Rendering Massive Virtual Worlds

SIGGRAPH 2013 Courses

Anaheim, California, July 21-25 2013

Graham Sellers
Advanced Micro Devices

J.M.P. van Waveren
ID Software

Patrick Cozzi, Kevin Ring
Analytical Graphics

Emil Persson, Joel de Vahl
Avalanche Studios

Abstract

In recent years, connectivity of systems has meant that online worlds with huge streaming data sets have become common and widely available. From applications such as map rendering and virtual globes to online gaming, users expect online content to be presented in a timely and seamless manner, and expect the volume and variety of offline content to match that which is available online. Generating, retrieving and displaying this content to users presents a number of considerable challenges. This course addresses some real-world solutions to the problems presented in the rendering of massive virtual worlds.

Massive worlds present many daunting challenges. In our first talk, we introduce the two most prominent challenges: data management and rendering artifacts. We look at handling massive datasets like real-world terrain with out-of-core rendering, including parallelism, cache hierarchies on the client and in the cloud, and level-of-detail. Then we explore handling jittering artifacts due to performing vertex transform on large world coordinates with 32-bit precision on commodity GPUs, and handling z-fighting artifacts due to lack of depth buffer precision with large near-to-far distances. This serves as an introduction to the following talk, World-Scale Terrain Rendering.

Rendering small and medium scale terrain is fairly straightforward these days, but rendering terrain for a detailed world the size of Earth is much more challenging. In our second talk, we discuss the design and implementation of a terrain engine that can render a zoomed-out view of the entire globe all the way down to a zoomed-in view where sub-meter details are visible. We discuss processing "off the shelf" terrain data for efficient streaming and rendering, an asynchronous load pipeline for bringing chunks of terrain through a cache hierarchy, efficiently culling chunks of terrain that are below the horizon, driving terrain level-of-detail selection based on an estimate of pixel error, and more. With these techniques, we are able to achieve excellent performance even in the constrained environment of WebGL and JavaScript running inside a web browser.

Once we have dealt with the topic of storing and streaming huge amounts of content, we must next contend with production and generation of that content. The next talk will go through the production pipeline at Avalanche Studios and the issues encountered when filling Just Cause 2 with interesting content. It is mix of horror stories, good practices, and lessons learned and applied to titles currently in production. Issues discussed will include the reliability problems for the content tool-chain, the long turn-around times we had, undocumented and poorly understood data dependencies, and all the problems that followed with these. Then we will cover how we have solved these problems for our current content pipeline. We will also talk about our approach to authoring the landscape, vegetation and

locations for our large game worlds, and how we maintain productivity without sacrificing variation.

Many of the concepts discussed to this point address efficient generation, storage, retrieval and transmission of content. Recent advances in graphics hardware allow GPUs to assist in functions such as streaming texture data, managing sparse data sets and providing reasonable visual results in cases where not all of the data is available to render a scene. In the next talk, we take a deep-dive into AMD's partially resident texture hardware, briefly cover sparse texture extensions for OpenGL and then explore some use cases for the hardware and software features, including some live demos.

In our final presentation, we discuss the practical challenges of integrating support for hardware virtual texturing into a real-world game engine, idTech5, which powers RAGE and a number of other titles. We will describe cases where hardware virtual texturing 'just worked', and cases where more effort was required to integrate the technology into an existing engine, whilst maintaining support for software virtual texturing without loss of performance or features.

We assume that course participants are familiar with modern graphics rendering techniques, data compression, cache hierarchies and graphics hardware acceleration. We will discuss in some detail virtual memory systems, culling techniques, level-of-detail selection and other related techniques.

About the Authors

Graham Sellers
Advanced Micro Devices
graham.sellers@amd.com

Graham Sellers is the manager of the OpenGL driver team and a software architect at AMD. He represents AMD at the OpenGL ARB and Khronos Group and is responsible for the design and delivery of new features in AMD's OpenGL implementation, including extensions and new versions of the OpenGL API. He has authored over 20 OpenGL extensions, many of which are now part of the core API specification. He is also co-author of the OpenGL SuperBible and the OpenGL Programming Guide. He holds a Masters' degree in Engineering from the University of Southampton, UK.

Patrick Cozzi
Analytical Graphics, Inc. and University of Pennsylvania
pjcozzi@siggraph.org
Patrick Cozzi is coauthor of 3D Engine Design for Virtual Globes, coeditor of OpenGL Insights, and a contributor to GPU Pro 4, Game Engine Gems 2, and SIGGRAPH. At Analytical Graphics, Inc., he leads the graphics development of Cesium, a WebGL virtual globe. He teaches GPU Programming and Architecture at the University of Pennsylvania, where he received a master's degree in computer science.

Kevin Ring
Analytical Graphics, Inc.
kevin@kotachrome.com
Kevin Ring is coauthor of 3D Engine Design for Virtual Globes and the lead architect of STK Components at Analytical Graphics, Inc. In recent years, he has immersed himself in the problem of massive terrain rendering and analysis while developing the terrain and imagery rendering engine for Cesium, a WebGL virtual globe. Kevin received a bachelor's degree in Computer Science from Rensselaer Polytechnic Institute.

Emil Persson
Avalanche Studios

emil.persson@avalanchestudios.se
Emil Persson is the Head of Reseach at Avalanche Studios, where he is conducting forward-looking research, with the aim to be relevant and practical for game development, as well as setting the future direction for the Avalanche Engine. Previously, Emil was an ISV Engineer in the Developer Relations team at ATI/AMD. He assisted tier-1 game developers with the latest rendering techniques, identifying performance problems and applying optimizations. He also made major contributions to SDK samples and technical documentation.

Joel de Vahl
Avalanche Studios
joel.de.vahl@avalanchestudios.se
Joel de Vahl works as a Senior Engine Programmer at Avalanche Studios, focusing primarily on graphics and engine technology development. Previously, Joel worked as an Engine Programmer at Starbreeze Studios, focusing on lighting and rendering technology.

J.M.P. van Waveren
ID Software
mrelusive@idsoftware.com

J.M.P. van Waveren studied computer science at Delft University of Technology in the Netherlands. He has been developing technology for computer games for over a decade and has been involved in the research for, and development of various triple-A game titles such as: Quake III Arena, Return to Castle Wolfenstein, DOOM III and RAGE.

Course Outline

10 minutes: Introduction
Graham Sellers

1. Introduction to the course and its goals, course overview and introduction of all speakers
2. Introduction to virtual world rendering

30 minutes: Handling Planetary Scale Datasets
Patrick Cozzi

1. An overview of handling massive datasets such as real-world terrain with out-of-core rendering, including parallelism, cache hierarchies on the client and in the cloud, and level-of-detail.
2. Handling jittering artifacts caused by performing vertex transform on large world coordinates with 32-bit precision on commodity GPUs, and handling z-fighting artifacts due to lack of depth buffer precision with large near-to-far distances.

30 minutes: World-Scale Terrain Rendering
Kevin Ring

1. Processing and organizing data for efficient streaming to the client system: tile pyramids, mesh simplification.
2. Selecting the subset of terrain to render in a given view by estimating screen-space error.
3. Mapping terrain to an accurate model of the Earth, like the WGS84 ellipsoid.
4. Culling terrain, especially terrain that is below the horizon.
5. The client-side pipeline for loading and preparing terrain.
6. Using the imagery that's available, even when it's not ideal, including:
   (a) Re-projecting web Mercator imagery on the GPU.
   (b) Rendering multiple textures to cover a given region of terrain.

30 minutes: Populating a Massive Game World
Emil Persson, Joel de Vahl

1. Filling a massive game (Just Cause 2) with interesting content.
2. Content production and pipeline issues associated with massive procedural content generation.
3. Authoring landscapes and placing vegetation and other entities.
4. Maintaining productivity without sacrificing variety.

30 minutes: Hardware Virtual Texturing
Graham Sellers

1. Hardware architecture of AMD's 'partially resident texture' support
2. Driver support and software API
3. Use cases and technical demos
4. Future development

30 minutes: High Quality Software and Hardware Virtual Textures
J.M.P. van Waveren

1. Virtual texturing in idTech5 (RAGE)
2. Integrating support for hardware virtual texturing in idTech5
3. Supporting trilinear and anisotropic filtering in hardware and software
4. Addressing practical limitations to hardware virtual texturing

10+ minutes: Conclusion and Discussion
All speakers

Contents

Chapter 1

Introduction

This course material covers the practical details involved in rendering massive virtual worlds. At SIGGRAPH 2012, speakers Obert, van Waveren and Sellers presented an overview of hardware accelerated virtual texturing and how it was used in the video game RAGE, by id Software. Expanding on the topic, this course covers more diverse subjects such as content generation and creation, streaming and out-of-core rendering, and the practicalities of integrating very large virtual texture and geometry datasets into production pipelines and real-world rendering applications.

