Content Conditioning and Distribution for Dynamic Virtual Worlds

Jeff Terrace

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Computer Science
Adviser: Professor Michael J. Freedman

November 2012

Abstract

Metaverses are three-dimensional virtual worlds where anyone can add and script new objects. Metaverses today, such as Second Life, are dull, lifeless, and stagnant because users can see and interact with only a tiny region around them, rather than a large and immersive world. The next-generation Sirikata metaverse server scales to support large, complex worlds, even as it allows users to see and interact with the entire world. However, enabling large worlds poses a new challenge to graphical clients to display high-quality scenes quickly over a network.

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 Sirikata's 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.

3D models are large—often several megabytes in size. This poses a challenge for online, interactive virtual worlds like Sirikata, where 3D content must be downloaded on-demand. When a client enters a scene containing many objects and the models are not cached locally on the client's device, it can take a long time to download, resulting in poor visual fidelity. Deciding how to order downloads has a huge impact on performance. Should a client download a higher texture resolution for one model, or stream additional vertices for another? Worse, underpowered clients might not be able to display a high resolution mesh, resulting in wasted time downloading unneeded content. Several metrics, such as the distance and scale of an object in the scene or the camera angle of the observer, can be taken into account when designing a scheduling algorithm. We present the design and implementation of a framework

for evaluating scheduling algorithms for progressive meshes and we perform this evaluation on several independent metrics and methods for combining metrics, including a linear optimization algorithm. After a thorough evaluation, our results show that a simple metric—solid angle—consistently outperforms all other metrics.

Acknowledgements

A heartfelt thanks goes to my adviser, Mike Freedman, for his guidance and support over the years. Mike's deep passion for academic research and system design has had a tremendous influence on my character, and I have him to thank for helping me dramatically improve my professional skills as a researcher and software developer.

I would also like to thank Phil Levis for serving as my unofficial second adviser and for his evaluation and support of my research.

I thank my committee members—Adam Finkelstein, Vivek Pai, and Szymon Rusinkiewicz together with Mike and Phil—for their helpful comments on constructing my thesis. I am also grateful for the administrative help from Melissa Lawson and Nicole Wagenblast that made my graduate career a smooth process.

I have had the pleasure of collaborating with a great team of graduate students from Stanford, including Ewen Cheslack-Postava, Daniel Horn, Behram Mistree, and Tahir Azim. Thank you for making me feel a part of the team, even while being on opposite coasts.

My time at Princeton has been enriched with the friendship of several peers, including Wyatt Lloyd, Will Clarkson, Michael Golightly, Siddhartha Sen, NG Srinivas, Lindsey Poole, Bill Zeller, Ari Feldman, Ana Bell, Muneeb Ali, Hao Liu, Aaron Blankstein, Kay Ousterhout, and Patrick Wendell.

A special thanks goes to my loving parents, Sandy and Bob, for providing for me and supporting me during my education. My sisters and their families—Rachel, Sam, Lily, and Tommy; and Debbi, Rob and Emily—have been encouraging and supportive to me during this time and I thank them for their unconditional love. To my late grandmother, Carrie, I miss you and thank you for all the advice and great memories. To my grandparents, Sam and Henny, you have continued to be an inspiration to me

in how I live my life and pursue my career. To Chloe, thank you for being a loving partner and standing by me for the last two years while I finished my thesis.

Thank you to my best friends—Paul C., Craig, Brian, Paul S., Tom, and Stacy—for adding joy to my life and always standing by me.

The research in this thesis was funded by National Science Foundation NeTS Award #0904860: Designing a Content-Aware Internet Ecosystem, National Science Foundation NeTS-ANET Award #0831374: A Network Architecture for Federated Virtual/Physical Worlds, and the Intel Science and Technology Center (ISTC) Visual Computing research initiative.

To my dad, Bob, for his continued encouragement of my work.

Contents

List of Figures

Chapter 1

Introduction

> *Your avatar can look any way you want it to, up to the limitations of your equipment. If you're ugly, you can make your avatar beautiful. If you've just gotten out of bed, your avatar can be wearing beautiful clothes and professionally applied makeup. You can look like a gorilla or a dragon, or a giant talking penis in the Metaverse. Spend five minutes walking down the street, and you will see all of these.*
>
> —Neal Stephenson, Snow Crash

With the computer becoming ever more powerful over time, applications which were previously never thought possible suddenly became a reality. One such application is the simulation of three-dimensional virtual spaces. Once only available exclusively in the realm of supercomputing, the early 1990s gave rise to the first widely-successful commercial 3D applications. Such hits as Wolfenstein 3D for the PC and Star Fox for the Super Nintendo captivated their audiences with three-dimensional graphics. From then on, the development of three-dimensional games and applications developed into a billion-dollar industry.

The traditional 3D application is static. A team of developers, artists, and producers work together to develop a simulated environment using a fixed set of 3D art. The rules of the virtual world are determined in advance, and users are constrained by the pre-determined limitations put in place by the designers of the application. This, however, works well for many types of applications and has given rise to truly awesome works where one can live the life of a gangster in Grand Theft Auto, take part as a soldier in the simulated World War II world of Call of Duty, or escape into an alternate simulated life in The Sims.

An alternative to the static virtual world—depicted in fictional works like *Snow Crash*—is the dynamic world. The **Metaverse**: a place where users are unconstrained by the rules of the virtual environment. New 3D art can be added to the world, new behaviors can be programmed into virtual avatars, and new applications can be built on a whim.

Similar to the constraints that prevented 3D consumer applications from being widespread prior the early 1990s, today's consumer devices and the realities of our Internet infrastructure so too make developing compelling dynamic virtual worlds a difficult problem. The goal of the **Sirikata** research initiative is to develop a platform for virtual worlds, enabling new and interesting types of applications and solving the problems associated with creating a metaverse within the constraints of today's resources.

1.1 Scalable Virtual Worlds

Metaverses are three-dimensional virtual worlds where anyone can add and script new objects. Metaverses today, such as Second Life, are dull, lifeless, and stagnant because users can see and interact with only a tiny region around them, rather than

a large and immersive world. Current metaverses impose this distance restriction on visibility and interaction in order to scale to large worlds, as the restriction avoids appreciable shared state in underlying distributed systems.

Sirikata is a metaverse platform with a goal of enabling huge, rich, compelling applications. The Sirikata server scales to support large, complex worlds, even as it allows users to see and interact with the entire world. It achieves both goals simultaneously by leveraging properties of the real world and 3D environments in its core systems, such as a novel distributed data structure for virtual object queries based on visible size.

Although the focus and contributions of this thesis lie within a single component of the Sirikata platform—namely, its persistence services—we first present a brief overview of the entire Sirikata platform's design in Chapter 2 to familiarize the reader the components of the system. The design and implementation of Sirikata's persistence services component is then presented in Chapter 3.

1.2 Server-Side Content Conditioning

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 the increasingly-popular 3D scanners that have recently been introduced to consumer markets. 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 to the plain cube with only eight vertices. 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, Chapter 4 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.

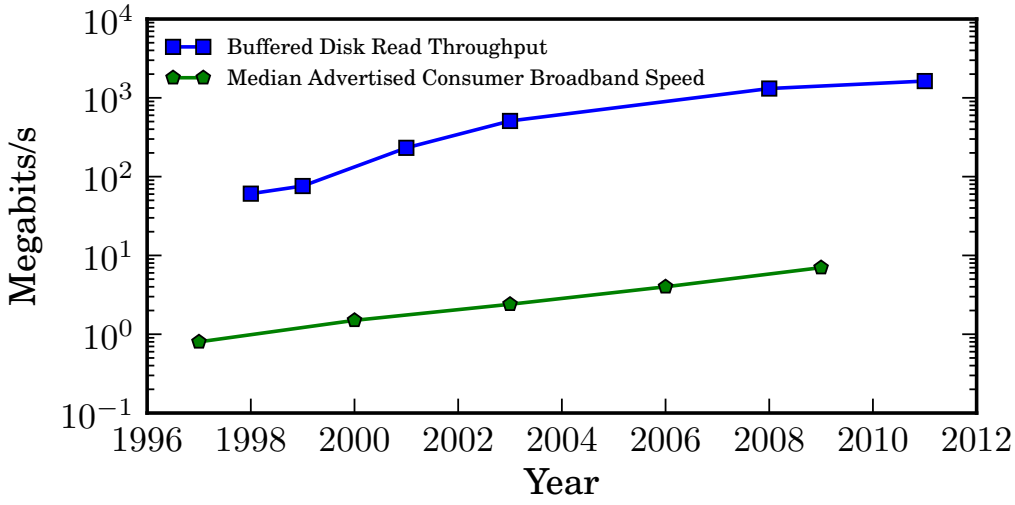


Figure 1.1: Disk read throughput (source: list of personal Seagate hard drives [27]) vs. Internet bandwidth (source: FCC Broadband Performance Report [50]) over time.

1.3 Scheduling Client-Side Downloads

Most 3D applications, like games, hire artists to create high-quality, efficient 3D models. This 3D content, often gigabytes in size, is bundled with the application, and therefore cached locally before the program runs. In contrast, a dynamic virtual world allows users to insert new content into the world, requiring it to be downloaded from a remote server. When dynamic worlds grow large, the time between when a user enters the world and when it can be fully rendered increases, resulting in a poor user experience.

The problem of downloading 3D content over the Internet is getting worse. The size of 3D models has been increasing exponentially over time. This is not a problem in fixed-asset applications because the buffered read throughput of consumer hard drives have also been increasing exponentially, with current devices being able to read hundreds of megabytes per second. On the other hand, consumer broadband speeds have consistently been two orders of magnitude slower than disks, as shown in Figure 1.1. The same loading screen that takes 30 seconds to read assets from disk

might take an hour over the Internet. This is obviously not a reasonable amount of time for a user to wait for an application to load.

In an age where users demand low latency applications, the only solution is to progressively load the metaverse. Similar to streaming video services like Netflix and YouTube, which adaptively change bitrate according to a user's capabilities, so should a 3D application render a low-resolution version of the world, shipping pixels to the screen as fast as possible. Progressively loading a scene, however, requires each individual 3D model to either have multiple levels of detail or be encoded in a streamable format. Otherwise, progressively loading a scene would result in a fragmented world while individual models (on the order of megabytes) load one-by-one.

The flexibility of a progressive format for individual models allows an end client to make decisions about what it should download. A client might download the lowest resolution of all models in a world first to present the user with a crude representation of the scene, while downloading higher resolutions in the background and swapping them in as they load. The question then becomes: in what order should it download the available formats of each model in the scene? Should a high-resolution texture for a large mountain in the distance be downloaded next, or would the user rather see a more detailed mesh of the avatar standing next to her? These questions are difficult to answer, as they are subjective. The right choice in one instance might be the wrong one in another. The scheduling of downloads might depend on the makeup of the scene, the form-factor of the user's device (e.g., a smartphone vs. a tablet vs. a high-performance desktop), or the application being used in the world. Chapter 5 attempts to answer these questions in an objective way using a framework for comparing download scheduling algorithms.

Taken together, this thesis describes the design and implementation of a complete platform for transcoding, hosting, and delivering 3D content in a dynamic virtual world, helping to enable the metaverse of tomorrow.

Chapter 2

The Sirikata Metaverse Platform

In this chapter, we present a high-level overview of the entire Sirikata metaverse platform. Although this thesis focuses on Sirikata's persistence services (Section 2.4), we first give a brief description of how the platform is divided into components in Section 2.1 and then present the Space, a set of servers simulating a virtual environment, in Section 2.2 followed by the Object Host, an end-client that connects to a space and hosts individual objects in the world, in Section 2.3. While this chapter is not part of the novel contributions of this thesis, it spans the breadth of the Sirikata platform to provide background and contains pointers to additional resources where detail is omitted.

2.1 System Overview

The goal of the Sirikata platform is to provide a robust, scalable, easy-to-use platform for dynamic virtual worlds.

Figure 2.1 shows an overview of the three main Sirikata components. The **Space** is the logical unit responsible for simulating the virtual world. Although it could be

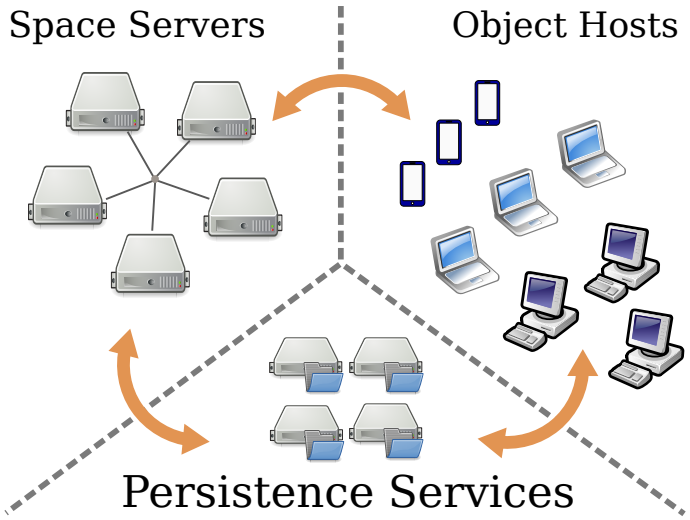


Figure 2.1: The Sirikata metaverse platform architecture.

a single server, in practice, it is typically distributed across multiple servers. The servers coordinate together to simulate the world.

When a new space is created, it is initially empty. To populate it with objects, the space allows an **Object Host** to connect to a space server and insert new objects into the world. While the space is authoritative for the properties of each object being simulated in the world, the object host can issue updates to any objects it owns. An object host can be a user's laptop hosting an avatar object or a server hosting static content like trees or houses. The space also provides a service that lets an object host query to discover other objects. Once discovered, the space also provides a means of communication between objects.

