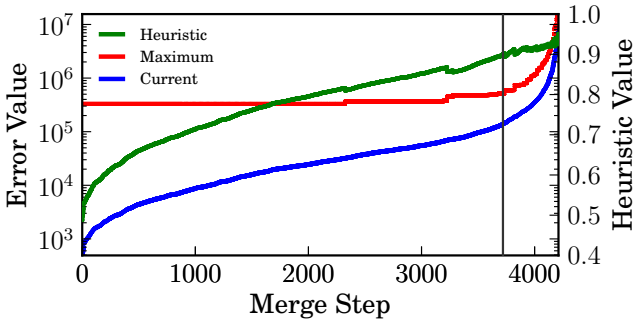


Fig. 3. The current error term, maximum error term in the priority queue, and a heuristic formula during each step of an example chart-merging operation. The vertical line is the chosen stopping point when using a threshold value of 0.9.

Ideally, the process should stop at a point that produces disc-like, planar charts that can be paramaterized into texture space with minimal error. Figure 3 plots the error term at each step of the merging process for the model in Figure 2, as well as the the maximum error term in the priority queue at each step. The error term becomes doubly exponential (note the log scale of the y-axis) around step 3800. The maximum error also remains stable until this point.

Our algorithm stops the merging process around such a point that corresponds to a phase change in the error rate. To automatically detect it, we use the following heuristic, which is also plotted in the figure:

$$\frac{log(1 + E_{current})}{log(1 + E_{max})} \quad (1)$$

where $E_{current}$ and E_{max} are the current error term and the maximum error term seen at each step, respectively. This represents how close the current error term is to the maximum in a log scale. We use a threshold of 0.9, which coincides with the error term becoming doubly exponential. Figure 2(d) shows the model's charts when stopped according to this heuristic metric. This heuristic and threshold work well in practice across a range of models: the error during the merge process for all models in the testing set are similar to that shown in Figure 3.

4.3. Sizing Charts

Once the charts for the mesh have been defined, each chart is paramaterized from 3D into 2D texture space, so that they can be packed into an atlas. We paramaterize the charts using an optimization algorithm based on texture stretch [7]. For charts that are only a single color, however, this paramaterization is trivial: we map all coordinates within the chart to a single coordinate in texture space containing the color.

Each chart has to be given a size in texture space. Sander *et al.* [7], on which our approach is based, define two texture stretch norms over a triangle T: (i) $L^2(T)$, the root-mean-

square stretch over all directions, and (ii) $L^{\infty}(T)$, the maximum singular value. The L^2 norm is used because, as noted, *"unfortunately there are a few triangles for which the maximum stretch remains high."* However, we found that, with a large sample of models, the L^2 norm can also be too large for some charts, leaving very little room for other charts. This is particularly bad when a chart covering a small portion of the mesh has high texture stretch.

Instead, we select a fixed target size for the texture, and we allocate texture space fairly across all charts based on texture stretch, relative surface area in 3D, and relative area in the original textures. The target texture size T is set as the minimum of the total original texture area referenced by all triangles and the maximum texture size for modern graphics cards (we currently use 4096x4096). Each chart is assigned a 2D area equal to:

$$A_c^{''} = \sqrt[3]{(\frac{L_c^2}{\sum L^2})(\frac{A_c}{\sum A})(\frac{A_c'}{\sum A'})} \cdot T \qquad (2)$$

where L_c^2 is the chart's texture stretch, A_c is the chart's surface area in 3D, A_c' is the chart's area in the original texture space, and $\sum L^2$, $\sum A$, and $\sum A'$ are the sum across all charts of texture stretch, 3D surface area, and original texture area, respectively.

4.4. Packing Charts into Atlas

After each chart has been paramaterized, they must be combined into a texture atlas. To enable an atlas to be resized into lower resolutions (Goal 3), a chart must not cross a power-of-two boundary of the atlas. Otherwise, it could bleed into adjacent charts [8]. We developed an efficient chart-packing algorithm to perform this encoding. The algorithm maintains a tree-based data structure, with each node in the tree representing a region of the atlas. Each chart is inserted in decreasing order by size, with the goal of finding a spot containing enough room for the chart's image without crossing a power-of-two boundary. To choose a placement for each chart, the tree is traversed recursively until a valid placement is found; the chosen node is split into the placement and any remaining free space. For all models in the testing set, this method successfully packs charts into a texture atlas within a power of two of the target size chosen in Section 4.3.