1.1 Virtual Worlds

In this context, we use the term *virtual* to mean data that is pageable, incomplete or where some or all of it is inaccessible at any given point in time. In general, these types of dataset fall into one of a few categories:

- Procedurally generated worlds consist of data that is created algorithmically, perhaps using an artist guided process either during production or just-in-time. Such data sets represent an overall design goal rather than something that must be conveyed with a great deal of accuracy.
- Real datasets such as satellite imagery, geospatial and survey data must be presented to the user with high fidelity and renderings should faithfully convey the world from which they were recorded.
- Artist generated data sets may not be as large as real-world data sets such as GIS data, but must nonetheless be stored, transmitted and rendered with accuracy demanded by the artist who authored them. Thus they present similar attributes and challenges as any other large data set.

Regardless of the source of the data set, similar challenges are encountered during their interactive or real-time rendering. For example, these data sets generally do not fit into a graphics processor's on-board memory and may not be entirely present in readily accessible form on the end-user's machine. Some or all of the data may be available over a network, or perhaps be stored locally in a heavily compressed form.

To further complicate matters, generation, pre-processing, authoring, managing and storing the volume of data required by modern virtual environment rendering applications is non-trivial. Such data sets can easily consume many terabytes of data when uncompressed. The number of discrete entities in a single virtual world can be very high — typically many tens of thousands, and perhaps into the millions. Clearly, having an artist design and place each individual tree, shrub and flower in a virtual forest is intractable. Therefore, we must find methods to generate huge numbers of entities, placing them semi-automatically, whilst allowing an artist to guide the process and edit the final results.

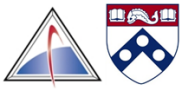
Chapter 2

Planetary Scale Datasets

Foundations for Rendering Massive Worlds

This is an incomplete draft. Get the latest version at http://cesium.agi.com/publications.html

Patrick Cozzi, @pjcozzi
Analytical Graphics, Inc.
University of Pennsylvania



SIGGRAPH2013

We're going to look at some fundamental problems when rendering massive worlds. These problems are interesting — or perhaps even annoying — in that a typical graphics engine may never need to deal with them. However, when we scale our world up to a massive size and still want fine-grained detail, precision problems start to manifest themselves as rendering artifacts.

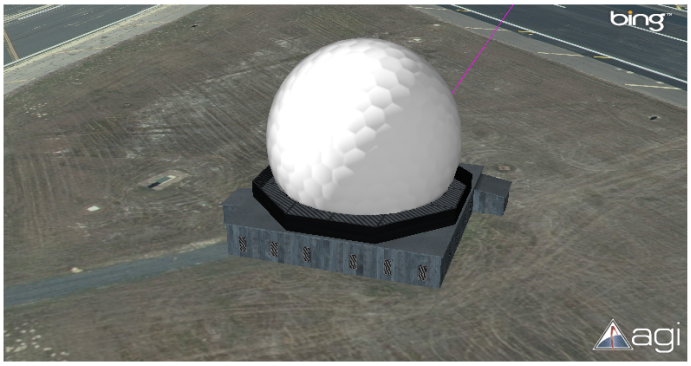
Over the years we've seen many forum posts like "I wrote a simple terrain engine and everything was great, but then I tried to cover world-wide terrain, and now there is z-fighting. Why?" and "I'm rendering the Earth using meters as units, and when I zoom in, static objects start to bounce around. Why?"

In this talk, we intuitively explain why, and present robust solutions that work for open

worlds and make very few assumptions, e.g., we can't necessarily put fog in the distant or ask artists to design a level such that the maximum view distance is only so far.

We've used these solutions in our 3D tools at Analytical Graphics, Inc. with great success. Some of them are well known and have been in use literally since I was in high school. Others are more recent. In both cases, I want to motivate their use and provide implementation tips.

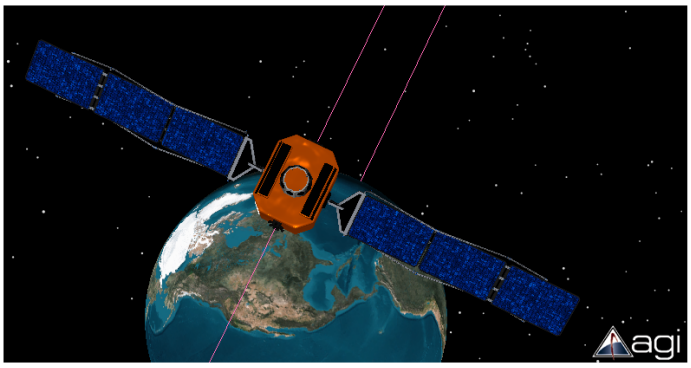
Our Work



SIGGRAPH2013

In our work at AGI, we simulate the real-world, from ground stations surrounded by high- resolution terrain and sub-meter imagery...

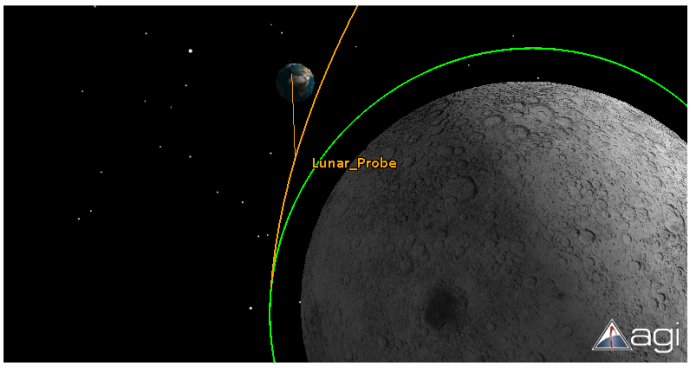
Our Work



SIGGRAPH2013

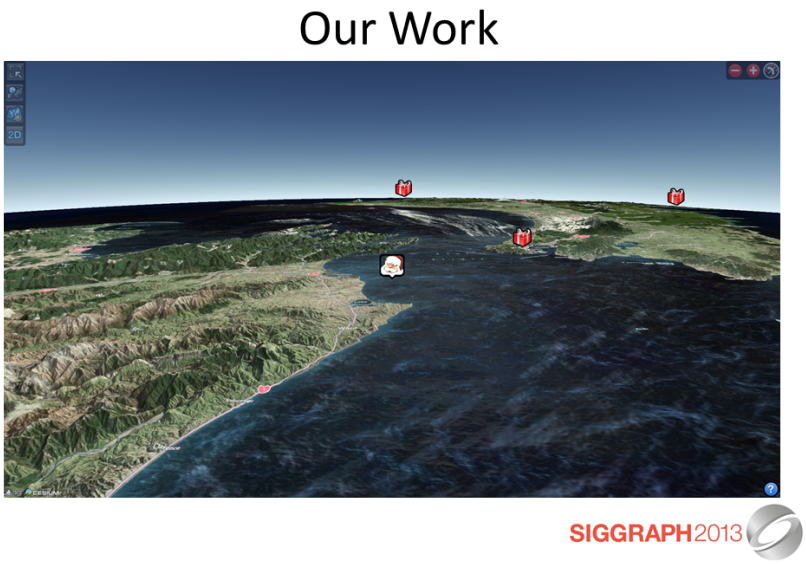
... to satellites in high-earth orbit at 40,000 km above the Earth...

Our Work



SIGGRAPH2013

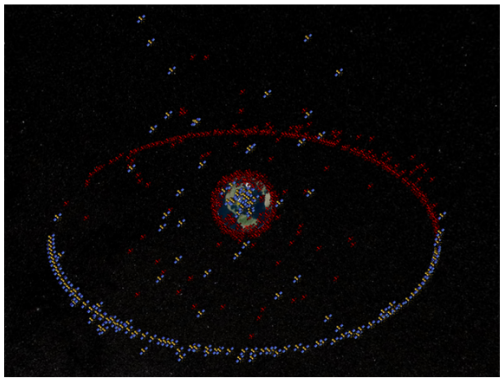
... to interplanetary missions to the moon or mars.



We've even helped NORAD with the very important job of tracking Santa on Christmas Eve.

These use cases require massive scale in terms of view distances and precision for coordinate systems. For example, we may be zoomed in close to a satellite orbiting the Earth, and still want to see the sun in the distance. Satellites have solar panels that point towards the sun.