The actual information that the space stores about each object is a small. Part of the information it stores is a URI pointing to a mesh that should be used to render the object. This allows for the decoupling of large files needed for rendering a 3D world from the actual physical properties of objects in the world. The URI for an object points to a networked server responsible for serving content. The **Persistence Services** component of Sirikata, therefore, is responsible for the storage and retrieval of this large 3D content.

2.2 Space

The Sirikata **Space** is responsible for simulating the virtual world. Its goals are to support worlds with millions of objects while maintaining a high visual fidelity for each object connected to the world. Rather than other architectures that partition the world into fixed regions or limit the radius with which an object can interact, Sirikata lets users see the *entire* world. Since there is inherent n^2 scaling problem with allowing each object to see and communicate with every other object, Sirikata degrades as gracefully as possible: when servers are overloaded, objects can still see an interact with the *most important* other objects, rather than some fixed partition or only the closest objects. To achieve graceful degradation and scalability, the Sirikata server uses a number of techniques, outlined in the following sections.

The space server only stores a small amount of information about each object:

- a unique identifier
- a vector representing the position of the object
- the radius of the object
- a vector representing the orientation of the object
- a string of text containing a URI pointing to the object's mesh
- optionally, application-specific information required for implementing physics

This information is intentionally small. It makes simulating the space and sending information about objects over a network fast and easy.

In the following sections, we describe the high-level architecture of the space server using a top-down approach. Many of the details are left out here. For a more detailed explanation of the Sirikata space server, see Cheslack-Postava, et al. [9].

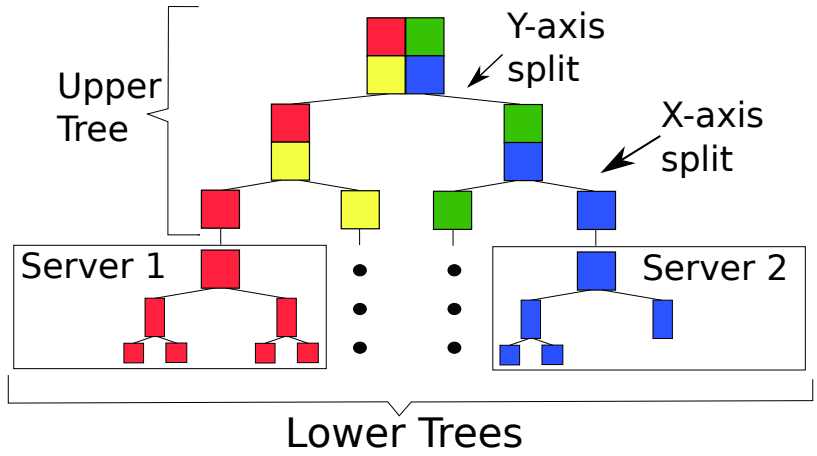


Figure 2.2: An example of the split-axis k-d tree used by the Sirikata world segmentation service.

2.2.1 World Segmentation

A single Sirikata space server can handle a large number of simulated objects, but as a virtual world grows, it eventually exceeds the capacity of a single server. To continue scaling to a larger world, the space must be split across multiple space servers.

To do this, Sirikata has a service called Coordinate Segmentation (**CSEG**) responsible for assigning regions of the infinite three-dimensional coordinate space of a virtual world to separate space servers. The service uses a split-axis k-d tree [3], a tree-based data-structure that partitions a k-dimensional space (here $k = 3$), using an alternating axis at each level of the tree.

An example of the split-axis k-d tree provided by the CSEG service is shown in Figure 2.2. The upper portion of the tree is replicated using a strongly consistent replication algorithm across multiple CSEG servers to provide fault-tolerance and availability. The k-d tree has the property that the upper portion of the tree is very stable, so the throughput requirement for updating the upper tree is low. Most of the updates to the k-d tree occur in the lower portions of tree. Each lower tree is hosted on a single CSEG server, allowing it to be quickly updated locally. If a CSEG server

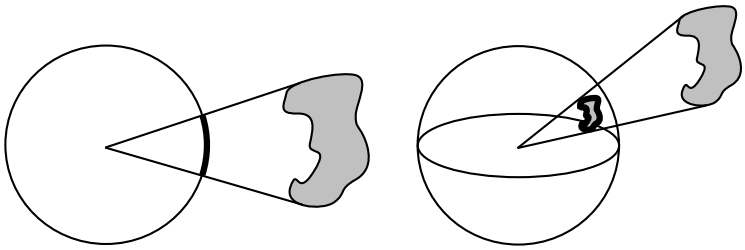


Figure 2.3: Solid Angle (left) is the extension of a planar angle (right) into three-dimensional space. The solid angle of an object is roughly equivalent to the number of pixels an object would take up on the screen of an observer if they were rendering the world in all directions.

fails, the lower portion of the tree it was responsible for is lost, but a new server can take over, querying each space server the CSEG server was responsible for to rebuild the soft-state tree.

To decide when to split a region of space or join two regions of space together, a simple metric is used. If the number of objects located in a region of space exceeds a threshold, T, the region is split in two. On the other hand, if two adjacent regions of space drop below $\frac{1}{4}T$ objects, the regions are joined. This helps prevent a region of space from being continually split and merged.

2.2.2 Object Discovery

When an object first connects to a space server and joins the world, it usually wants to discover other objects. Sirikata calls the system that provides this service **P**otentially **Int**eresting **O**bjects, or **Pinto** for short. When queried, it is responsible for returning a subset of the full objects in the world that are most relevant to the querier. It is also responsible for *standing queries*: that is, when a querier wants to be continuously updated with new results.

The way object importance is determined is based on *Solid Angle*. As shown in Figure 2.3, the solid angle of an object is the area it takes up when projected onto a

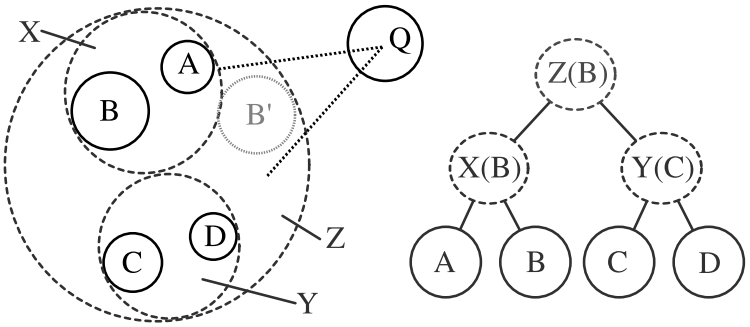


Figure 2.4: An example of the Largest Bounding Volume Hierarchy (LBVH) tree data-structure used by Sirikata's object discovery system.

unit sphere centered at the observer. If the observer were to render the world in all directions, the solid angle of an object would be roughly congruent to the number of pixels it takes up on the observer's screen. This means that larger and closer objects are considered more important than objects that are smaller or farther away.

The problem with using a metric like solid angle to determine object importance is that the most important objects with respect to an observer could be anywhere in the world. In contrast to using distance as an importance metric, a space server might have to search all other space servers to answer a Pinto query. However, using distance would mean that an observer can only see objects within a fixed distance—resulting in a poor user experience.

The Pinto service runs as a component of the space server. Let us first consider the case of answering solid angle queries from a single server. A Pinto query asks a space server to return a list of object identifiers with a solid angle less than some value S with respect to an observer. A naive way to answer this query is to simply calculate the solid angle with respect to the observer for all of a space server's objects. Since this is an inefficient, expensive operation, the Pinto service instead maintains a data structure called a Largest Bounding Volume Hierarchy (LBVH). Similar to a BVH tree, each object's bounding volume is a leaf node of the tree, with interior nodes being a bounding volume encompassing their children, until the root node which contains

all objects. An LBVH also maintains the size of the largest child at each interior node. An example LBVH tree is shown in Figure 2.4. The leaf nodes—A, B, C, and D—represent objects in the world, while interior nodes—X, Y, and Z—contain their children.

Consider searching the LBVH tree in Figure 2.4 for objects that satisfy a solid angle query for observer Q. When the traversal of the tree reaches a node for which the solid angle of the interior node does not satisfy the query, traversal can stop. However, since an interior node's bounding volume is typically much larger than its children, traversal of the tree instead uses the known largest child of the interior node instead—placing it as close as possible to the observer. For example, when evaluating node Z, the largest child node, B, is placed at location B'. The solid angle of B' from the observer Q is the largest possible solid angle for any object below Z. This makes traversing the tree much more efficient because more branches of the tree are immediately cut off.

To distribute the LBVH tree across multiple space servers, each space server maintains two LBVH trees. One contains only the objects for which that space server is responsible for and is used to answer Pinto queries from other space servers. A second LBVH tree merges together its own local LBVH with Pinto results from other space servers. When a space server evaluates multiple Pinto queries from its local objects, it queries each external space server with an aggregated, maximum solid angle so as to reduce query load.

To increase the stability of the LBVH tree, there are actually two separate LBVH trees: one for static objects and one for dynamic objects. In most virtual world workloads, about 95% of objects never move. This makes the static LBVH tree very stable and makes the dynamic LBVH tree small.

To handle standing queries, the Pinto service maintains a *cut* through the LBVH tree for each querier. A pointer to the cut is maintained at each node through which

it traverses. When a space server updates the LBVH tree because of a new object insertion or when an object moves, it checks to see if a querier's cut needs to be updated. If so, it streams new object updates to the querier.

A problem with the Pinto service is that a large cluster of small objects in the distance might not satisfy a solid angle query but could actually be very visibly important to an observer. Consider, for example, a large forest made up of thousands of trees. No individual tree is significant on its own, but the forest in aggregate is important. To alleviate this problem, the space server generates aggregate meshes for interior nodes of the LBVH tree. These interior nodes can than be returned in Pinto queries, allowing clients to display a single mesh containing the entire forest.

2.2.3 Messaging

Once an object discovers other objects in the world, it naturally might want to communicate. A chess application might have players send their moves to a chess board object, while an outer-space simulation might have ships fire their guns at each other. To handle application-specific messaging between object pairs, the Sirikata space server has a message routing and forwarding component.

An object can send a message to any other object in the world. As mentioned previously, the space maintains a unique identifier for each object in the world. This identifier is included with results from Pinto, so an object can use it to send a message through the space. Since it can be anywhere in the world, the destination object of a message could be connected to a different space server than the sender. The space server must forward this message to the other space, but it first must know which space server the given object is connected to. To handle this lookup, Sirikata uses a popular, open-source, reliable key/value database called Redis [38]. The value stored in Redis maps an object identifier to the server it is currently connected to. The

space servers aggressively cache these entries so as to avoid an external lookup when possible. If an entry is stale and a space tries to send a message to the wrong server, it invalidates its cache entry and re-queries Redis. As an optimization, when an object gets migrated from one space server to another, the previous server maintains a forwarding record for a period of time so as to quickly update servers that try to route messages to it.

Objects sending messages at a high rate could easily exhaust the messaging capacity of the space. An ideal message forwarder would allocate its capacity with equal fairness to each flow being routed through the space, while also allow the full capacity to be used when under-utilized. When the space receives a low amount of traffic, it simply forwards all messages. When under load, it uses an algorithm similar to the inverse-square falloff of electromagnetic radiation in the real-world. That is, objects farther away from each other in the virtual world receive less bandwidth than those that are closer. This has the nice property that if an object wants to communicate with another object at a higher rate, it can simply move closer to it. To efficiently enforce the forwarding bandwidth weights, the space server uses a forwarding algorithm similar to Core-Stateless Fair Queueing [47]. For a detailed explanation and analysis of Sirikata's forwarder, see Reiter-Horn's PhD thesis [39].

2.3 Object Host

An Object Host in Sirikata is a process that connects to a space and adds objects to the world. If an object host disconnects from a space, the objects it was hosting are removed from the world. An object host can issue updates to the space to change the location, orientation, scale, or mesh of one of the objects it hosts. If some object in

the virtual world sends a message (see Section 2.2.3) to another object, it gets routed to the object host hosting the object.

An object host can be any application that follows the Sirikata protocol. A user's laptop might connect to the space and add a single object, the user's avatar, to the world and start rendering other objects on screen using a graphics engine. A headless server might connect to the world to simulate a group of flocking birds. The reference Sirikata object host is a C++ application, sometimes called **cppoh**, designed in a modular fashion so that it can be a headless host to multiple objects, a graphics host to a single object, or a hybrid of the two. There is also an experimental JavaScript object host implementation, allowing an object to be hosted inside a web browser.

2.3.1 Scripting

So far, the Sirikata virtual world has been described as just a collection of objects in a virtual three-dimensional coordinate system. To give application meaning to a world, Sirikata's reference object host has a scripting engine—allowing programmatic access to the properties of objects and the messaging subsystem.

Sirikata's scripting language is called Emerson. It is a programming language based on JavaScript with domain-specific extensions. Emerson provides an easy-to-use interface to an application writer, giving her programmatic access to create new objects, modify objects, and send messages through the space to other objects in the world. The Emerson language is extremely extensible because it can transfer script code from one object host to another. Similar to how a web browser executes JavaScript in an isolated sandbox, Emerson runs third-party application code inside a secure environment, such that no script can perform a privileged instruction unless the object host (and by extension, the user) allows it to. For more details on the power of the Emerson language, see Chandra et al. [8] and Mistree et al. [26].