4.5. Simplification

While creating charts and mapping them to a single texture addresses Goal 1, achieving Goal 2 requires reducing the complexity of the model. As in Sander *et al.* [7], we use a greedy edge-collapse algorithm based on a combined metric of quadric error [2] and texture stretch. The algorithm generates a low resolution base mesh and a list of refinements which can be applied to reconstruct the original mesh.

The mesh could be simplified until there are no longer any valid edge collapses, but this often results in parts of the mesh's volume collapsing completely, *e.g.*, an avatar's fingers disappear. We found that the combined error metric for mesh simplification follows the same property as the error term for merging charts. To avoid oversimplifying a model, therefore, we apply Equation 1 from Section 4.2 and stop simplification once the metric reaches a threshold of 0.9. This approach works well for the models tested, simplifying the models to reduce their complexity, but leaving the base mesh at an appropriate level-of-detail so as to not lose too much volume.

Some models are not worth simplifying because the cost of sending a batch of triangles to the GPU dominates the cost of rendering the full mesh. For example, sending 10,000 triangles often has no additional cost over sending 5,000 triangles. Therefore, if a model is less than 10,000 triangles or if the progressive stream is less than 10% of the size of the original mesh, we revert the simplification process and encode the base mesh as the full resolution model.

4.6. Progressive Encoding

The ideal progressive encoding (Goal 4) would satisfy the following three properties:

1. The simplified base mesh can be downloaded and displayed without downloading the rest of the data.
2. Progressive refinements can be streamed, allowing a client to continuously increase detail.
3. The mesh's texture can be progressively streamed, allowing a client to increase texture detail.

While there are existing progressive mesh formats, to our knowledge none support complex meshes with textures and materials, and none are widely used. To provide such a progressive format, we start by encoding the base mesh using COLLADA, as it is an open, widely-supported format for 3D interchange. It allows for referencing complex materials and textures, and since it is widely-adopted, existing platforms can use the base mesh without modification.

Mesh data in COLLADA is encoded as indexed triangles. A vertex in a triangle contains an index into a source array for each attribute (*e.g.*, positions, normals, and texture coordinates). This per-attribute indexing allows for the efficient deduplication of source data (*i.e.*, eliminating redundancy), which serves to reduce file size. When sending mesh data to a graphics card, however, a client must create a single set of indices, such that the source attributes are aligned in memory.

The progressive stream format for meshes must balance these opposing requirements. It should be encoded efficiently so as to require less bandwidth. But, it also should not require a client to maintain both the original and converted data to efficiently apply mesh updates, *i.e.*, vertex additions, triangle additions, and index updates.

Our progressive stream is encoded as a sequence of refinements, each comprising a list of individual updates to be

applied together. However, it is encoded assuming it will be applied to the decoded base mesh where indices for vertex data have been converted to a single index while deduplicating index tuples to minimize data size. Because it uses a single index, the progressive stream is slightly larger but allows a client to efficiently add progressive detail to a loaded mesh.

Unfortunately, there are also no existing widely-supported progressive texture formats that meet our needs. We require a format which provides good overall compression, allows a client to download and load a low resolution version, and supports the progressive addition of detail. Decoding most progressive formats, such as JPEG or PNG, require a client to use a full-resolution buffer regardless of how much of the image is loaded. The DDS format encodes compressed mipmaps, but it uses fixed-rate compression for hardware-accelerated texture mapping, resulting in poor compression. Additionally, because it is not a common image format, some platforms (*e.g.*, web browsers) do not have decoders readily available.

Instead, our encoding resizes the full-resolution texture to generate multiple levels of detail in the form of power-of-two mipmaps, each encoded as a JPEG. The mipmaps are then concatenated in a tar file. This approach has several practical benefits: it achieves good compression, allows a client to directly index offsets into the file, *e.g.*, using a simple HTTP range request if content is served over the web, and supports multiple, contiguous resolutions being downloaded with a single request. As with the progressive mesh format, this requires downloading more data overall, but allows a client to load low-resolution textures much more quickly. Critically, the full resolution texture may never be required if an object is far in the distance, a client is moving quickly through a scene, or a client is rendering a scene at low resolution.