Our Work



SIGGRAPH2013

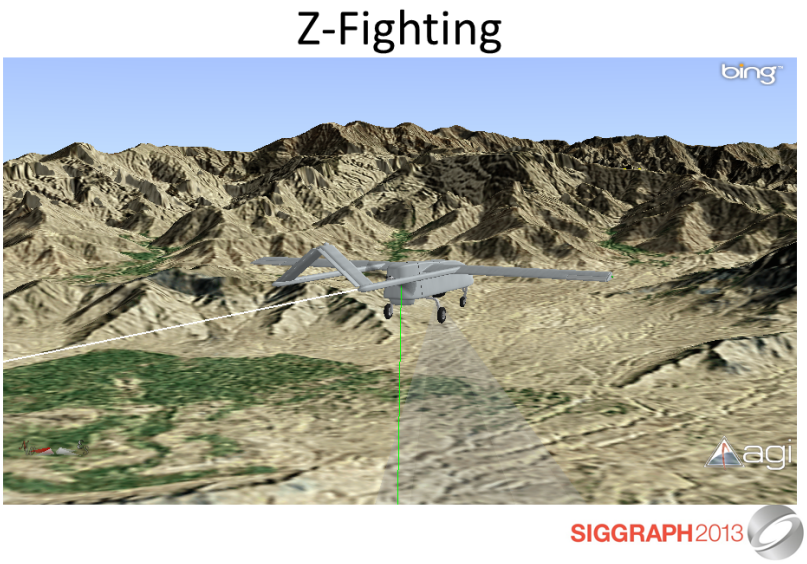
A lot of classic tricks like eliminating z-fighting by introducing fog in the distance to preserve a sane far-to-near ratio won't always work for us. For example, some of our users want to see all the active unclassified satellites in space at the same time.

Today, I'm going to present rendering techniques we use to deal with these type of scale while still allowing fine-grained detail. In particular, we are going to solve two types of rendering artifacts:

- z-fighting
- jittering

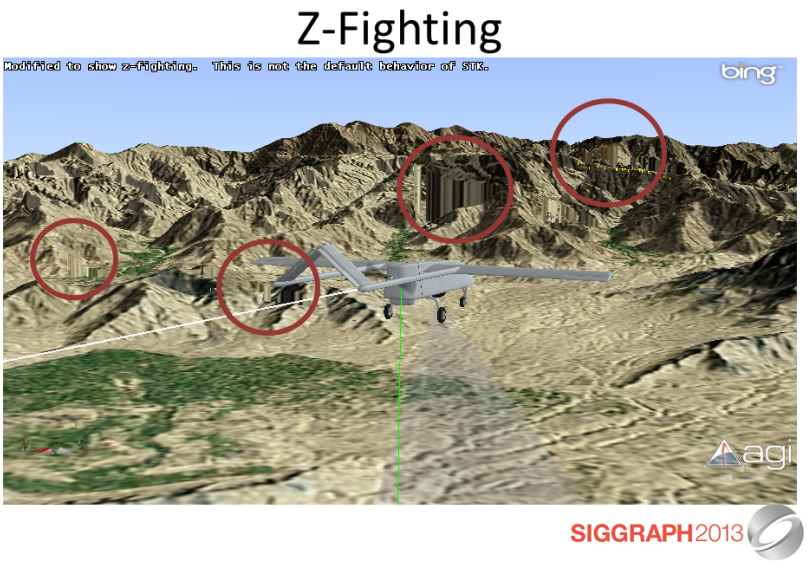
The presented solutions are very general and apply to geospatial visualization like work done by Esri and virtual globes like Google Earth and NASA World Wind. They also apply to massive- world games.

We'll start by looking at examples of z-fighting and jittering, then we'll look at the cause of each, and solutions.



Here we have a Shadow UAV flying a mission through the mountains.

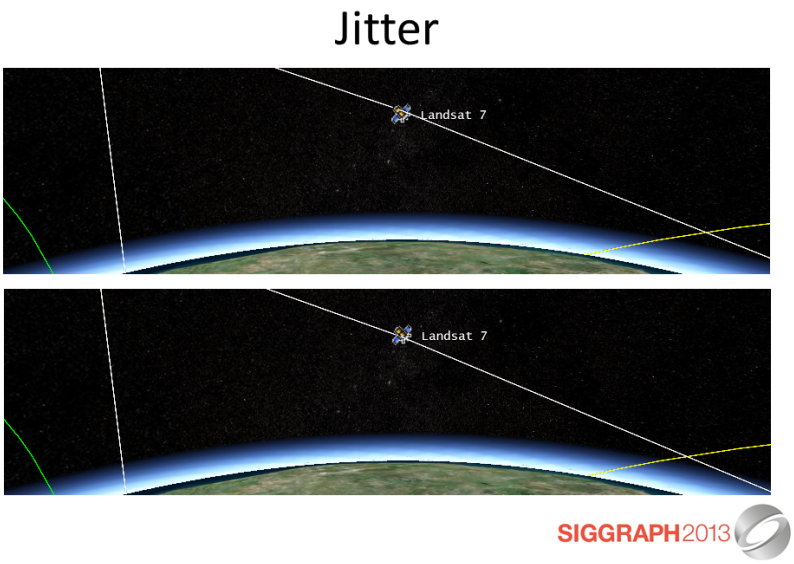
There are no z-fighting artifacts in this images, but if we use a close near plane and a very distant far plane, we see z-fighting artifacts like these...



There is z-fighting throughout the distant mountains. I circled a few of the major areas.

Z-fighting is even more irritating when things are moving as shown in the video [videos/z-fighting.wmv]. I promise that our software doesn't do this by default; I hacked it to make the video.

Notice how there is no z-fighting closer to the viewer, for example, with the UAV or nearby foothills.



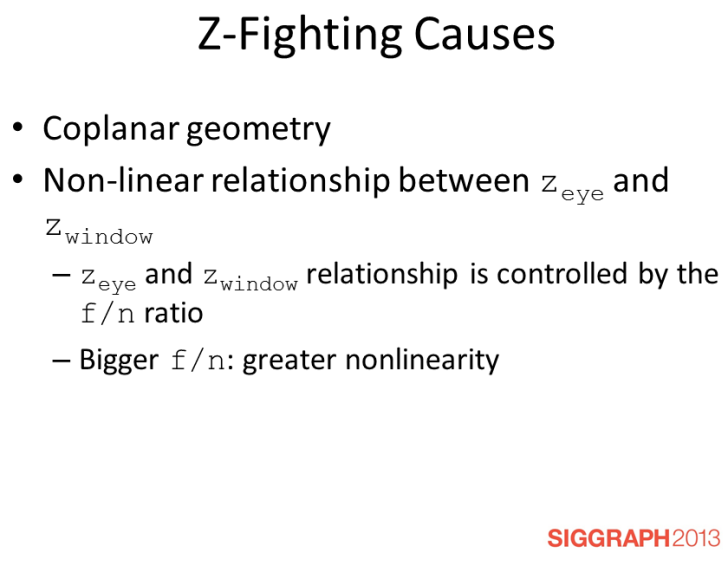
The other artifact we're going to look at today is jitter.

Here we have the Landsat 7, which is an imaging satellite in orbit at about 700,000 m above Earth. What's the different between the top image and the bottom image?

In both images, the billboard and text have the same world position. However, a small change in the viewer position resulted in the white polyline representing the satellite's orbit, to bounce — or jitter — as can be seen at the billboard, where the orbit on the top is above the billboard, and the orbit on the bottom is below. (This is also noticed where the orbit overlaps the yellow orbit on the right).

Like z-fighting, jittering is most noticeable and irritating when things are moving. In this video [videos/jitter.wmv], we see a red polyline that looks innocent enough. As we zoom in and rotate, it starts to bounce around. This polyline represents the boundary for Pennsylvania, and it is defined in WGS84, which is a coordinate system that has an origin at the center of the Earth. Now we zoom out and see the Landsat 7 satellite again. With the video it is easy to see the polyline bounce up and down.

For this video, I modified our engine to show jitter for the polyline, but not for the billboard and text.



Now let's take a closer look at z-fighting in massive worlds and solutions.

The classic z-fighting example is a piece of paper on a desk. If the paper and the desk are coplanar, we can see z-fighting artifacts due to floating-point round-off error. This particular case can be solved many ways including:

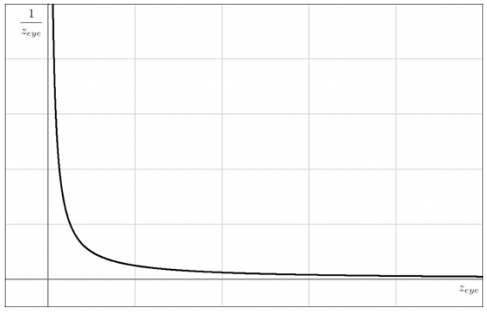
- Slightly adjusting the geometry, either when authoring or in a view-dependent way at runtime with a call like polygonOffset[1].
- Stenciling. Stencil out the paper. Render the desk. Render the paper.