2.4 Persistence Services

The persistence services subsystem of Sirikata is responsible for hosting and transcoding the large files required for the graphical simulation of a three-dimensional virtual world. Some examples of file types that might be needed are:

Meshes

A mesh is made up of vertices, edges and faces that together describe the surface of a three-dimensional object.

Images

An image files are used to map color information (often called texture mapping) onto the surface of a mesh. Image files can also be used for rendering a billboard: a flat two-dimensional surface always facing the camera.

Animation

An animation is used to simulate movement of a mesh through space over time. The most common animation formats are skeletal animation and morph animation.

Audio

Audio files can be used to create sound in the world.

Scripts

Scripts are text files containing a programming language for execution on the object host. The Emerson language, described in Section 2.3.1, uses script files.

The Sirikata architecture decouples the servers hosting this content and the servers simulating the space. In fact, the space servers only store a URI [4] when referencing an external asset, so the *delivery* mechanism with which a client might acquire the

referenced resources is completely abstract. A URI allows for a multitude of protocols, e.g., `http`, `ftp`, `torrent`, etc.

The rest of this thesis focuses on the **Sirikata Content Distribution Network**: the implementation that provides persistent services to the Sirikata platform. The focus of this work is on meshes and their associated textures. Chapter 3 details the design and implementation of this content network, Chapter 4 describes the automatic transcoding process that converts meshes into an efficient format for real-time rendering and transmission, and Chapter 5 describes how the Sirikata object host schedules the downloading of meshes to maximize visual fidelity given the fixed bandwidth constraints of delivering content over the Internet.

Chapter 3

The Sirikata Content Distribution Network

In this chapter, we present the design and implementation of the Sirikata Content Distribution Network (Sirikata CDN). In Section 3.1, we detail the interface the Sirikata CDN presents to end-users and the Application Programming Interface (API) that is exposed to application developers. Since the Sirikata CDN is distributed across multiple geographically separate datacenters, we first describe the design and implementation of a single datacenter's architecture in Section 3.2 and then explain how data replication works in Section 3.3.

3.1 External Interface

The external interface to the Sirikata CDN is how end-users and applications can interact with the service. The goals of the system are to provide a reliable hosting platform for 3D assets associated with virtual worlds, provide a searchable index of

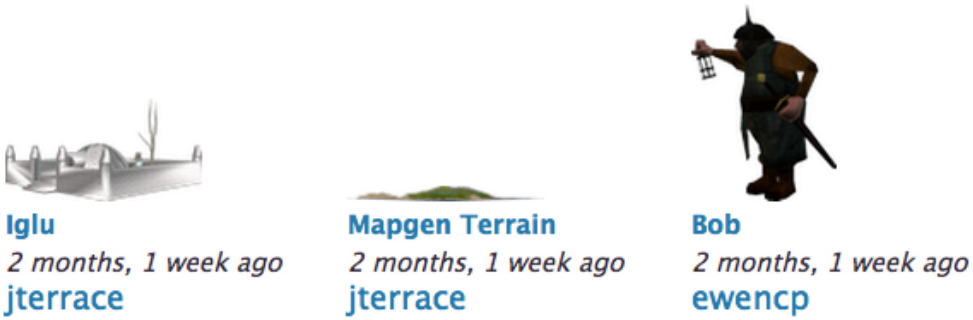


Figure 3.1: The Open3DHub website allows browsing of 3D meshes.

those assets, and provide an intuitive interface for users and applications to upload and download assets.

3.1.1 Website

Today's modern interface of choice to a cloud-hosted Internet service is the website. Running on almost every consumer device in existence today—from handheld smartphones to desktop computers to powerhouse servers—the web browser is one of the only truly portable application interfaces. As such, our choice for the main interface for end-users of the Sirikata CDN is a website running at open3dhub.com [30].

As shown in Figure 3.1, the Open3DHub website allows users to browse the site's 3D content, showing thumbnail image renderings of each model. Users can also search the website, returning models that match their search criteria.

The website maintains metadata about each 3D model in its database. Figure 3.2 shows an example list of the metadata a user can see when they go to a model's page. This includes properties about the models like the number of triangles, materials and textures it contains. Users can download any model stored on the site in a convenient zip file that contains its mesh file and textures.

File Info	
Download Zip	fillmore.dae.zip
Zip Hash	d98bffb2b476e20
Download Mesh	fillmore.dae
Mesh Hash	3f41309223380ac
Triangles	44975
Materials	33
Images	1
Vertices	136059
Texture RAM Usage	353.9 KB
Draw Calls	33
Mesh Size	3.2 MB
Sirikata URL	meerkat:///jterrac
Direct Download	http://open3dhub
Zernike	[0.012156699999
Subfile 1	/jterrace/fillmore.

Figure 3.2: The Open3DHub website shows users properties of 3D models and links for downloads.

Users of Open3DHub can upload 3D models with a web upload form. Uploading is only enabled after a user is authenticated. For authentication, Open3DHub uses the OpenID [31] protocol: an open standard that allows third-party services to provide authentication details about its users. For example, users can log in to Open3DHub with their Google or Yahoo! accounts. Once authenticated, 3D models can be uploaded in the COLLADA [21] format. We chose COLLADA because it is a widely-support, open standard for exchanging 3D assets. Most 3D authoring tools will export to the COLLADA format, making it easy for anyone who has 3D models to upload them to the site.

After a file is uploaded, Open3DHub ensures that it is a valid COLLADA file, converts it to an efficient format (see Chapter 4), generates screenshots, makes the metadata

mentioned above available to users, and makes the file available for download to other users and the API (see Section 3.1.2).

3.1.2 Application Programming Interface (API)

A robust Application Programming Interface (API) is becoming increasingly important for services that operate over the Internet. One of the goals of the Sirikata platform is to make its components easy to use and flexible for developers to incorporate. To this end, the Open3DHub website has an API that runs over the HTTP protocol, the protocol that underlies the web and is readily available for use in all modern programming tools.

As mentioned in Section 2.2, the Sirikata platform uses a URI to locate a mesh for use within the virtual world. The URI scheme used by the Sirikata CDN takes the form `meerkat://[HOSTNAME]/BASENAME/FORMAT[/VERSION]` with fields described in Figure 3.3.

With a URI to a model, an application can perform various actions. A JSON string containing metadata about a model can be retrieved, latest models can be fetched, searches can be performed, and new models can be uploaded. The API methods available to applications are listed in Figure 3.4.

The Sirikata space server also uses these upload URLs to upload aggregate meshes, as described in Section 2.2.2. Since aggregate meshes are ephemeral—that is, they get deleted once they are no longer in use—an additional API URL is available at `/api/keepalive` for the space to send a keep-alive message.

This has been a very brief overview of the API. For more detailed information about the Open3DHub API, see the Sirikata CDN API Documentation [45].

Field	Description
HOSTNAME	*(Optional)* The DNS hostname of the server containing the referenced asset. If not specified, an object host can use a configured default. The hostname for Open3DHub is `open3dhub.com`. Other implementations of the Sirikata CDN protocol are free to operate on other hostnames.
BASENAME	The user-chosen path to the referenced resource, e.g., `/jterrace/duck.dae`.
FORMAT	The format of the referenced model. The Open3DHub site defines three formats: `original`—the uploaded model in its original format, `optimized`—a format optimized for real-time rendering, and `progressive`—a format that can be progressively streamed (see Chapter 4 for details).
VERSION	*(Optional)* The version number of the model. If not specified, the latest version is returned.

Figure 3.3: Descriptions of the fields in Sirikata's URI

Action	URL
Metadata Retrieval	`/api/modelinfo/BASENAME/VERSION`
Browsing	`/api/browse/[?start=TIMESTAMP]`
Searching	`/api/search?q=QUERY&start=START&rows=ROWS`
Uploading	`/api/upload`
Upload Status	`/upload/processing/TASK_ID?api&username=USERNAME`

Figure 3.4: Descriptions of the Open3DHub API methods

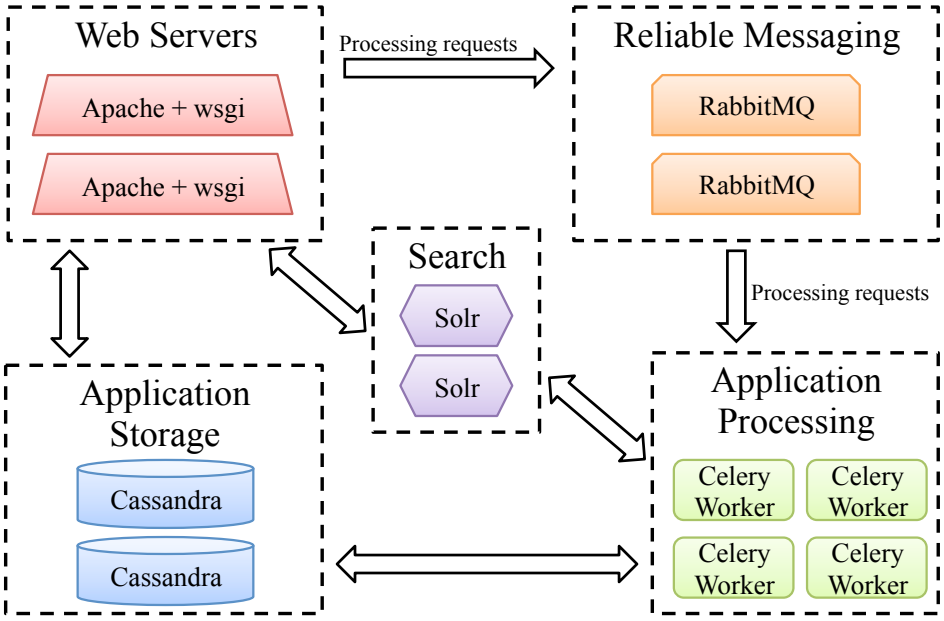


Figure 3.5: The Open3DHub Sirikata CDN datacenter internals. Each dashed-line box makes up a component of the architecture, and the arrows represent communication between components.

3.2 Datacenter Internals

This section presents the internal implementation of the Open3DHub Sirikata CDN service. The architecture uses multiple geographically-distance datacenters to provide fault tolerance, high availability, and good performance, but this section only describes the internals of a single datacenter. The datacenter architecture of Open3DHub is shown in Figure 3.5. The following sections describe the implementation of each component of the datacenter in detail.

3.2.1 Web Server

The web server is the front-end interface to users of the service. Open3DHub uses the Apache Web Server [1] to serve the HTTP protocol. The application code serving the website is available with an open-source license as the Sirikata CDN Project [46]. It

consists of about 3000 lines of Python [35] code, using the Django [11] web framework. The Python code gets executed using WSGI [53], a gateway interface between web servers and Python application code.

The web server processes requests from both the end-user website (Section 3.1.1) and the API (Section 3.1.2).

All services on Open3DHub are provided with open-access, with the exception of uploads, which require authentication so that they can be tied to a specific user. Authentication for end-users is handled using OpenID [31]: an open standard that allows third-party services to provide authentication details about its users. The API instead uses the OAuth [29] protocol—an open, secure protocol that enables users to grant access to applications on their behalf. For example, using an OAuth token, a user can authorize a third-party application to upload models to Open3DHub on his behalf.

As shown in Figure 3.5, the web server application interfaces with the application storage component for retrieving information about models and with the search component for executing search queries. When uploads are initiated by a user or through the API, upload processing requests are sent to the reliable messaging component for later execute by an application processing server.

3.2.2 Application Storage

At its core, Open3DHub is a storage service for 3D content. The traditional approach to application storage is to use a Relational Database Management System (RDBMS) to store application state. Although a valid approach, we decided to instead use Cassandra [6]—a highly-available, fault-tolerant, scalable NoSQL database with great support for cross-datacenter replication.

Column Family	Description
Users	Stores a list of users who have authenticated with OpenID.
Names	Stores a list of the 3D models in the database with their associated metadata.
TempFiles	Temporarily stores the binary file data of uploaded files until they have been processed.
Files	Stores the binary file data for uploaded and verified files.
Sessions	Stores HTTP session information used by the Django framework to look up session state associated with a user's browser cookie.
OpenIdAssocs, OpenIdNonces	Stores OpenID authentication information for users.
CeleryResults	Stores the result of application processing tasks (see Section 3.2.3).
APIConsumers	Stores a list of consumers of the API for use with the OAuth protocol.

Figure 3.6: A list of Open3DHub's Cassandra column families and their descriptions

A full discussion of our database schema is out of the scope of this thesis, but we present a brief description of each of the column families (Casandra's name for what an RDBMS calls a table) in Figure 3.6.

Cassandra has a flexible consistency model. For example, applications within the datacenter can read or write from a single database server or from a quorum of servers, ensuring strong consistency and durability.

The data stored within Cassandra is actually distributed across multiple datacenters. See Section 3.3 for a discussion of cross-datacenter replication.

3.2.3 Reliable Messaging

The life of a web request is typically short-lived. The Open3DHub web application is responsible for web and API requests to the service, but background tasks are handled with the application processing component. When the web server application has a task to be performed that would take longer than a typical web request, it inserts a message to the reliable messaging component that contains a task name and arguments.

We use RabbitMQ [36]—a reliable messaging queue that ensures messages get delivered. The messaging services writes message contents to stable storage before acknowledging receipt, ensuring that they will eventually get delivered. RabbitMQ is designed as a queue of messages. The web server inserts messages to the queue and the application processing servers take messages off the front of the queue to be processed.

3.2.4 Application Processing

The application processing component uses the Celery [7] distributed task framework for handling background operations. Celery allows the Open3DHub web server to asynchronously execute a task on a remote server and poll for the result of the task. The reliable messaging layer that Celery runs on ensures that tasks are eventually executed on an application processing server.