5. RESULTS AND ANALYSIS

As part of our Sirikata platform, we created a web service where users upload 3D models in COLLADA format which are converted using the process in Section 4. The conversion is implemented in an open-source library, available at github.com/sirikata/sirikata-cdn, currently running at open3dhub.com.

5.1. Render Efficiency

The primary goal of the conversion process is to improve the rendering efficiency of models by reducing the number of draw calls. Figure 4 shows the throughput, in frames per second, attained when rendering each model using a Macbook Pro with a 2.4 GHz P8600, 4GB RAM, and 256MB NVIDIA GeForce 9400M graphics card. The original models span a wide range, with many showing poor performance. Even after flattening the original model, an expensive operation that a client might be able to perform, the models still span a large range. The converted progressive base format both improves performance and gives more consistent frame rates for the

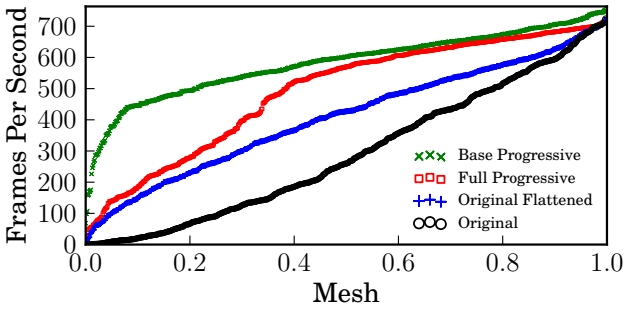


Fig. 4. Render throughput of the original, flattened, base, and full progressive format for each model in the testing set.

		Texture Resolution				
		128	**256**	**512**	**1024**	**2048**
Progressive	**0%**	0.53	0.63	0.81	1.03	1.35
	25%	0.65	0.75	0.97	1.16	1.45
	50%	0.74	0.85	1.02	1.26	1.58
	75%	0.79	0.95	1.11	1.34	1.70
	100%	0.88	0.99	1.20	1.44	1.82

Table 1. Mean size of progressive format as a fraction of the original across all test models, shown as a function of the progressive stream downloaded and texture resolution.

majority of models. The full version of the progressive format predictably has lower throughput for some large models. About 10% of models always have low throughput, however. These models are difficult to simplify because they are not well-connected, *e.g.*, trees. We plan to use other techniques, such as image-based rendering, to handle these models.

5.2. File Size

The size of the converted mesh is also important for performance, as metaverses stream content to clients on-demand. Table 1 shows the mean file size of the progressive encoding as a fraction of the original, across a range of texture resolutions and fraction of the progressive stream downloaded, both cumulative. The mean size for the lowest resolution is half the original, so clients can begin displaying the model earlier. The higher texture resolution is responsible for a large fraction of the full download size, so clients that use lower texture resolution receive a significant bandwidth savings. An example model at multiple resolutions is shown in Figure 5.

Figure 6 plots the change in the number of kilobytes required to display each model. It compares the original against the base mesh with textures no more than 128x128 pixels (corresponding to the top-left cell of Table 1). For 80% of models, the amount of data that has to be downloaded by a client before being able to display the model decreases. The majority of the rest only have a small increase in file size, while a select few increase substantially. Some of the in-

(a) Base Mesh + 128x128 Texture (334 KB)

(b) Base Mesh + 25% Stream + 256x256 Texture (568 KB)

(c) Base Mesh + 50% Stream + 512x512 Texture (923 KB)

(d) Base Mesh + 75% Stream + 1024x1024 Texture (1755 KB)

(e) Base Mesh + 100% Stream + 2048x2048 Texture (4385 KB)

(f) Original Mesh (913 KB)

Fig. 5. Example of a teddy bear model at different resolutions of the progressive format (1 draw call) and its original format (16 draw calls). The size in KB assumes downloading progressively, *e.g.*, 5(e)'s size includes lower-resolution textures.