However, z-fighting can occur when geometry is not coplanar, even when geometry is far apart — in some sense. This is due to the non-linear relationship between z in eye coordinates and z in window coordinates when using a perspective projection.

[1] http://www.opengl.org/archives/resources/faq/technical/polygonoffset.htm

Z-Fighting Causes

- z_{window} is proportional to $1 / z_{eye}$



SIGGRAPH2013

Image courtesy http://www.virtualglobebook.com

The relationship between z-eye and z-window is controlled by the far-to-near ratio. The bigger the ratio the greater nonlinearity.

Why? When using perspective, if we multiply out the model-view-projection matrix, perspective division, and viewport transform, z-window becomes:

```
Z-window = ((((f + n) / (f --- n)) + (2fn / z-eye(f --- n))) + 1) / 2
```

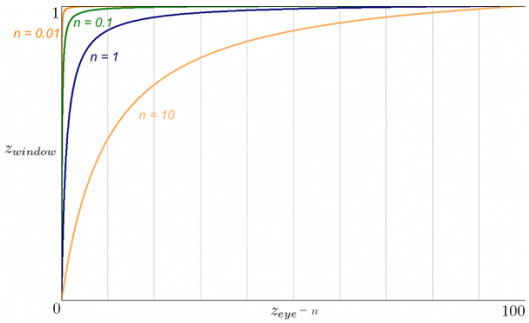
(in the common case when the depth range near is zero and far is one).

For a given perspective projection, everything on the right-hand side is constant except for z-eye, therefore z-window is proportional to 1/z-eye. This is the result of the perspective divide, which causes perspective foreshortening, making objects in the distance appear smaller.

As we can see in the graph, a small change in z-eye when z-eye is small results in a big change in 1/z-eye because 1/z-eye is a quickly moving function here. That same change in z-eye when z-eye is large results in a smaller change to 1/z-eye because 1/z-eye is slowly moving then. In the later case, a small change to z-eye may yield the same z-window, creating rendering artifacts.

Z-Fighting Causes

- Relationship between z_{window} and z_{eye} also depends on n and f.



SIGGRAPH2013

Image courtesy http://www.virtualglobebook.com

The relationship between z-window and z-eye also depends on the near and far distances as shown in this graph.

The x-axis is the distance from the near plan in eye coordinates, from 0 to 100. The y-axis shows z-window from 0 to 1. In all cases the view distance (f — n) is 100. Here we see that as n gets smaller, and hence the far-to-near ratio gets bigger, the precision gets pushed to the near plane.

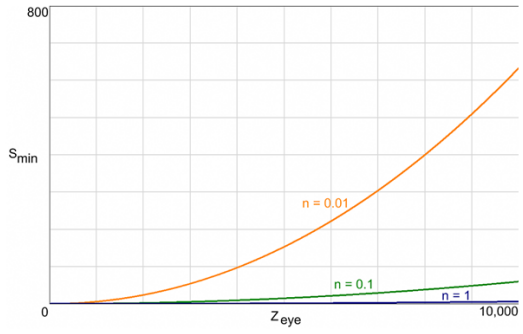
A ratio of 1,000 is commonly considered acceptable for a 24-bit fixed-point depth buffer [Akeley90]. In practice, we've gotten away with higher values depending on how far apart geometry is. However, 1,000 or even 10,000 is not going to cut it for our massive worlds where Earth's semiminor axis alone is over 6,300,000 meters.

In the video [videos/zfightingcauses.wmv], we keep pushing the near plane out to eliminate z- fighting between the plane and the globe. As we zoom in, we do not see z-fighting because precision gets better; however, if we zoom out, z-fighting will occur again.

Code: `https://github.com/virtualglobebook/OpenGlobe`
(see Chapter06DepthBufferPrecision)

Z-Fighting Causes

- Minimum Triangle Separation



SIGGRAPH2013

Image courtesy http://www.virtualglobebook.com

The minimum triangle separation, s-min, for a distance from the eye, z-eye, is the minimum world-space separation between two triangles required for correct depth occlusion.

[Baker99] provides an approximation to s-min for an x-bit fixed-point depth buffer:

```
S-min = (z-eye * z-eye) / ((2^x)n --- z-eye)
```

[Akeley06] also show that window coordinate precision, field of view, and error accumulated by single-precision projection, viewport, and rasterization arithmetic contribute to effective depth buffer resolution.

Z-Fighting Solutions

- Far-to-near tuning
- Remove distant objects / imposters
- Multiple frustums
- Complementary depth buffering
- Logarithmic depth buffer
- W-Buffer

SIGGRAPH2013

Multiple Frustums

- A few frustums can cover a large view distance
  - Frustum 1: 1 to 1,000
  - Frustum 2: 1,000 to 1,000,000
  - Frustum 3: 1,000,000 to 1,000,000,000

SIGGRAPH2013

In practice, we'll overlap the frustums a bit.

With a far-to-near ratio of 1,000, only a few frustums are needed to cover a large view distance. Three frustums can cover from 1 meter to a 1 billion meters.

Each frustum has the same field-of-view and aspect ratio, but a different near and far plane.

Multiple Frustums

- Simple implementation

```
foreach frustum in back-to-front order
{
   clear depth;
   find objects that overlap frustum;
   render objects;
}
```

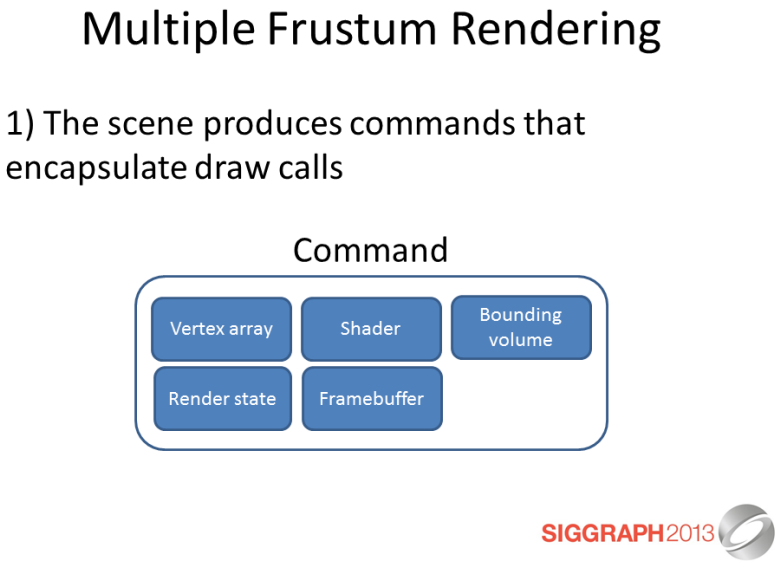
In practice, we'll overlap the frustums a bit.

SIGGRAPH2013

Given that most objects will only overlap a single frustum, this can be more efficient. Let's aim to:

- Use the fewest frustums possible based on the needs of the current view, not a fixed near and far plane
- Minimize the number of objects overlapping more than one frustum (by pushing out the near plane — and having a larger initial frustum)
- Minimize CPU overhead

Let's improve on this implementation.



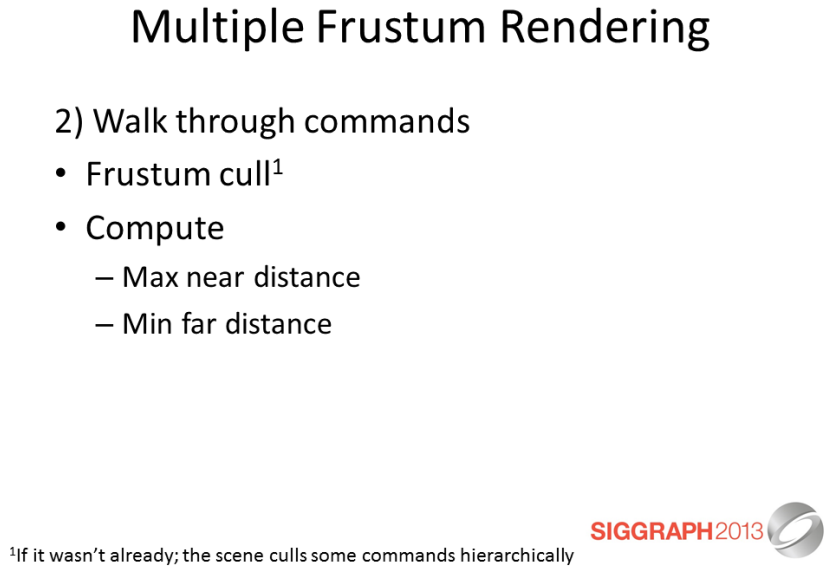
The scene produces commands for the renderer to execute. In other literature, a command is also called a "draw call" or a "batch." Commands may come from the terrain engine or other entitles like 3D models, billboards, or polylines.