There are several background processing tasks available for execution:

Upload Processing

When a file is uploaded to Open3DHub, it must be validated and inserted into the database. The application processing component uses pycollada [34], a Python library for parsing and modifying COLLADA [21] files and meshtool [25], a Python library for executing operations on COLLADA files using pycollada. This task also executes additional processing tasks after the upload is validated.

Generating Metadata

After an upload is validated, this task uses meshtool to export metadata about the 3D model: properties such as the number of triangles in the mesh and the number of textures referenced by the model.

Generating Screenshots

After an upload is validated, this task uses the Panda3D [32] graphics engine to render the model. Since the application processing servers are running in a headless environment, the X virtual frame buffer (Xvfb) is used to simulate a display.

Search Index Updates

When a model is first uploaded or after it's been edited, the search index needs to be updated. This is done in a background task, contacting the search server to issue the update.

Transcoding

After a file is uploaded, it gets automatically converted to an efficient format for real-time rendering and delivery. See Chapter 4 for details.

3.2.5 Search

To allow users and API clients to search the contents of the Open3DHub repository, we use Apache Solr [2], which supports full-text indexing and faceted search. It can also distribute the search index across multiple servers and supports replication. We currently only index the text-based fields of 3D models: title, description, tags, and filename. However, Solr can also index other types of information, so we could index models based on their metadata. For example, a useful search query might be to find all models in the database that have less than N triangles. We leave this implementation as future work.

3.3 Cross-Datacenter Replication

The Sirikata CDN architecture allows for replicating its services across multiple geographically-diverse datacenters. This is beneficial for fault-tolerance and availability. It also allows for low-latency access to the CDN for faster downloads. Our prototype implementation currently has datacenters hosted at Princeton University in Princeton, New Jersey and at Stanford University in Stanford, California. To route clients to the nearest datacenter, we use the DONAR [52] DNS routing system.

A distributed datacenter architecture raises interesting questions about consistency and replication. The two main components that need to be replicated from the Open3DHub datacenter components outlined in Section 3.2 are the Cassandra database and Solr search. A depiction of our replication scheme is shown in Figure 3.7.

Cassandra has cross-datacenter wide-area replication built-in to its architecture. It uses an eventually-consistent approach to replication across the wide-area. Within a datacenter, all operations that read from and write to Cassandra within the

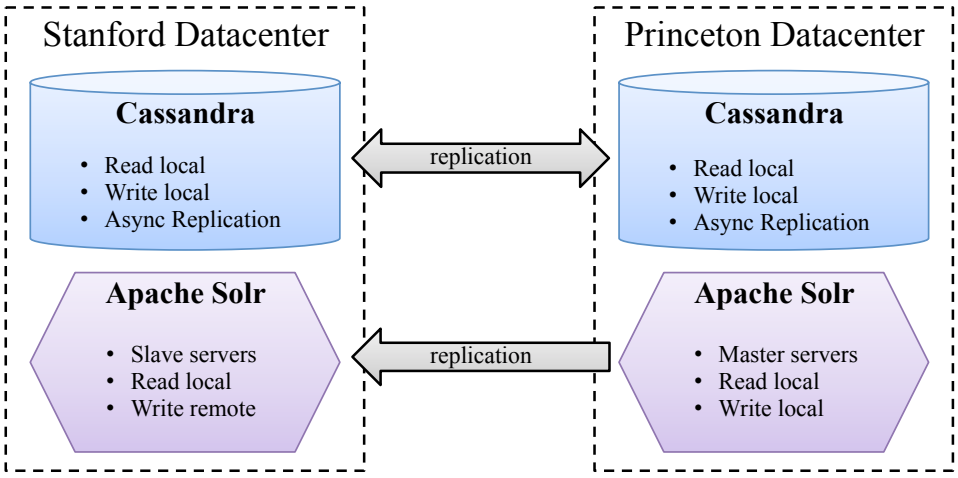


Figure 3.7: Replication between datacenters in the Open3DHub Sirikata CDN.

Open3DHub code base use Cassandra's `LOCAL_QUORUM` consistency flag, which requires that a quorum of servers in the local datacenter respond to the given query. Since clients are pinned to a single datacenter by DONAR, client applications will get predictable semantics, such as read-after-write consistency. It is possible for data loss to occur if a datacenter goes down before it has a chance to replicate outstanding data to an additional datacenter. However, as long as the datacenter eventually comes back online, replication will continue as expected.

There is one inconsistency that can occur with our Cassandra architecture. If a Sirikata client uploads content to Open3DHub and immediately adds it to a space server, another client connecting to a different Open3DHub datacenter might not immediately have access to the new content. We circumvent this issue by having clients retry requests to the CDN if a valid URI from the space is not yet available. Once the data is eventually replicated, the client will be able to download it as expected.

The Apache Solr service provides replication by using a master server and a number of slave servers. Writes can only be issued to the master server, with replication being performed asynchronously to slave servers. The way we handle this is to have

a single datacenter contain the master Solr server, with writes to the search index from that datacenter being local operations. The rest of the datacenters only contain a slave server, so writes cannot be issued locally. Writes from slave datacenters write remotely to the master datacenter, but we encapsulate the search index updates into a Celery task. If the remote datacenter is not available when the task is run, the task gets re-executed later, using an exponential backoff for the amount of time to wait before re-executing. Once the remote datacenter comes back online, the Celery task will succeed in updating the search index. The search tasks are executing using the reliable messaging layer, ensuring that tasks will eventually get executed.

Chapter 4

Content Conditioning

A key feature of Sirikata, and dynamic virtual worlds in general, is to allow users to insert new content into the world. Arbitrary 3D content, however, is often not optimized for real-time rendering or streaming over a network. To provide a streamable, efficient format to end-users, this chapter presents an automated, unsupervised conditioning pipeline for 3D content, which Sirikata's CDN servers execute upon upload of content from a user. 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. However, 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.

4.1 Motivation

During the summer of 2009, a group of 15 students at Stanford and Princeton were asked to create sample Sirikata applications. As part of this process, they uploaded 3D models to the Sirikata CDN hosted at Open3DHub. 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. Since a GPU can only render a full scene with 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 4.1 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

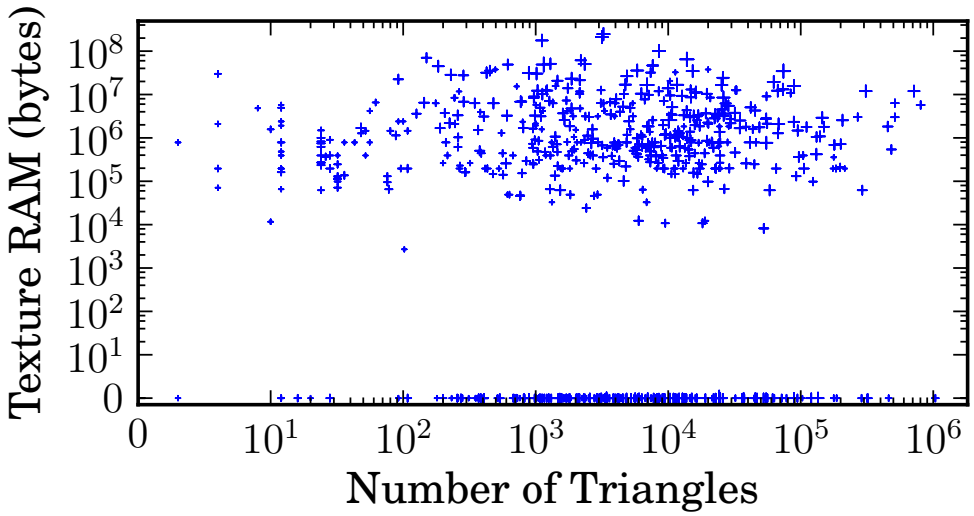


Figure 4.1: Number of triangles (x-axis), texture RAM (y-axis), and number of draw calls (marker size) for 748 test models.

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.

4.2 Related Work

Early work on the simplification of polygonal models was focused on reducing the complexity of geometry alone [13, 20], while later work also considered additional attributes such as colors, normals, and texture coordinates [14, 19]. The first progressive encoding [17] 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 [10, 42]. Our work closely follows Sander, et al. [42] with a few modifications (see Section 4.3), most importantly to allow the process to run unsupervised.

4.3 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:

- Cleaning and normalizing the model (Section 4.3.1);
- Breaking the model into charts, contiguous submeshes used to map the mesh into a texture (Section 4.3.2);
- Fairly allocating texture space to charts (Section 4.3.3);
- Packing charts into a texture atlas (Section 4.3.4);
- Simplifying the model (Section 4.3.5); and

- Encoding the result into a progressive, streamable format (Section 4.3.6).

4.3.1 Cleaning and Normalizing

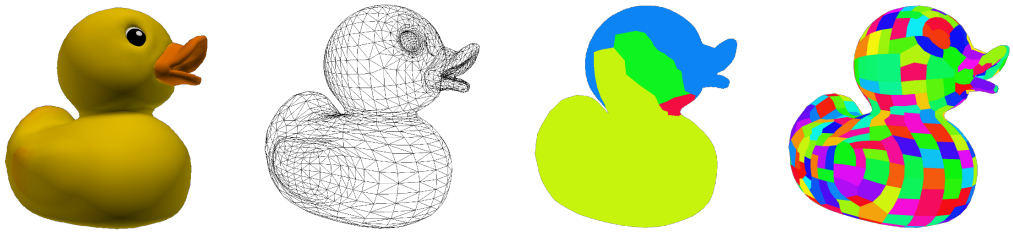
Before the conversion process, the system normalizes the model by performing the following standard steps:

- 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.

4.3.2 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 [28]. The problem with this approach is twofold. First, input models often

[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.



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

Figure 4.2: Example model of a duck, showing its original rendering (4.2a), wireframe (4.2b), result of merging its charts completely (4.2c), and result of merging its charts when using the heuristic in Formula 4.1 for determining when to stop (4.2d).

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 [42]. 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. [42] and Garland et al. [15], where each merge operation is assigned a cost that measures both its planarity and compactness. However, we had to make 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 4.3.3). 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.

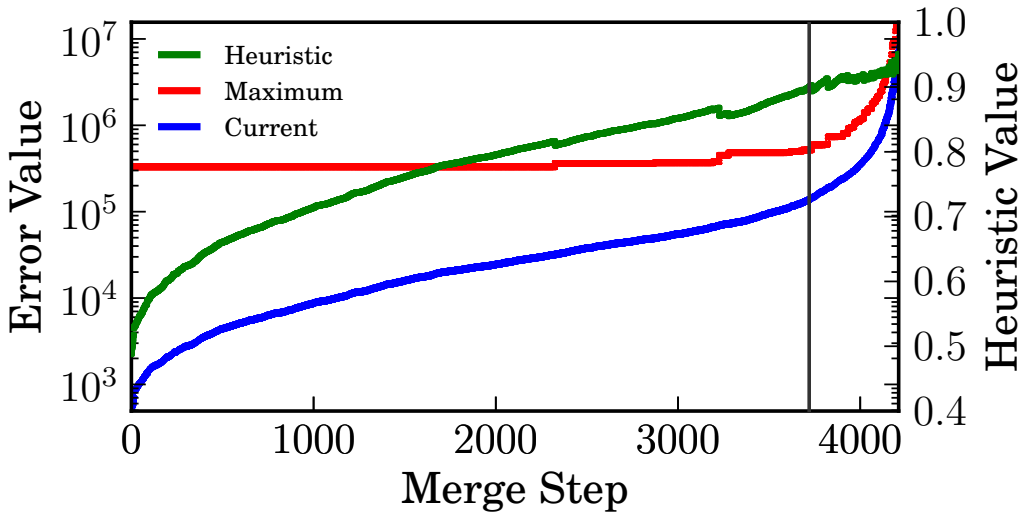


Figure 4.3: The current error term, maximum error term in the priority queue, and a heuristic formula during each step of an example chart-merging operation. The vertical line is the chosen stopping point when using a threshold value of 0.9.

While prior work [42] required an operator to manually choose when to stop the merge process, our algorithm must determine this stopping point automatically. Figure 4.2 demonstrates why selecting a stopping point is important. If the merge process is left to run to completion, only four charts remain (per Figure 4.2c). 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 [15].

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 4.3 plots the error term at each step of the merging process for the model in Figure 4.2a, 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:

$$\frac{log(1 + E_{current})}{log(1 + E_{max})} \tag{4.1}$$

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 4.2d 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 4.3.

4.3.3 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 [42]. 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. [42], 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:

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

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.

4.3.4 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 [28]. 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 4.3.3.

4.3.5 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. [42], we use a greedy edge-collapse algorithm based on a combined metric of quadric error [13] 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 4.1 from Section 4.3.2 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.

4.3.6 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.

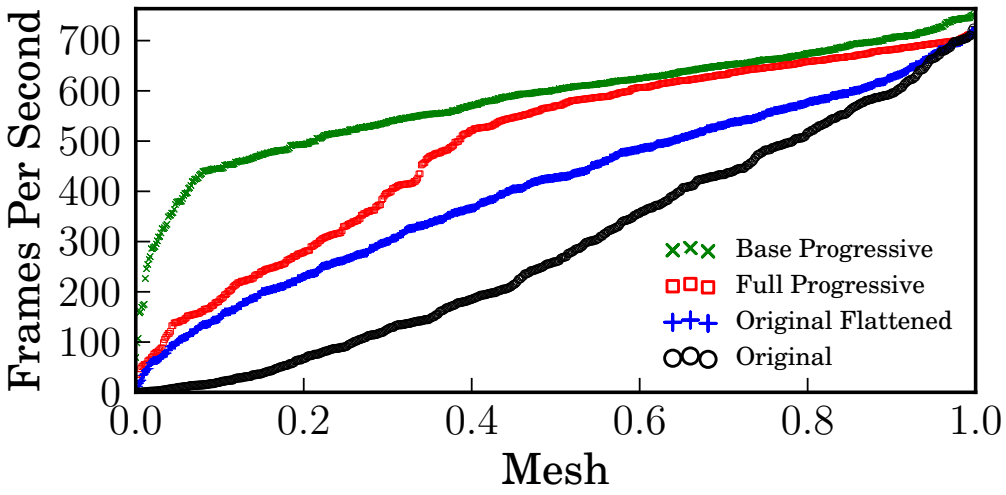


Figure 4.4: Render throughput of the original, flattened, base, and full progressive format for each model in the testing set.