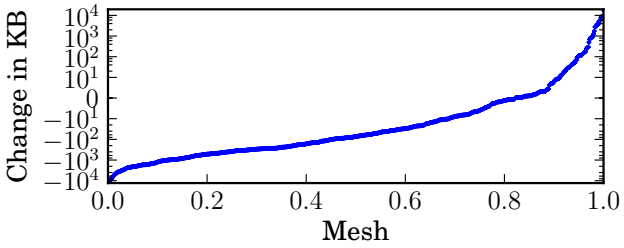


Fig. 6. The change in download size between original models and base converted models at 128x128 texture size.

crease can be attributed to adding extra information to the model that was not present before (*e.g.*, normals and texture coordinates), but the majority is due to flattening instanced models. For heavily instanced models (*e.g.*, trees and grass), this can result in a significant increase in file size, although it still preserves the ability the use a single draw call. As previously mentioned, we are exploring other techniques such as image-based rendering to handle these models.

5.3. Perceptual Error

Besides improving performance, the conversion process should not compromise the appearance of models. We evaluate the visual fidelity by comparing screenshots of the progressive mesh to a screenshot of the original. We start with

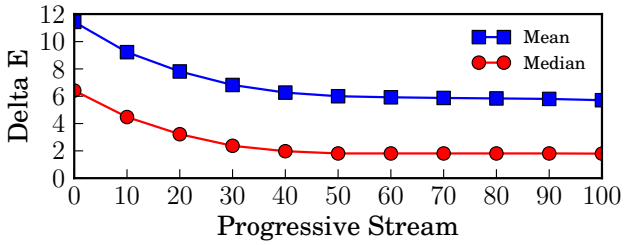


Fig. 7. The perceptual error between original model and converted model as a function of the percentage of progressive stream downloaded. Texture size at $x = 0$ starts at 128x128, increasing by a power of two for each 10%.

the base mesh with 128x128 textures and then, at each step of the experiment, increase the mesh quality of the progressive stream by 10% and the texture quality by a power of two. We compare screenshots using the CIEDE2000 [10] (Delta E) color comparison metric, disregarding background pixels. A CIEDE2000 delta of less than 1 is not noticeable by the average human observer, while deltas between 3 and 6 are commonly-used tolerances for commercial printing. As shown in Figure 7, the perceptual error declines quickly, with the majority of the error becoming indistinguishable once 40% of the progressive stream is loaded.

6. CONCLUSION

In this paper, we describe and evaluate a content-conditioning process that transforms arbitrary 3D content into an efficient representation. Our analysis shows that the resulting format can be efficiently displayed in a real-time rendering environment, while staying faithful in quality to the original format. This conversion process is currently running at open3dhub.com, automatically processing user-submitted content uploads for the Sirikata virtual world.

References

[1] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle, "Mesh optimization," in *Proc. SIGGRAPH '93*, 1993.

[2] Michael Garland and Paul S. Heckbert, "Surface simplification using quadric error metrics," in *Proc. SIGGRAPH '97*, 1997.

[3] Michael Garland and Paul S. Heckbert, "Simplifying surfaces with color and texture using quadric error metrics," in *Proc. VIS '98*, 1998.

[4] Hugues Hoppe, "New quadric metric for simplifying meshes with appearance attributes," in *Proc. VIS '99*, 1999.

[5] Hugues Hoppe, "Progressive meshes," in *Proc. SIGGRAPH '96*, 1996.

[6] Jonathan Cohen, Marc Olano, and Dinesh Manocha, "Appearance-preserving simplification," in *Proc. SIGGRAPH '98*, 1998.

[7] Pedro V. Sander, John Snyder, Steven J. Gortler, and Hugues Hoppe, "Texture mapping progressive meshes," in *Proc. SIGGRAPH '01*, 2001.

[8] NVIDIA, "Sdk white paper: Improve batching using texture atlases," 2004.

[9] Michael Garland, Andrew Willmott, and Paul S. Heckbert, "Hierarchical face clustering on polygonal surfaces," in *Proc. I3D '01*, 2001.

[10] G. Sharma, W. Wu, and E. N. Dalal, "The CIEDE2000 color-difference formula: implementation notes, supplementary test data, and mathematical observations," *Color research and application*, , no. 1, pp. 21–30, 2005.