A command has everything needed to execute a draw call using the underlying graphics API — WebGL in my case. Our engine has both DrawCommands for issuing drawElements/drawArrays calls and ClearCommands for clearing the framebuffer. Here, we're just concerned with DrawCommands, which have:

- Vertex array, offset, and count
- Shader program and its uniforms, including a model matrix
- Render state that defines the fixed-function state of the pipeline
- Framebuffer, which is the target of the draw call

Strictly speaking, the bounding volume is not used for issuing the draw call, but it is needed for determining what frustums a command belongs to. Any bounding volume could work, but we use spheres everywhere for now.

Code: https://github.com/AnalyticalGraphicsInc/cesium/blob/b16/Source/Renderer/DrawCommand.js



The near and far plane distances can varying frame-to-frame. The farther we can push out the near plane and the closer we can bring in the far the plane, the fewer frustums we will have. So instead of just using the application-defined near and far distances, we compute them dynamically using each command's bounding volume and the application-defined distances as extremes.

In our engine, we do both frustum culling and horizon culling, which is occlusion culling with the ellipsoid, as Kevin will discuss in the following talk.

Code: createPotentiallyVisibleSet() in
https://github.com/AnalyticalGraphicsInc/cesium/blob/b16/Source/Scene/Scene.js

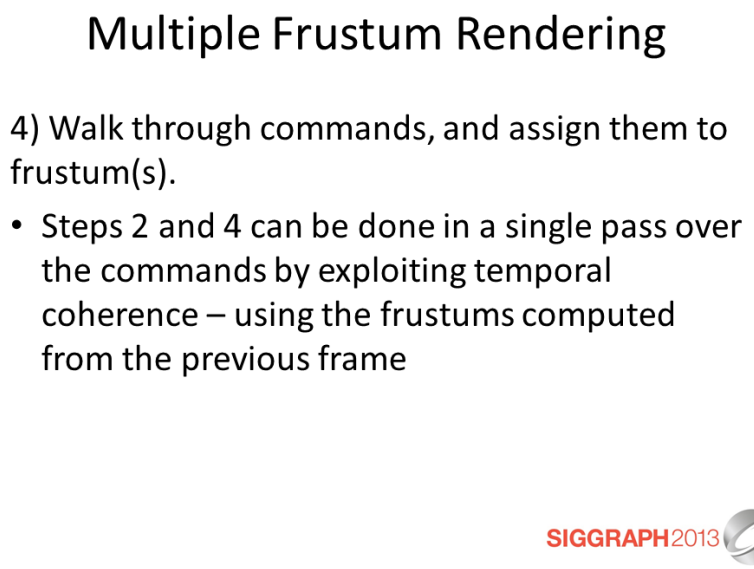
Multiple Frustum Rendering

3) Given desired far-to-near ratio, determine the number of frustums

```
var numFrustums = Math.ceil(
    Math.log(far / near) / Math.log(farToNearRatio));

for (var m = 0; m < numFrustums; ++m) {
    var n = Math.max(near, Math.pow(farToNearRatio, m) * near);
    var f = Math.min(far, farToNearRatio * curNear);
    // ...
```

Code: updateFrustums() in https://github.com/AnalyticalGraphicsInc/cesium/blob/b16/Source/Scene/Scene.js



Now that we know the length of each frustum, we can walk through the commands again, and assign to them frustums that they overlap. As we'll see some commands will wind up in more than one frustum.

So far, the renderer has made two passes over commands — the first pass culls and computes the frustums, and the second pass assigns commands to frustums. A typically application will have 100s to 1,000s of commands so it would be nice to be able to do this in a single pass. We can by exploiting temporal coherence. It is very likely that the frustums computed in the previous frame can be used in the current frame. We define our acceptance criteria as:

- desired near >= previous near
- desired far <= previous far
- desired number of frustums == previous number of frustums

Code: createPotentiallyVisibleSet() and insertIntoBin() in https://github.com/AnalyticalGraphicsInc/cesium/blob/b16/Source/Scene/Scene.js

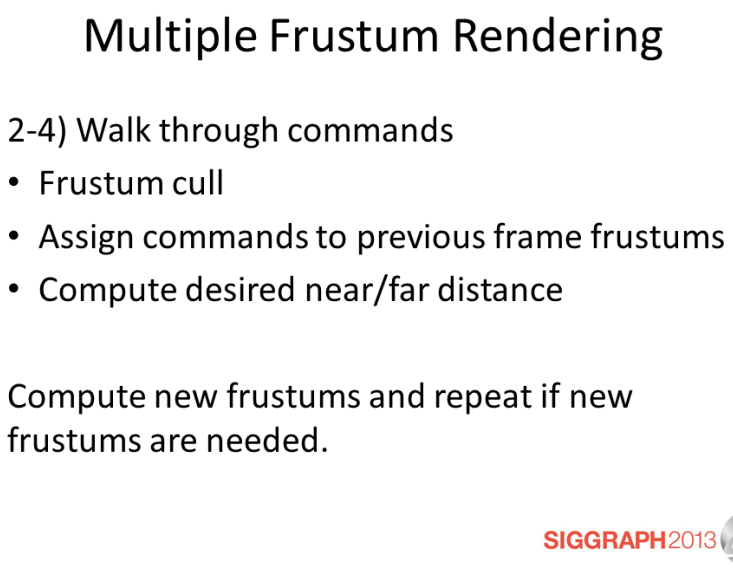
Multiple Frustum Rendering

SIGGRAPH2013

Coherence can be really good, especially when the user-defined near distance is used because the closest bounding volume intersects it because, for example, the viewer is deep in the mountains or following a satellite as shown here.

Coherence is not so good in other cases, like, when zooming in to an object because the desired near distance keeps getting smaller and smaller.

We could improve coherence by scaling the computed near/far to be slightly farther apart, but this runs the risk of requiring new frustums.



So Steps 2-4 are really combined into a single pass over the commands. We cull a command as before, then assign it to frustum(s) using the previous frame's frustum. As we walk through each command, we still compute the desired near/far distances. Then we use these to determine if the previous frame's frustums will work. If they won't we compute new frustums, and call the function again knowing that the computed frustum will work.

Multiple Frustum Rendering

- Render like we did before

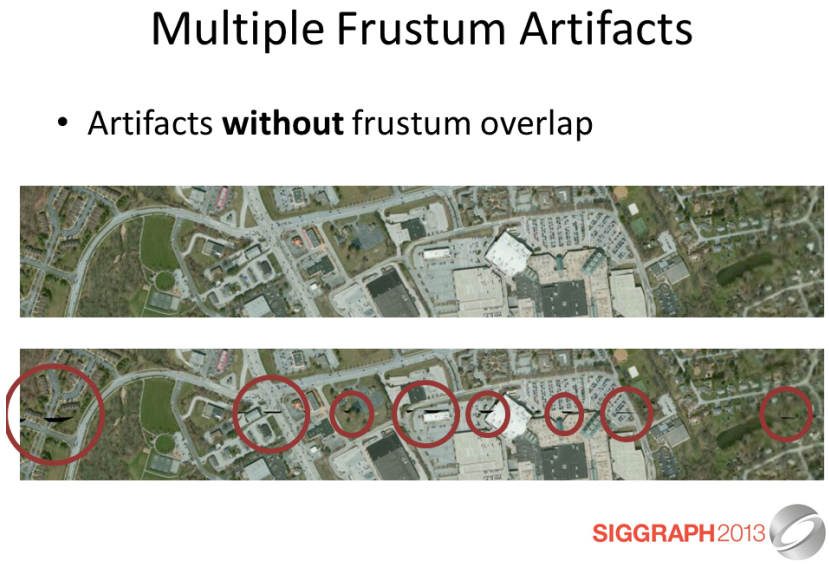
```
foreach frustum in back-to-front order
{
   clear depth;
   render commands overlapping this frustum;
}
```