4.4 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 4.3. The conversion is implemented in an open-source library, available at https://github.com/sirikata/sirikata-cdn. The library is currently running on a production server at open3dhub.com.

4.4.1 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 4.4 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

Progressive	128	256	512	1024	2048
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

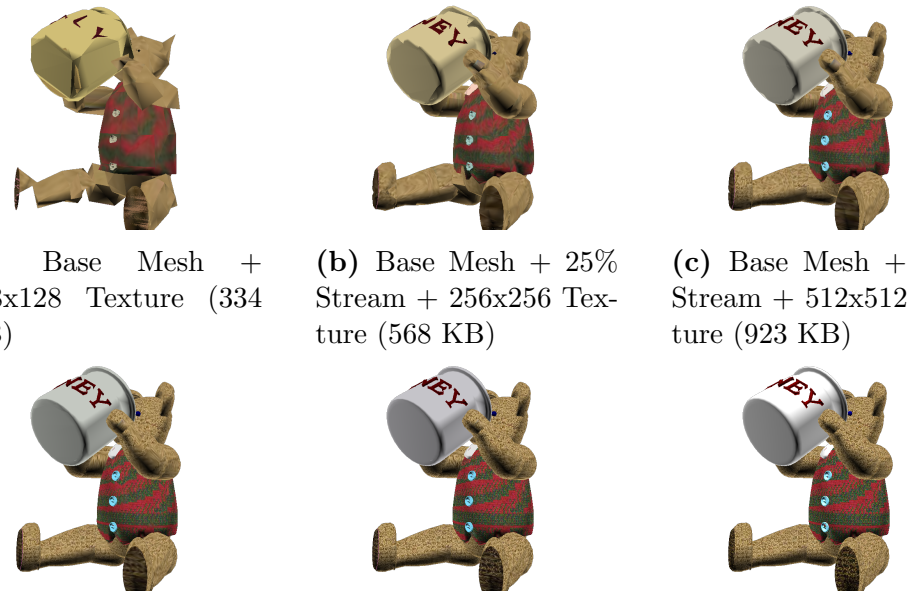
Figure 4.5: Mean size of progressive format as a fraction of the original across all test models, shown as a function of the progressive stream downloaded and texture resolution.

both improves performance and gives more consistent frame rates for the 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. Other techniques could be used for these models, such as image-based rendering.

4.4.2 File Size

The size of the converted mesh is also important for performance, as metaverses stream content to clients on-demand. Table 4.5 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 4.6.

Figure 4.7 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 4.5). For 80% of models, the amount of data that has to be downloaded by a client before being able to display the model



(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)

Figure 4.6: 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., 4.6e's size includes lower-resolution textures.

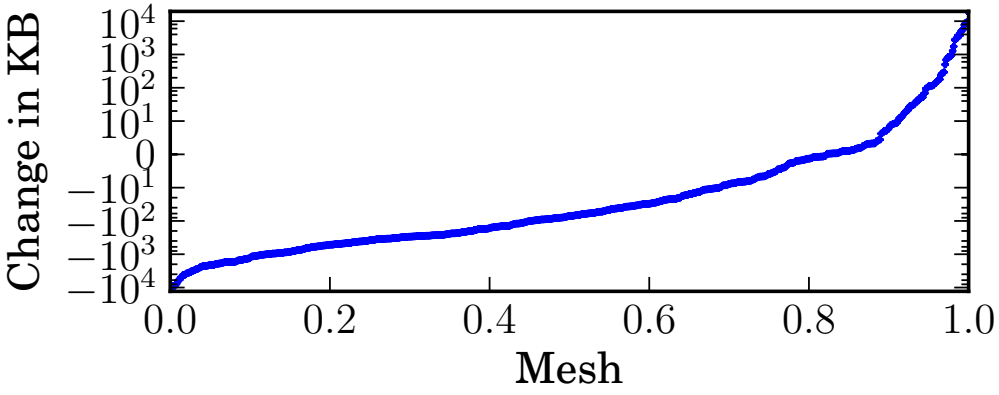


Figure 4.7: The change in download size between original models and base converted models at 128x128 texture size.

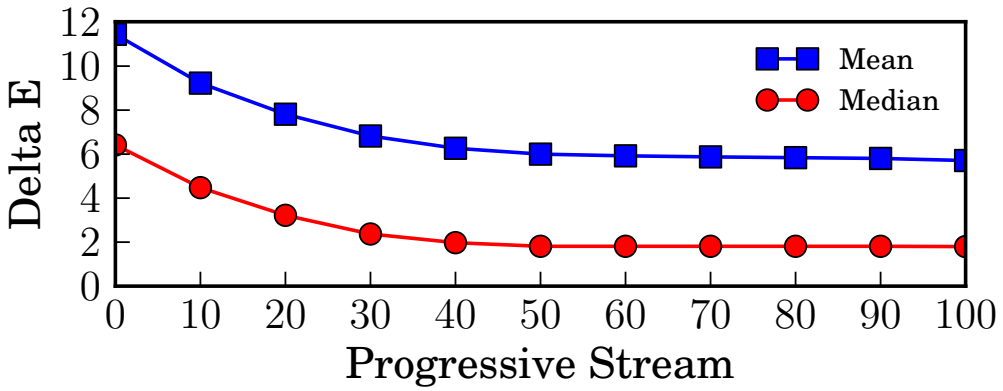


Figure 4.8: The perceptual error between original model and converted model as a function of the percentage of progressive stream downloaded. Texture size at $x = 0$ starts at 128x128, increasing by a power of two for each 10%.

decreases. The majority of the rest only have a small increase in file size, while a select few increase substantially. Some of the increase 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.

4.4.3 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 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 [44] (Delta E) color comparison metric,

disregarding background pixels [2]. 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. Although this method is dependent on the camera angle and more advanced comparison techniques exist [37], it offers a simple and intuitive measurement of perceptual error. As shown in Figure 4.8, the perceptual error declines quickly, with the majority of the error becoming indistinguishable once 40% of the progressive stream is loaded.

[2] The CIEDE metric is a subset of the perceptualdiff metric used in Section 5.4.2 and produces an equivalent output on our datasets.

Chapter 5

Download Scheduling

Chapter 4 presents an automated process that converts 3D content into an efficient format. This enables 3D models to be progressively streamed to an end-client, but large scenes can be made up of hundreds of objects. The order in which a client downloads pieces of each 3D model's progressive stream has a large impact on the visual makeup of the scene. Therefore, in this chapter, we present a framework for comparing download scheduling algorithms for progressive meshes in dynamic virtual worlds. Our goal is to define an *objective* metric with which to evaluate scheduling algorithms, attempting to *minimize visual error* over time for a client downloading resources from a remote server. Our framework records the rendering of the world as a scheduler runs, using an image comparison metric to find an objective measure of how well one algorithm performs compared to another. Armed with this framework, we evaluate several different individual metrics for scheduling downloads as well as combinations of these metrics. We also run two different linear optimization algorithms to try and reach an optimal scheduling algorithm. After a thorough evaluation, we find that a single, simple metric—solid angle—consistently outperforms all other metrics.

5.1 Related Work

5.1.1 Rendering Complex Scenes

Previous work by Funkhouser et al. [12] is similar to our work in that it estimates a *cost* and *benefit* of available levels of detail for each object in a 3D scene. A optimization algorithm greedily chooses as many objects as it can to minimize visual error while maintaining an interactive frame rate. A key difference is that rendering a model's LOD from memory is several orders of magnitude faster than loading via a networked link. The work focuses on optimizing for a consistent frame rate, rather than minimizing visual error. As the latency required to load a model increases, the importance of the metric used to decide what to load next increases. Therefore, we evaluate multiple possibilities for a benefit calculation rather than using a fixed formula.

Methods for accelerating walkthroughs [43] of complex 3D scenes have been devised that construct a tree-based structure of the scene and then cache rendered frames for later use. This works well in reducing the rendering time of subsequent frames, allowing a client to maintain interactive frame rates for complex environments. This work does not apply to download scheduling in dynamic virtual worlds, where end clients still have to download large amounts of information before a frame can be rendered. This method could, however, be used to accelerate the rendering of subsequent frames after an object has been downloaded locally.

View-Dependent Level-of-Detail calculation and rendering [18] has been used to accelerate the rendering of very complex meshes, with a particular application to terrain. A preprocessing step generates a set of geomorphs that are then used to stream progressive meshes [17] depending on the view of the camera. Although the work focuses on single, complex objects, it could easily be extended to scenes. However, it requires

preprocessing the scene, which isn't possible in a dynamically changing environment. Each individual object could instead have its own view-dependent stream, considered separately from other objects, but no indiciation is given on how to prioritize multiple objects, which is what we deal with here.

5.1.2 Streaming Meshes

A method for progressive transmission of multi-resolution models from To et al. [49] uses a progressive mesh file, along with state maintained at a server that allows a client to stream a mesh progressively according to its viewing angle. In our system, we avoid server state to scale the virtual world, so a view-dependent stream method such as this would be difficult to provide. The choice given to the client of what to download next is made using a simple metric based on its distance and size, without further evaluation of different scheduling algorithms.

Rather than using a progressive mesh format, subdivision surfaces can be progressively transmitted [24] instead, with a similar goal of allowing clients to selectively stream additional detail. The main advantage of subdivision surfaces over progressive meshes is that they allow for unordered transmission and reconstruction of the mesh. On the other hand, since we use a large progressive mesh chunk size and ordered-delivery via TCP, this advantage wouldn't gain much unless the network medium was unusually lossy.

QSplat [40] uses a point-based rendering system together with a bounding-volume hierarchy to allow objects to be progressively refined in detail. Its followup work [41] allows this format to be streamed over a network by explicitly requesting nodes of the hierarchy to be transmitted from server to client. These nodes are ordered based on their screen size, without discussion for other possible ordering schemes.

Work from Yang et al. [54] divides each mesh into several partitions, allowing each partition to be individually streamed to end-clients. A server computes visibility and decides what partitions to stream to a client next, e.g., occluded objects would receive a lower resolution than visible objects. Unfortunately this technique does not scale with an increased number of clients. Servers would have to continually update this visibility information, thereby requiring a linear increase in work as the number of clients increase. In the Sirikata architecture, we avoid maintaining state on the server, instead pushing work to the client when possible.

Kim et al. [22] provide a framework for view-dependent streaming of progressive meshes. Their technique scales well from the client's perspective because a compact data structure is maintained at the server indicating which parts of the progressive mesh have been transmitted so far. The client sends its current viewing angle to the server, receiving back a list of refinements. As mentioned previously, maintaining and updating this state at the server doesn't scale well to large, distributed worlds or large numbers of clients.

5.2 Download Scheduling

In this section, we describe the problems and questions associated with the scheduling of downloads in an interactive 3D application, present several metrics that might be used when deciding on the priority of downloads, and then discuss the methods for combining multiple metrics together to make a decision about what should be downloaded next.

5.2.1 Problem

Unlike in application where assets are fixed and can therefore be preloaded, a dynamic 3D environment allows users to add new objects to the world. For another client to display this object, it must download the data needed to load it into its GPU.

When a client enters a virtual world, it gets a list of objects in the world that need to be displayed. Some objects might have associated model data cached on the local disk that was previously downloaded: either at the full resolution or some lower, partial resolution version. Other objects might have nothing available locally and can't be displayed until something is downloaded over a network.

The goal of the client should be to minimize the visual error of each rendered frame with respect to the full-resolution version of that frame. The client has a fixed amount of bandwidth with which to download missing objects, so the question the client needs to answer is: given the list of objects in the scene and the list of model data that has already been downloaded, what should be downloaded next to minimize visual error? Since the latency of downloading an object is several orders of magnitude greater than the latency of rendering a frame, the order in which downloads are executed determines what the user sees when loading a scene. We outline the exact objective metric we use to compute visual error, modeled after how a human would perceive it, in Section 5.4.2.

5.2.2 Metrics

There are many possible metrics that a client could take into account when deciding how to schedule its next download:

Distance

How far away the object is from the user's camera. Objects that are closer might be more important to download before objects that are farther away.

Scale

How large the object is in the scene. A larger object might be more important.

Solid Angle

How large the object appears when looking from the user's camera. Encompassing both distance and scale, the solid angle roughly approximates how many pixels the object would take up on the user's screen. An object with a larger solid angle might be more important for scheduling downloads.

Camera Angle

The angle of rotation between the center of the camera and the object. Objects closer to the center of the field of view of the observer might be more important than, eg an object behind the camera.

Precomputed Perceptual Error

The perceptual error (see Section 5.4.2) between a lower-resolution version of the model and its full resolution. Given a model format that is streamable (see Section 5.3.1), having an indication of the benefit of downloading a higher resolution version of a model could be helpful for scheduling downloads.

Motion Prediction

A prediction of what the user's location will be in the future. If the user is moving very fast in some direction, objects located in the vicinity of that direction might be more important. This could be implemented with simple linear interpolation or a more complicated algorithm.

Occlusion

Whether or not the object is occluded by another object. If an object is not visible because it is occluded, downloading it might be less important. This might be estimated based on the bounding volume of each object in the scene or calculated using an advanced raytracing technique.

5.2.3 Making a Decision

Given several of the metrics from Section 5.2.2 for each object in a scene, deciding how to use the information to generate a decision for what to download next is an open problem. Although the space of possible combinations is infinite, we propose several feasible strategies:

Single Metric

One option is to ignore all of the metrics except for a single choice. For example, the scheduler could simply always download the largest object in the scene first, thereby only taking into account the Scale metric. This has the advantage of being very easy to implement, but might not be the optimal choice.

Linear Combination

Combining multiple metrics in a linear fashion. That is, normalize each metric to a numerical value between 0 and 1, weighting each value by a constant factor, and taking the sum to get a result value. The difficulty with this method is how to decide what the weights should be for each metric. This is particularly difficult because the scale of the independent metrics varies.

Multiplication

Combining multiple metrics by multiplying their normalized values. This allows each metric to contribute to the final result value in an intuitive way. The problem

with this approach is that because of the independent scale of each metric, a single metric could end up dominating the ordering of the result.

Other

There are various other methods that could be devised, such as a combination of linear weights and multiplication or a non-linear combination of metrics.

To get an idea for how each metric contributes to the scheduler, we first thoroughly explored each individual metric in isolation. This presents a baseline for how well the metric does at approximating visual utility. We also explore linear combinations and multiplications in our evaluation in Section 5.4. Although other metrics could be devised, we could think of no immediately apparent justificiation for more complex approaches to combining metrics. We leave this as a possible avenue of exploration for future work.

5.3 Implementation

This section presents a detailed explanation of the 3D progressive file format we use for our implementation and describes the comprehensive testing framework we built for evaluating download scheduling algorithms.

5.3.1 3D Progressive File Format

As mentioned in Section 1.3, a progressive file format is necessary to allow clients to have fine-grained control over what they download. The progressive file format we use is based on our previous work [48], presented in Chapter 4, on designing an unsupervised conversion framework for 3D models. A user can upload any valid

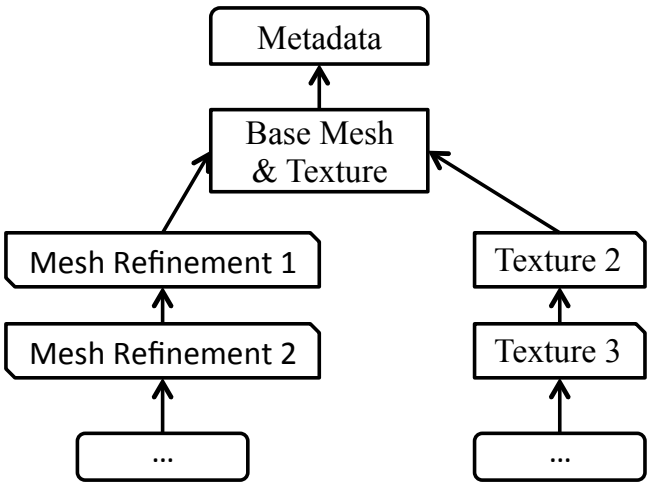


Figure 5.1: Dependency graph for progressive format downloads

3D model to the content repository and it gets automatically converted into the progressive format.

The progressive file format consists of three types of downloads:

Base Mesh

A base, low-resolution version of the mesh that is small in download size and efficient to render

Mesh Refinement

A chunk of vertex additions, triangle additions, and triangle index updates that refines the mesh from its current level of detail into a higher level of detail

Texture Update

A new texture image that can be swapped in to increase the model's texture detail, e.g., from 128x128 to 256x256.

The base mesh must be downloaded before a mesh refinement or texture update can be applied. Each mesh refinement depends on the mesh refinement before it. Texture updates are encoded as a full, independent image at each resolution. Sim-

ilar to mipmapping techniques, this results in a 33% overhead vs. storing only the full-resolution texture or a progressive texture encoding. We chose to encode each resolution independently for compatibility with platforms (e.g., the web) that only support decoding standard image formats. We treat texture updates as if they were dependent on the previous resolution to enable using a progressive, streamable image format in the future.

Before downloading anything for a model, a client must first find out what is available to download. Given a URL for the model, it downloads a small chunk of metadata that contains information about the base mesh, mesh refinements, and texture updates and details about the model such as how many triangles and vertices it has. The dependency tree for model downloads is shown in Figure 5.1.

5.3.2 Scheduling Algorithm Evaluation Framework

To evaluate download scheduling algorithms, we built a framework in Python consisting of about 2500 lines of code. At the core of the framework is a progressive scene loader that renders a scene using the Panda3D graphics engine [16], allowing for a pluggable algorithm when determining how to schedule its downloads. The scene loader takes a scene file as input that determines the objects in the scene, and a motion path file containing camera positions and orientations over time. The scene loader then interpolates the movement of the camera through the scene, taking a screenshot once a second.

To decide how to download the scene, a priority algorithm is given a set of inputs about the current state of the scene and outputs the next item for download. Multiple TCP connections are opened to the content server, allowing multiple downloads (currently 4) to be outstanding at the same time, a common approach used in, e.g.,

web browsers, to get good throughput. When a download slot becomes empty, the priority algorithm gets queried for the next download that should be started.

The input to the priority algorithm is the set of metrics described in Section 5.2.2[1] for each object and the state of each object in its download dependency tree (Figure 5.1). Each metric is normalized to a value between 0 (lower priority) and 1 (higher priority). The expected output from the priority algorithm is a number representing the utility of that object, U_i. Since each model, m, can be instanced multiple times in the scene, represented by I_m, the utility values of each instance for the model are summed together to get a final utility value for the download. Each final utility value is then divided by the number of bytes that would be required for the download. This resulting priority value, P_m, is calculated as:

$$P_m = \frac{\sum_{i \in I_m} U_i}{S_m}$$

Given c free download slots, the scheduler chooses the c items with the highest value for P_m. When a download completes, the client can cache the result on disk to avoid having to download it again in the future. A simple least-recently-used algorithm can be used to discard items from the cache if it grows beyond a user-specified threshold.

5.4 Evaluation

In this section, our testing dataset is first presented, followed by an analysis of the perceptual image comparison metric that is then used to compare and analyze different scheduling algorithms.

[1] The exception being occlusion because we found it too difficult to accurately estimate object occlusion in real-time using only the CPU.

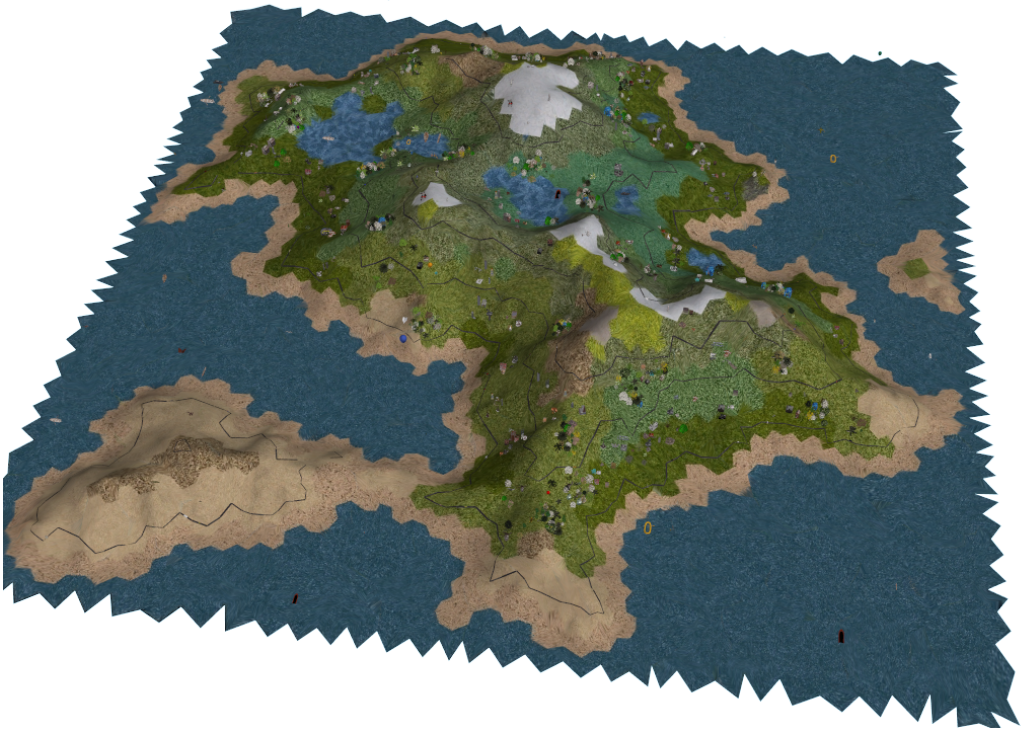


Figure 5.2: Rendering of the generated island scene

5.4.1 Test Data

Acquiring free, high-quality 3D content is a notoriously difficult problem. There are a number of commonly used open datasets of 3D models and scenes, but they typically only have a small number of low-quality models. As part of an evaluation of the Sirikata metaverse platform [9], 15 users were asked to create a set of sample applications. To aid in 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. In total, about a thousand openly-licensed 3D models of varying quality were uploaded to the Sirikata content repository at open3dhub.com.

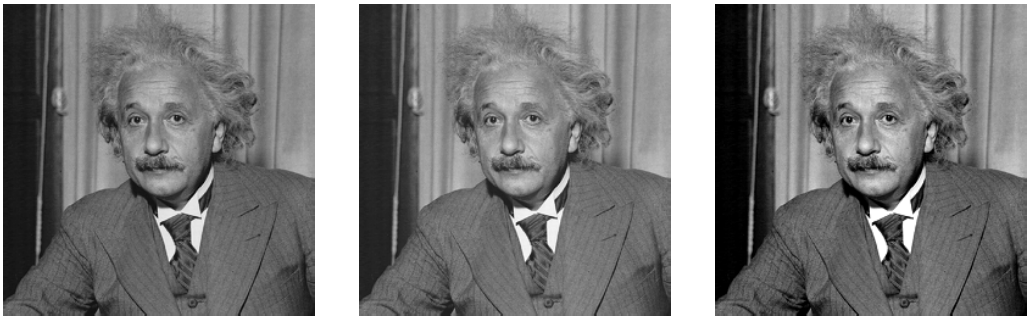
There are a few 3D scenes available for benchmarks, but they tend to be very small, indoor scenes. Instead, we could create a random large scene made up of an as-

sortment of models from our repository, but we wanted to create a representative, compelling scene that one might find in an online virtual world. We chose to use an island map generator tool [33] that generates a large island consisting of regions that are each assigned a biome (e.g., ocean, lake, sand, forest, desert, etc.). We then randomly placed appropriate categories of models in different locations around the island, such as residential houses, trees, flying object, boats, vehicles, and commercial buildings. We think this scene is representative of what a common dynamic virtual world might look like. A rendering of the island scene is shown in Figure 5.2. The island scene comprises 2227 objects, 237 of which are unique. The scene is made up of 65 million triangles, requiring 3GB of (uncompressed) texture RAM and a download size of 577MB [2]. At that size, the entire scene would take about 8 minutes to download on a 10Mbit connection, or 50 minutes on a 1.5Mbit connection.

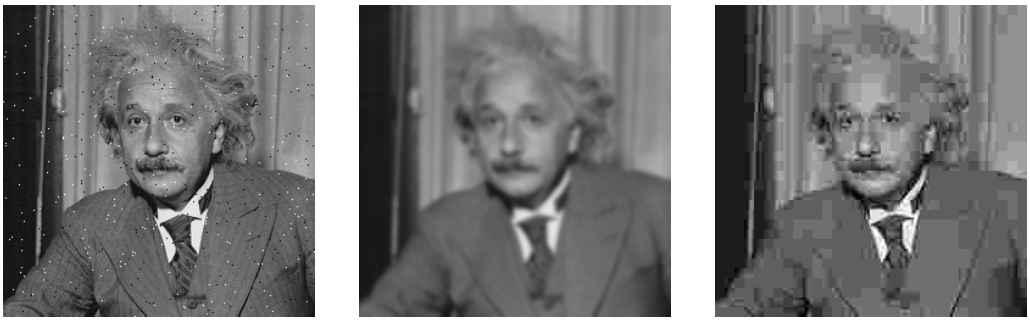
To simulate user movement through the scene, we recorded camera angles and motion paths of a human, with the goal of spanning a large range of the typical application use cases. The following is the list of motion paths we used, where the number in the label is the duration of the motion capture, in seconds.

- **brownian-45s** - moving straight in a direction for 5 seconds, then switching to a new, random direction.
- **meander-75s** - meandering along the hills of the island.
- **spinning-45s** - spinning in-place in the center of the island.
- **still-15s** - standing still facing a small hill.
- **still-90s** - standing still high above the island with all objects in view.
- **straight-fast-10s** - moving in a straight line quickly.

[2] Textures are compressed into the JPEG format before transmission, while numerical data is gzipped text.



(a) Original **(b)** Meanshift (M) **(c)** Contrast (C)



(d) Impulse (I) **(e)** Blur (B) **(f)** JPG (J)

Figure 5.3: Einstein images for evaluating image comparison algorithms.

- **straight-slow-30s** - moving in a straight line slowly.