SIGGRAPH2013

Multiple Frustum Artifacts

- Artifacts **without** frustum overlap

SIGGRAPH2013



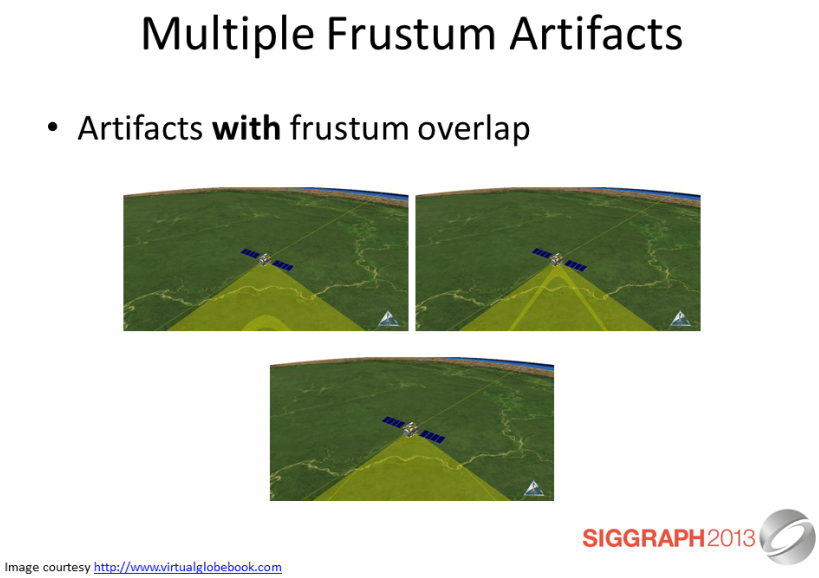
If adjacent frustums do not overlap slightly, we'll see tearing artifacts where the near plane of one frustum meets the far plane of the closer frustum. In the lower image here, we see several tears revealing the background color where two frustums meet.

When we first saw this artifact, we thought it was a driver bug, but we saw it across hardware vendors. With the near plane of a frustum equal to the far plane of the closer frustum, I can imagine one pixel artifacts, but these artifacts are much larger.

To solve these, we move the near plane of each frustum, except the closest, slightly closer.

Code: updateFrustums() in
https://github.com/AnalyticalGraphicsInc/cesium/blob/b16/Source/Scene/Scene.js

Video: videos/frustumOverlap.wmv

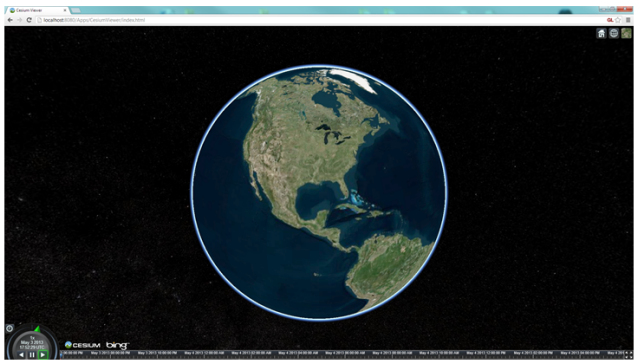


Overlapping frustums creates a new artifact for objects rendered with blending. In our engine, we render a lot of large translucent objects to represent sensors, e.g., a view volume of a camera attached to a satellite. Since these objects are large, they often overlap more than one frustum. Since they use alpha blending for translucency, the blending occurs twice where the frustums overlap leading to artifacts that can appear to slide back and forth when the viewer zooms.

It should be possible to eliminate these artifacts with the stencil buffer, but we haven't investigated it yet.

Multiple Frustum Performance

- Redundant draw calls



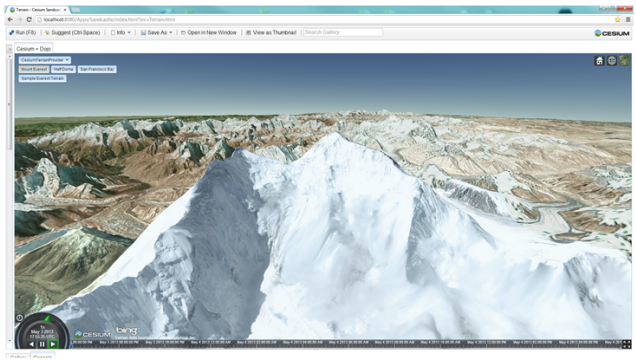
SIGGRAPH2013

When a bounding volume overlaps more than one frustum, not only does it have the potential to create artifacts, but it leads to commands being executed redundantly, that is, executing draw calls for the same object in more than one frustum. How, it's actually not as bad as you'd think. Let's look at a few different scenes.

For this full world view, we execute 20 commands (ignoring clears, the sky box, and 2D overlays). They all fit within one frustum because we are able to dynamically push the near plane back really far, which allows the first frustum to be large enough to include the entire globe.

Multiple Frustum Performance

- Redundant draw calls

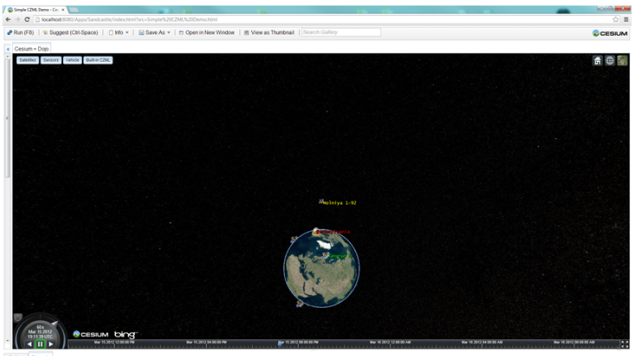


SIGGRAPH2013

Here we are zoomed in much closer to the ground look at Mount Everest. For this scene, we execute 106 commands (horizon views are always a challenge; note there is no fog here allowing us to not draw distant tiles). The good news is 104 commands are executed once, and only two commands are executed twice.

Multiple Frustum Performance

- Redundant draw calls



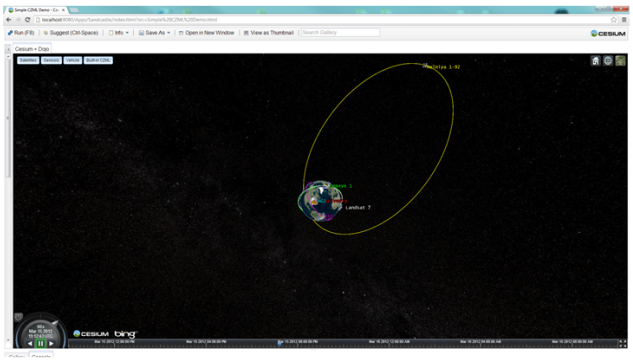
SIGGRAPH2013

We're not always this lucky with redundant calls. For example, we are high in space in this scene. We need three frustums total.

This scene only needs six commands, but two of them execute in all three frustums. Can you guess which? The billboards for the satellites are batched together as are the labels for the satellites. Since the bounding sphere for these includes the Molniya orbit high in space and right in front of the viewer — which peaks at 40,000 km above the globe, and the Geoeye in low earth orbit. This creates a huge bounding volume that overlaps all three frustums.

Multiple Frustum Performance

- Redundant draw calls

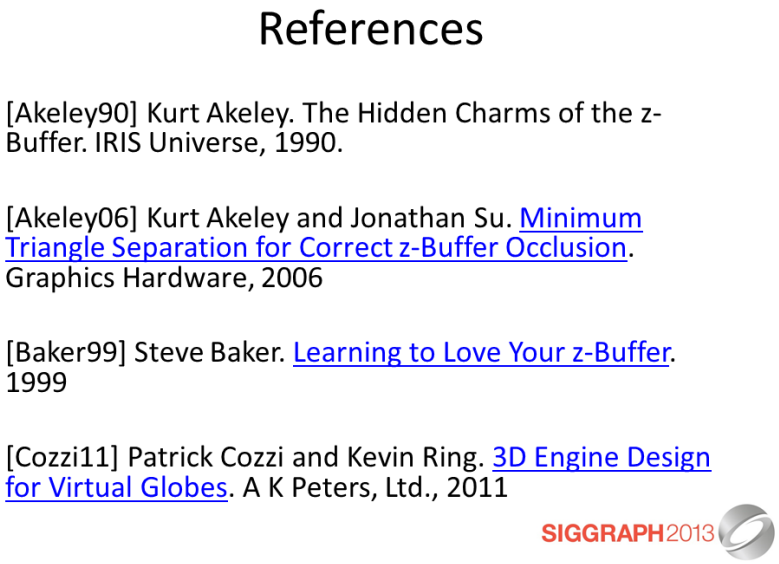


SIGGRAPH2013

Here's another screenshot showing the orbit line for us to better appreciate the scale.