Although we only use this single island scene with the seven motion paths for our evaluation, it would be actually be ideal to include several more scenes and hundreds of motion paths through each scene. Unfortunately, as mentioned before, openly-licensed, free, high-quality scenes are hard to acquire, and the running time of experiments increases linearly with the number of scene/motion path pairs. Since the framework outlined in Section 5.3 is available under an open-source license, it can easily be used in the future as more scenes become available.

5.4.2 Objective Perceptual Comparison

To evaluate different scheduling algorithms, an objective measure of how each algorithm is performing is needed. This is a well-studied problem in the vision and graphics literature. The classic approaches for comparing two images are to use the

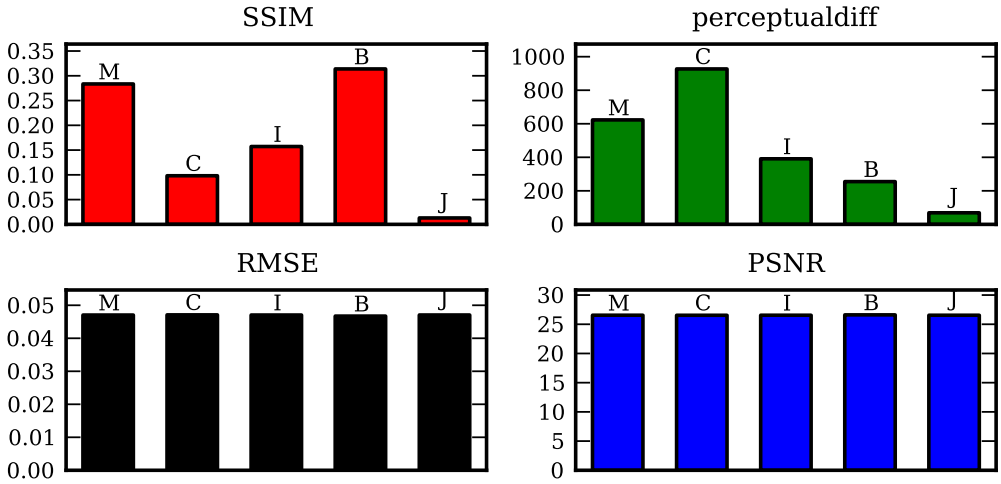


Figure 5.4: Image comparison algorithms evaluated on Einstein image dataset. Note: SSIM and PSNR values are inverted for easier comparison to perceptualdiff.

Root Mean Square Error (RMSE) or Peak Signal-to-Noise Ratio (PNSR). The problem with these methods is they don't take into account human perception. More recent methods such as SSIM [51] and perceptualdiff [55] try and estimate the human *perceptual* difference between two images. Using sample images of Albert Einstein (Figure 5.3) that have each been modified in a different way, we evaluate these different algorithms. The results in Figure 5.4 show that RMSE and PNSR are not useful at differentiating these images. SSIM and perceptualdiff agree (up to scale) on the error rates of Meanshift and JPG, but differ on their ordering of Contrast, Impulse and Blur. However, these standard test images are not representative of our workload.

To analyze how these algorithms compare in our domain, namely screenshots of a rendered 3D scene, we evaluate the four algorithms over time for a sample run of the download scheduler. The results are shown in Figure 5.5. Here we see a different story. RMSE and SSIM seem to be almost identical (with respect to their scale) and PNSR and perceptualdiff also show similar shapes. All of the metrics seem to compare about equally within our domain. We chose to use perceptualdiff as our

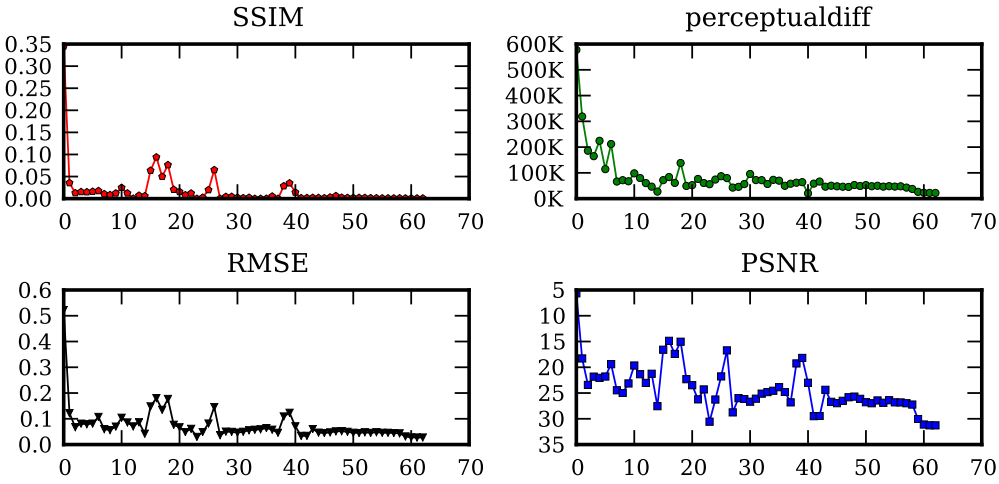


Figure 5.5: Image comparison algorithms for a sample run of the download scheduler. Note that lower values on the y-axis mean lower error.

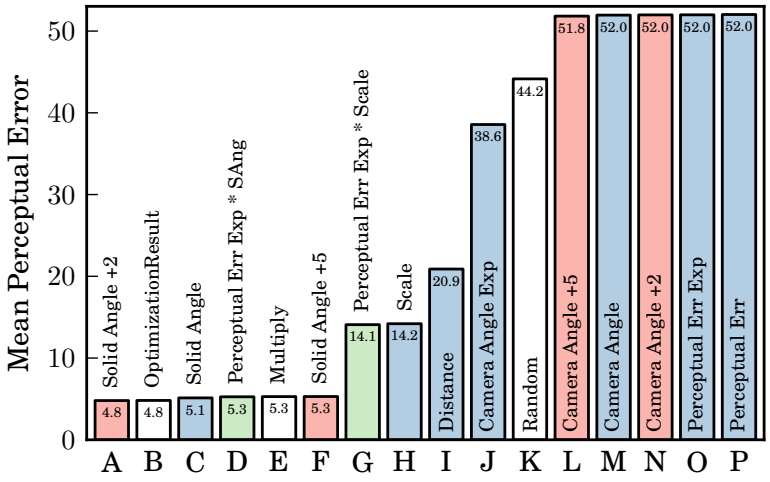


Figure 5.6: Comparison of download scheduling algorithms

evaluation metric because it provides an intuitive output value: the number of pixels that are perceptually different to a human observer, and also because it uses MPI to scale linearly with the number of cores in a machine.

5.4.3 Metric Comparison

To compare the different metrics outlined in Section 5.2.2, Figure 5.6 shows a comparison of several different download scheduling algorithms. Each bar is the mean of the seven motion paths from Section 5.4.1. Each motion path is the mean of three trials. The variance between each trial is low ($\approx 1\%$), indicating that the results are reproducible. A metric followed by "Exp" is the metric with an exponential falloff. We included exponential falloff versions of camera angle and perceptual error because these metrics tend to cluster around 1.0, so an exponential falloff helps to separate different values. A metric followed by "+N" is the metric predicted N seconds into the future using linear interpolation. Bar K, Random, is an algorithm that randomly chooses the next download, useful for a baseline comparison.

We see that Solid Angle (C), Scale (H), Distance (I) and Camera Angle Exp (J) are individual metrics that perform better than random. We also see that predicting metrics into the future (A, F, L, and N) is negligibly different than the metrics at the present time. The motion prediction does do slightly better for motion paths that are moving straight, but a more complicated non-linear approach to predicting future motion paths would be needed to perform well on other motion paths. We leave this as future work.

Perceptual Error by itself (O and P) is not a good metric for scheduling downloads. Multiplying Perceptual Error by both Scale (G) and Solid Angle (D) produce similar results as Scale and Solid Angle themselves. This is likely attributable to Perceptual Error being an estimate of error that doesn't take into account the user's viewing angle.

Multiplying all metrics together (E) performs about as well as Solid Angle, indicating that multiplying the multiple metrics together ends up being dominated by the scale of solid angle.

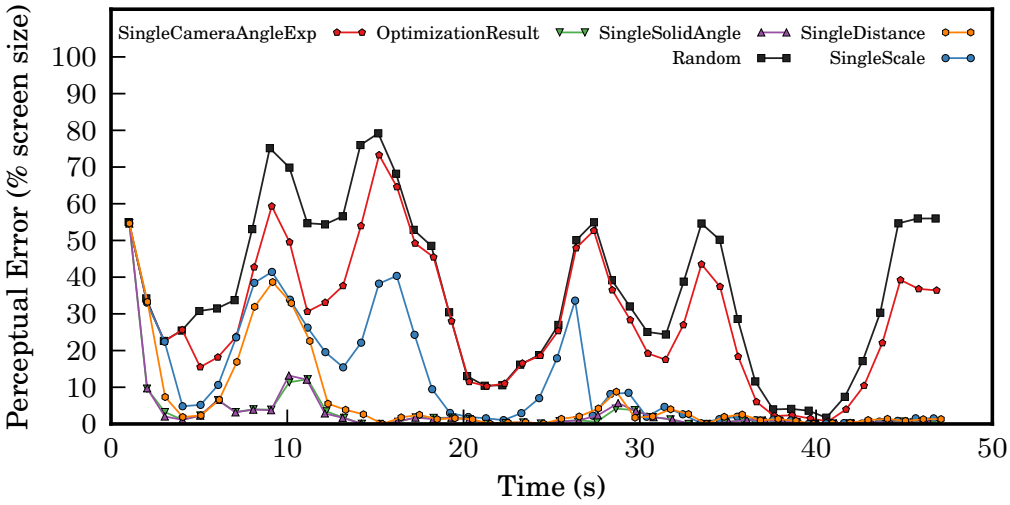


Figure 5.7: Example motion path run (Spinning), showing the perceptual error over time for several of the download scheduling algorithms.

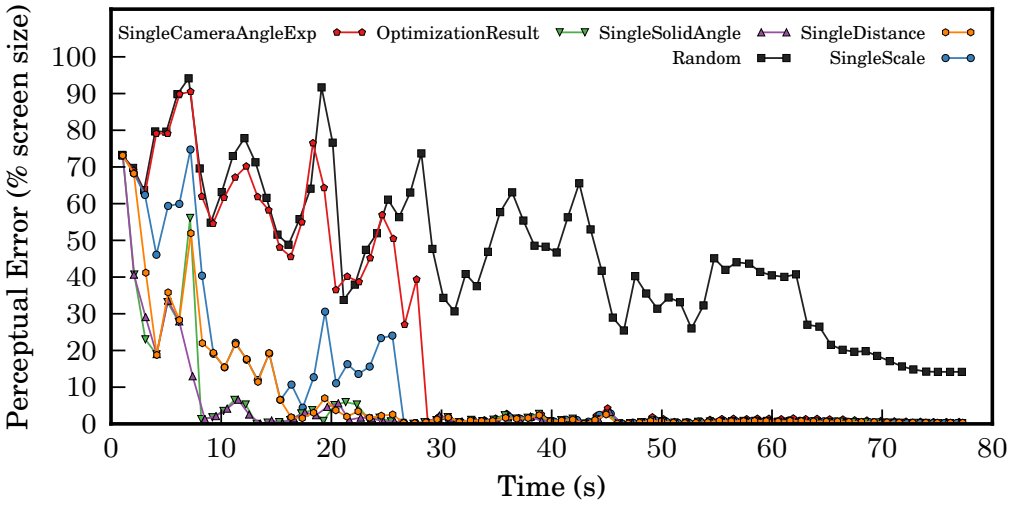


Figure 5.8: Example motion path run (Meander), showing the perceptual error over time for several of the download scheduling algorithms.