These shows that there is tension between batching and culling even more so when using multiple frustums. Traditionally, to make the best use of the GPU, and minimize CPU overhead, we always try to reduce the number of draw calls we make during a frame by batching. When we introduce multiple frustum rendering, batching generally increases bounding volume sizes, making it more likely that a command (batch) will need to be executed in more than one frustum, which increases the number of draw calls. Given that applications built on our engine are almost always CPU bound, we still see a win by batching aggressive even if it leads to some redundant draw calls.

Ed did our original multi-frustum implementation in the late 90s. Dan did most of the recent implementation work in Cesium, our WebGL engine.



In addition to these more formal references, also see the implementation notes with pseudo- code for our engine - https://github.com/AnalyticalGraphicsInc/cesium/wiki/Data-Driven-Renderer-Details.

Chapter 3

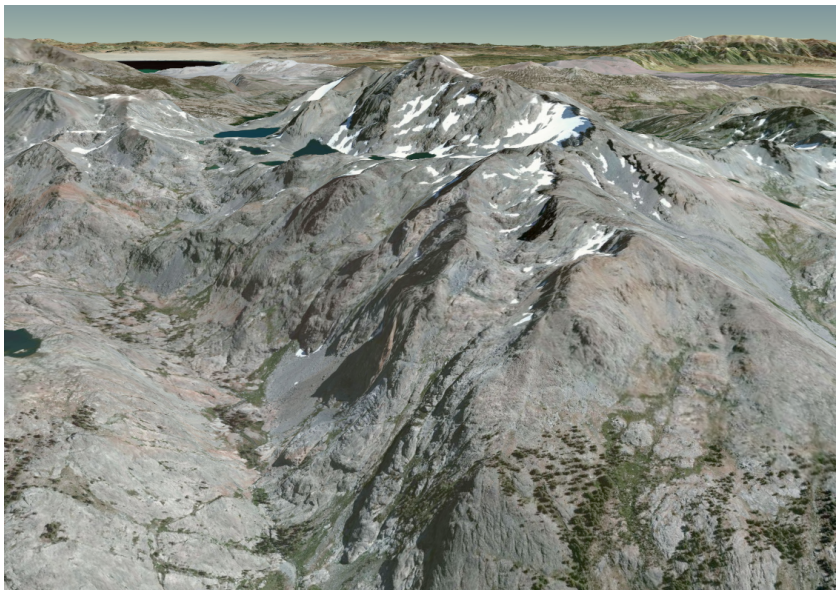
World-Scale Terrain Rendering

The most up-to-date version of these notes can be found at http://cesium.agi.com/publications.html.

3.1 Introduction - Massive worlds render massive quantities of terrain

Terrain datasets for massive worlds — especially those that aim to represent the massive world we call Earth — can easily measure in the terabytes. Add in detailed textures for the surface, such as color maps derived from satellite imagery or aerial photography, and it is not at all uncommon to see datasets measured in the hundreds of terabytes. Such datasets are much too large to fit in memory, and even too large to fit on a local system. Working with such enormous datasets requires that we process and organize the source data for efficient streaming from a remote server, to disk or memory on a local client, and finally into GPU memory for rendering. We discuss how we solved these problems in Cesium, an open source virtual globe that runs inside a web browser without the need for a plugin.

Cesium renders an accurate model of the Earth, from a global view where the entire planet is visible down to the street level where individual houses, cars, and trees are visible. The terrain surface is streamed as a mesh or heightmap from remote servers, and overlaid with multiple layers of imagery from different sources, such as Web Map Service (WMS) servers, ArcGIS MapServers, Bing Maps, and more. The imagery need not share the same extent, tiling scheme, or even map projection as the terrain, nor do all the imagery sources need to share these attributes with each other. Cesium combines the disparate imagery on-the-fly, applies adjustments to each layer independently such as hue, saturation, and gamma, and renders the completed scene inside a web page at well over 60 FPS even on modest hardware.



3.2 Organizing and processing terrain data

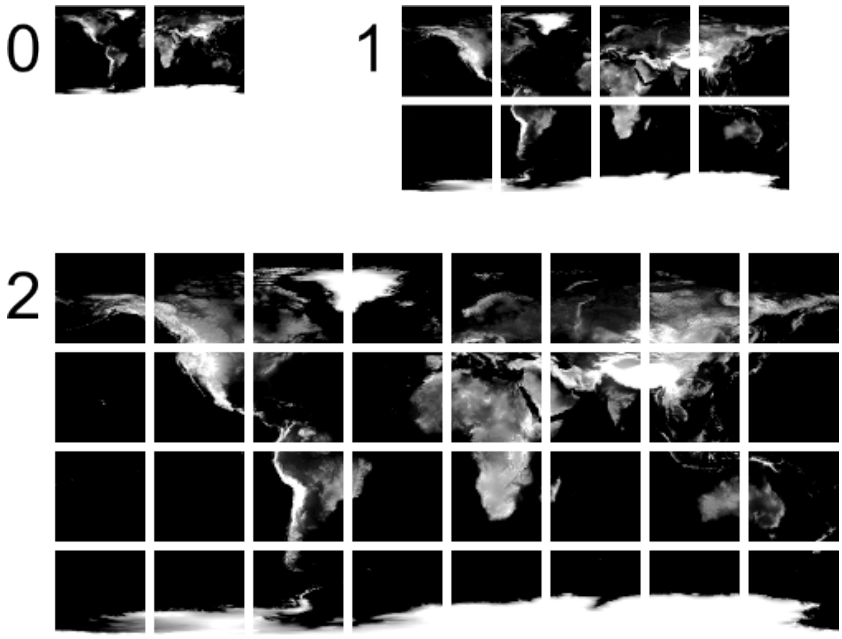
Real terrain data for Earth and other planets is collected using satellite and aerial instruments. For example, the Shuttle Radar Topography Mission (SRTM) obtained elevation data for most of the Earth's surface with an instrument flying onboard the Space Shuttle Endeavor. Such datasets are usually provided as giant heightmaps, or perhaps as a large collection of smaller heightmaps, where height samples are arranged in a regular grid in some projection.

It is not feasible to render such data directly. Instead, we preprocess it into a form that is amenable to streaming in small, multi-resolution chunks. For this sort of real-world terrain data, which conforms to an ellipsoidal model of the Earth and does not have any overhanging sections anyway due to the way it is collected, a quadtree is a very natural spatial data structure for managing culling and level-of-detail. Each node in the quadtree, known as a tile, represents a subset of the terrain at a particular resolution. The entire quadtree is known as a tile pyramid, because it has a small number of tiles at the root, usually between one and four, and millions or perhaps even billions of tiles at the leaves.

For a world extruded from a plane, it is straightforward to use a quadtree to uniformly divide the world. For a spherical or ellipsoidal world like Earth, however, the quadtree only uniformly divides a 2D projection of the world. The choice of map projection is important, because no projection can represent an ellipsoidal world without some distortion.

We use a simple geographic projection, also known as equidistant cylindrical or plate carrée, for our tile quadtree. This leads to singularities at the poles, where all samples in a row of the heightmap map to a single point on the globe. This is generally acceptable for Earth, however, and is easy to work with in fragment shaders during rendering. More

sophisticated approaches are possible without changing the fundamental rendering algorithm, such as the Ellipsoidal Cube Maps described by Lambers and Kolb[1].



A single terrain dataset at rendering time is built from multiple input datasets. For example, one terrain dataset might be a mosaic of GTOPO30 data covering the entire world, SRTM data between -60 and 60 degrees latitude, and the National Elevation Dataset (NED) for the United States. At rendering time, we work with a single preprocessed tile pyramid and the many sources of the original data do not influence the rendering algorithm. This is in contrast to our handling of imagery, where multiple imagery sources are mosaiced at rendering time using multitexturing.

The process for incorporating a new source dataset into the tile pyramid is as follows:

1. Compute the deepest level in the tile pyramid to be populated by the source data. This is a function of the source data resolution and the desired maximum number of samples in a single tile at the deepest level.
2. If the source data does not cover the entire world, compute the portion of the tile pyramid overlapped by the source data.
3. Execute a depth-first, post-order traversal of the overlapped portion of the tile pyramid, down to the deepest level to be populated, so that leaf tiles are processed first and parent tiles are only processed after their children are processed.
   (a) For each leaf (deepest-level) tile:
      i. Determine the set of height samples in the source that overlap the tile.