5.4.4 Linear Optimization

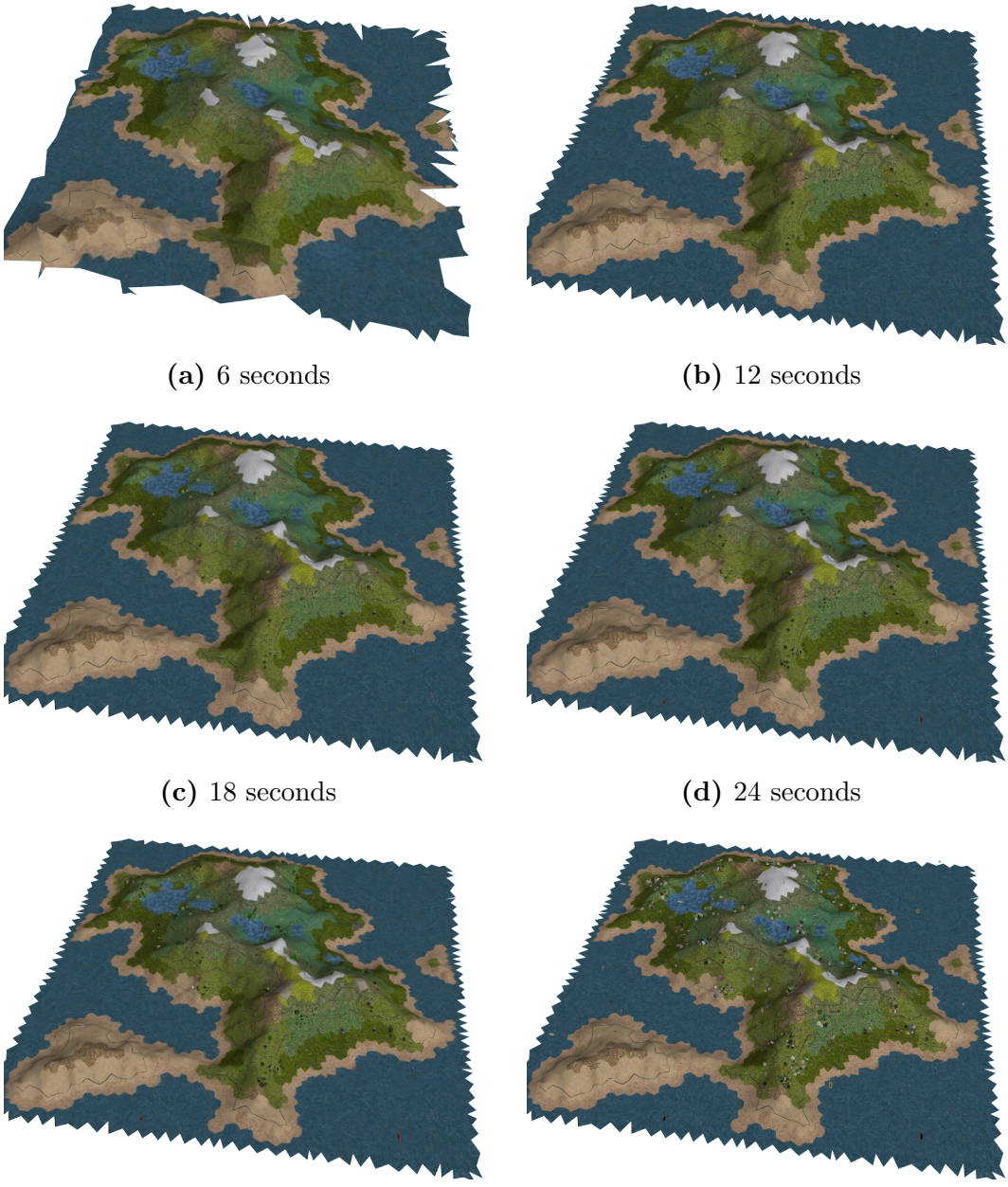
Given that we have several metrics that perform better than random, a linear combination of these metrics might perform better than any individual metric. Since the scale of each metric is independent, finding appropriate weights for each individual metric is non-intuitive. To find weights, we tried two different linear optimization techniques, BFGS [5] and Simulated Annealing [23]. Our objective function to mini-

mize for the optimization is the mean perceptual error over all motion paths, or the value of the bar in Figure 5.6. For the input variables, we decided to choose the four independent metrics that perform better than random: Solid Angle, Distance, Scale, and Camera Angle Exp. We initially set the weights for each metric to 1.0 and let the optimization algorithm choose a new set of weights for each iteration.

After running both optimization algorithms, we find that no linear combination of metrics performs better than the single Solid Angle metric. The optimization algorithms end up converging on weighting solid angle alone, while using a zero weight for all other metrics. This result is shown as bar F from Figure 5.6. We show two example motion paths with the perceptual error rate for each of the individual metrics, Random, and the result of the optimization algorithm in Figures 5.7 and 5.8. A clear oscillating behavior can be seen when the camera is spinning, while a strict downward trend is seen in the meandering motion path.

5.4.5 Example Screenshots

Example screenshots of the progressive scene loader compared with the full-quality island scene can be seen in Figure 5.9.



(a) 6 seconds

(b) 12 seconds

(c) 18 seconds

(d) 24 seconds

(e) 30 seconds

(f) full resolution

Figure 5.9: Example screenshots from a sample run of the progressive loader compared with a screenshot taken from the same location at full-resolution. The loader was connected to the content repository with a 10Mbit connection.

Chapter 6

Conclusion

6.1 Summary of Contributions

In this thesis, we have discussed the goals for providing a robust, scalable implementation of the persistence services required of a dynamic virtual world. The algorithms and techniques described have been implemented and are currently deployed in a production environment, serving the developers and users of the Sirikata metaverse platform.

Specifically, this thesis provided:

- A high-level overview of the complete Sirikata metaverse platform, broken down into its subcomponents.
- A description of the production-quality implementation of the persistence services for the Sirikata platform, currently running at open3dhub.com.
- A complete, robust conversion framework that automatically conditions 3D content into an efficient format for real-time rendering and transmission over a network. We devised algorithms for choosing a stopping point for existing su-

pervised algorithms, allowing the techniques to be executed without supervision, and we apportion the constrained texture space of 3D models efficiently, minimizing loss in quality.

- A framework for comparing scheduling algorithms for progressively downloading 3D models over a network. We used this framework with an objective measure of perceptual image quality loss to evaluate several possible metrics and algorithms for the scheduling of downloads. We found that a single, simple metric—solid angle—consistently outperforms all other algorithms we evaluated.

6.2 Future Work

There is still much work to be done with the Sirikata platform:

- Ongoing work is continuing to improve the scalability and effectiveness of Sirikata's space server. A new version of the Pinto object discovery mechanism described in Section 2.2.2 is in development that drastically reduces the number of aggregate meshes that need to be generated within the LBVH tree, gives flexibility to the object host to modify its object discovery queries, and reduces load on space servers.

- The generation of aggregate meshes using instance-aware simplification techniques is in development that will improve the efficiency of rendering large worlds.

- The conversion process outlined in Chapter 4 currently only handles the diffuse color channel of 3D models. More engineering effort is needed to implement techniques required for preserving other color channels, such as transparency, normal maps, and glow effects. The process could also benefit from a more

efficient texture atlas packing algorithm and more advanced compression techniques.

- The scheduling framework outlined in Chapter 5 was only evaluated on a single island scene and a handful of motion paths. As the Sirikata platform gains more users, more data will be available with which to re-evaluate the scheduling techniques.

Every piece of software developed for the Sirikata platform is available under an open, free software license. This enables other researchers to freely use the platform to evaluate their new ideas for improving dynamic virtual worlds. The Sirikata developers are proud to make our software available, and hope that our effort inspires an active community around the platform, with the eventual goal of realizing the dream of a truly flexible, scalable, immersive platform for hosting a metaverse.

Bibliography

[1] Apache httpd. `http://httpd.apache.org/`.

[2] Apache Solr. `http://lucene.apache.org/solr/`.

[3] J. L. Bentley. Multidimensional Binary Search Trees used for Associative Searching. *Communications of the ACM*, 18(9):509–517, 1975.

[4] T. Berners-Lee, R. Fielding, and L. Masinter. RFC 2396: Uniform Resource Identifiers (URI): Generic Syntax, 1998.

[5] R. H. Byrd, P. Lu, J. Nocedal, and C. Zhu. A Limited Memory Algorithm for Bound Constrained Optimization. *SIAM Journal on Scientific Computing*, 16(5):1190–1208, 1995.

[6] Cassandra. `http://cassandra.apache.org/`.

[7] Celery. `http://celeryproject.org/`.

[8] B. Chandra, E. Cheslack-Postava, B. F. T. Mistree, P. Levis, and D. Gay. Emerson: Scripting for Federated Virtual Worlds. In *Proc. CGAMES '10*, 2010.

[9] E. Cheslack-Postava, T. Azim, B. F. T. Mistree, D. Reiter-Horn, J. Terrace, P. Levis, and M. J. Freedman. A Scalable Server for 3D Metaverses. In *Proc. USENIX ATC '12*, 2012.

[10] J. Cohen, M. Olano, and D. Manocha. Appearance-Preserving Simplification. In *Proc. SIGGRAPH '98*, 1998.

[11] Django. `https://www.djangoproject.com/`.

[12] T. A. Funkhouser and C. H. Séquin. Adaptive Display Algorithm for Interactive Frame Rates During Visualization of Complex Virtual Environments. In *Proc. SIGGRAPH '93*, 1993.

[13] M. Garland and P. S. Heckbert. Surface Simplification using Quadric Error Metrics. In *Proc. SIGGRAPH '97*, 1997.

[14] M. Garland and P. S. Heckbert. Simplifying Surfaces with Color and Texture using Quadric Error Metrics. In *Proc. VIS '98*, 1998.

[15] M. Garland, A. Willmott, and P. S. Heckbert. Hierarchical Face Clustering on Polygonal Surfaces. In *Proc. I3D '01*, 2001.

[16] M. Goslin and M. R. Mine. The Panda3D Graphics Engine. *Computer*, 37(10):112–114, 2004.

[17] H. Hoppe. Progressive Meshes. In *Proc. SIGGRAPH '96*, 1996.

[18] H. Hoppe. Smooth View-Dependent Level-of-Detail Control and its Application to Terrain Rendering. In *Proc. Visualization '98*, 1998.

[19] H. Hoppe. New Quadric Metric for Simplifying Meshes with Appearance Attributes. In *Proc. VIS '99*, 1999.

[20] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle. Mesh Optimization. In *Proc. SIGGRAPH '93*, 1993.

[21] Khronos Group Inc., The. COLLADA - Digital Asset Schema Release 1.4.1 Specification (2nd Edition). `http://www.khronos.org/files/collada_spec_1_4.pdf`, 2008.

[22] J. Kim, S. Lee, and L. Kobbelt. View-Dependent Streaming of Progressive Meshes. In *Proc. SMI '04*, 2004.

[23] S. Kirkpatrick, C. D. Gelatt Jr, and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, 1983.

[24] U. Labsik, L. Kobbelt, R. Schneider, and H. P. Seidel. Progressive Transmission of Subdivision Surfaces. *Computational Geometry*, 15(1):25–39, 2000.

[25] meshtool. `https://github.com/pycollada/meshtool`.

[26] B. F. T. Mistree, B. Chandra, E. Cheslack-Postava, P. Levis, and D. Gay. Emerson: Accessible Scripting for Applications in an Extensible Virtual World. In *Proc. OOPSLA '11*, 2011.

[27] T. Muth. A Little Hard Drive History and the Big Data Problem. `http://tylermuth.wordpress.com/2011/11/02/a-little-hard-drive-history-and-the-big-data-problem/`, 2011.

[28] NVIDIA. SDK White Paper: Improve Batching Using Texture Atlases. `http://developer.download.nvidia.com/SDK/9.5/Samples/DEMOS/Direct3D9/src/BatchingViaTextureAtlases/AtlasCreationTool/Docs/Batching_Via_Texture_Atlases.pdf`, 2004.

[29] OAuth. `http://oauth.net/`.

[30] Open3DHub. `http://open3dhub.com/`.

[31] OpenID. `http://openid.net/`.

[32] Panda3D. http://www.panda3d.org/.

[33] A. Patel. Polygonal Map Generation for Games. http://www-cs-students.stanford.edu/~amitp/game-programming/polygon-map-generation/, 2010.

[34] pycollada. https://github.com/pycollada/pycollada.

[35] Python. http://www.python.org/.

[36] RabbitMQ. http://www.rabbitmq.com/.

[37] G. Ramanarayanan, K. Bala, and J. A. Ferwerda. Perception of Complex Aggregates. *ACM Transactions on Graphics (TOG)*, 27(3):60, 2008.

[38] Redis. http://redis.io/.

[39] D. Reiter-Horn. *Using a Physical Metaphor to Scale Up Communication in Virtual Worlds*. PhD thesis, Stanford University, 2011.

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

[41] S. Rusinkiewicz and M. Levoy. Streaming QSplat: A Viewer for Networked Visualization of Large, Dense Models. In *Proc. I3D '01*, 2001.

[42] P. V. Sander, J. Snyder, S. J. Gortler, and H. Hoppe. Texture Mapping Progressive Meshes. In *Proc. SIGGRAPH '01*, 2001.

[43] J. Shade, D. Lischinski, D. H. Salesin, T. DeRose, and J. Snyder. Hierarchical Image Caching for Accelerated Walkthroughs of Complex Environments. In *Proc. SIGGRAPH '96*, 1996.

[44] 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*, 2005.

[45] Sirikata CDN API Documentation. http://sirikata.com/wiki/index.php?title=CDN_API_Documentation.

[46] Sirikata CDN Project. https://github.com/sirikata/sirikata-cdn.

[47] I. Stoica, S. Shenker, and H. Zhang. Core-Stateless Fair Queueing: Achieving Approximately Fair Bandwidth Allocations in High Speed Networks. In *ACM SIGCOMM Computer Communication Review*, volume 28, pages 118–130. ACM, 1998.

[48] J. Terrace, E. Cheslack-Postava, P. Levis, and M. J. Freedman. Unsupervised Conversion of 3D Models for Interactive Metaverses. In *Proc. ICME '12*, 2012.

[49] D. S. P. To, R. W. H. Lau, and M. Green. A Method for Progressive and Selective Transmission of Multi-Resolution Models. In *Proc. VRST '99*, 1999.

[50] U.S. Federal Communications Commission. Broadband Performance OBI Technical Paper. Technical Report 4, Federal Communications Commission, Aug 2010.

[51] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. Image Quality Assessment: From Error Visibility to Structural Similarity. *IEEE Trans. on Image Processing*, 13(4):600–612, 2004.

[52] P. Wendell, J. W. Jiang, M. J. Freedman, and J. Rexford. DONAR: Decentralized Server Selection for Cloud Services. In *Proc. SIGCOMM '10*, 2010.

[53] WSGI. `http://wsgi.org/`.

[54] S. Yang, C. S. Kim, and C. C. J. Kuo. A progressive View-Dependent Technique for Interactive 3-D Mesh Transmission. *IEEE Trans. on CSVT*, 14(11):1249–1264, 2004.

[55] H. Yee, S. Pattanaik, and D. P. Greenberg. Spatiotemporal Sensitivity and Visual Attention for Efficient Rendering of Dynamic Environments. *ACM Trans. on Graphics (TOG)*, 20(1):39–65, 2001.