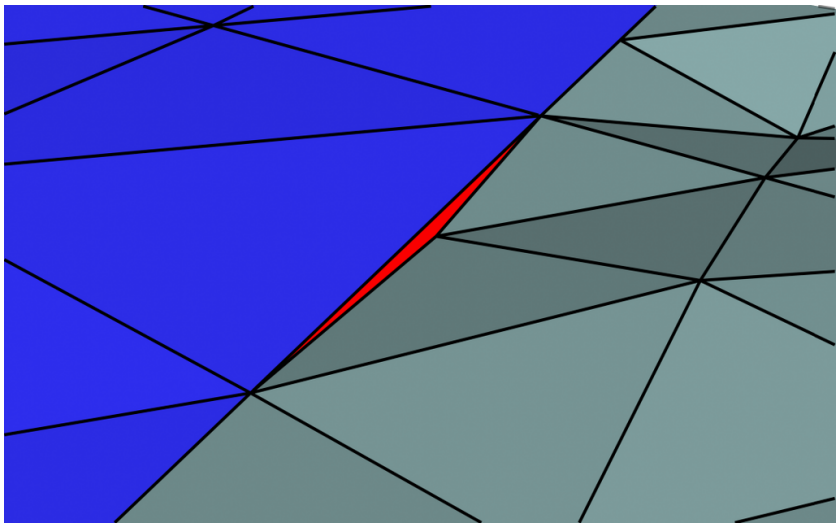
[1] http://ecm-planet.sourceforge.net/lambers12ecm.pdf

ii. Transform each sample to ellipsoid-centric Cartesian coordinates, usually WGS84 for Earth. Typically, the projected (X, Y, Height) coordinates are transformed to geodetic (Longitude, Latitude, Height) before they are transformed again to ellipsoid-centric (X, Y, Z) coordinates.

iii. Add the sample Cartesian position to the tile mesh as a vertex.

iv. Add additional vertices at the edges of the tile by interpolating, because the edges and corners are unlikely to align with samples in the source dataset.

v. Add a skirt around the perimeter of the tile. During rendering, the skirt will hide cracks between adjacent tiles with different levels of detail.



vi. Connect vertices to form faces, using a regular triangulation of the source grid.

vii. Store the finished tile on disk.

(b) For each non-leaf (parent) tile:

i. Build a new mesh that is the sum of all of this tile's child tile meshes, minus the skirt.

ii. Simplify the mesh by contracting vertex pairs until the desired geometric error limit is reached. The geometric error limit is discussed below.

iii. Add a skirt around the perimeter of the tile.

iv. Store the finished tile on disk.

This entire process is highly amenable to parallelization, by both utilizing multiple threads on a single machine and by utilizing multiple machines in a cluster. While preprocessing the terrain, we also compute and store per-tile metadata to aid in rendering, such as minimum and maximum heights, bounding volume, and a horizon occlusion point.

The horizon occlusion point is a proxy for the tile that we can use in occlusion culling against the Earth ellipsoid. If the horizon occlusion point is below the horizon, we can be certain that the entire tile is below the horizon as well. The computation and use of the horizon occlusion point is discussed in more detail in the *Horizon Culling* section.

We choose the target geometric error of the root level — level zero — by answering a question: if a root tile contained a regular grid of vertices with a chosen number of rows and columns, what is the worst-case geometric error of this representation relative to the real world? We can make a worst-case estimation of this error without getting into the details of the actual topology of the world. Imagine that Mount Everest rose straight out of the Mariana Trench. And now imagine that our grid of vertices happened to have two adjacent samples in the Mariana Trench, such that our aqueous Mount Everest is not represented at all. Certainly that's a worst case scenario in terms of the geometric error of our terrain representation. The height error would be the difference in height between Mount Everest and the Mariana Trench, or about 20 kilometers. The geometric error is, in fact, much higher, however, because we need to take into account the curvature of the Earth. For a grid width of 65 vertices at the equator, the maximum error is approximately 150 kilometers, and this is the error we target while simplifying tiles in level zero.

For levels greater than zero, the geometric error limit is determined by a simple relationship. Each deeper level in the tile pyramid has half the geometric error of the level above it. At level 18, the geometric error drops below one meter. For tiles at all levels, vertex pairs are collapsed until collapsing the next pair would cause the estimated error of the tile to cross the maximum allowed for tiles at this level in the tile pyramid.

Because the tile pyramid is built from multiple input datasets, and those datasets themselves may have regions of no data (voids), the maximum depth of the tile pyramid varies across the globe. We use a separate table of contents file, generated during the preprocessing step, to inform the rendering engine of which tiles are present.

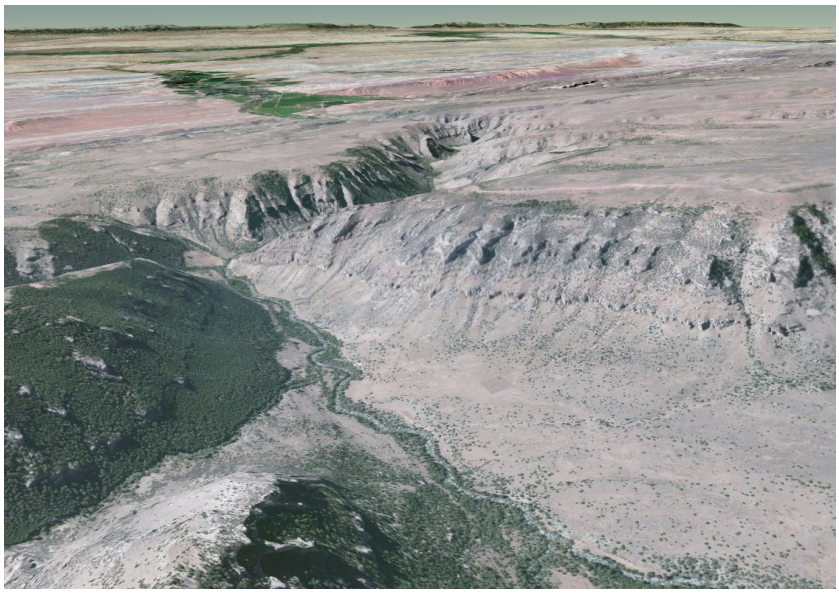
3.3 Tile selection for rendering

Our preprocessing work does not reduce the absolute size of the terrain dataset; in fact, it increases it. All that work up front, however, puts us in a good position to be able to very efficiently select a subset of the terrain to render in any given frame.

Our rendering algorithm closely follows the "Chunked LOD" approach presented by Thatcher Ulrich at a SIGGRAPH course in 2002[2].

Tile selection proceeds recursively from the root of the tile pyramid. The entire tile pyramid is not expected to be resident on the client, but portions of it will instead be streamed as needed from a remote server. We discuss that process in detail in the *Life of a*

[2] http://tulrich.com/geekstuff/chunklod.html



Tile section.

For each visited tile, the first step is to perform simple view-frustum culling. If the tile lies entirely outside of the current view frustum, we don't need to render it or visit any of its children. Because our terrain is mapped to a globe, we also perform another type of visibility culling: horizon culling. The concept is simple: if we can detect that the tile is entirely below the horizon of the ellipsoid, as viewed from the current camera position, we don't need to render it. See the *Horizon Culling* section for more details.

Once we have determined that a tile is visible, we really have two options: we can render this tile and stop our traversal of its branch of the quadtree, or we can refine. Refining means continuing traversal with this tile's child tiles, and rendering them, or their descendents, instead. Effectively, if we choose to refine, we are increasing the rendered level-of-detail for the region of the globe covered by this tile.

The standard technique used in hierarchical LOD algorithms, and the one used here, is to drive refinement by an estimate of the error, in pixels on the screen, that would result from rendering this tile rather than refining. We choose the tolerable screen-space error for our application, usually a small number like one or two pixels, and we refine whenever the estimated screen-space error for the tile is greater than the tolerance. By varying this tolerance, we are able to trade rendering accuracy for performance.

Screen-space error (SSE) is estimated by projecting the estimated geometric error of the tile into screen space, using this standard LOD equation:

$$SSE = \frac{\varepsilon x}{2d \tan \frac{\theta}{2}}$$

Where ε is the geometric error of the tile in meters, x is the viewport width in pixels, θ is the camera's field-of-view angle in radians, and d is the distance to the tile.

The field-of-view angle of the camera and the width of the viewport are known. We also know the geometric error because it is a simple function of the tile's level in the tile pyramid. Because we only know the maximum geometric error, not the local error throughout the tile, we conservatively assume that the point on the tile closest to the viewer has the maximum error. Even the distance to the closest point on the tile is expensive to compute, however. Instead, in our LOD equation, we use the distance to the closest point on a bounding volume that bounds the tile.

While bounding spheres are useful for culling, we've found them to be very poor for estimating distance for LOD selection. The problem is that the viewer is very frequently inside the bounding sphere of candidate tiles as a result of the sphere extending far above and out from the edges of the tile. When the viewer is inside the bounding sphere, a conservative estimate of distance to the region of the tile with maximum geometric error is literally zero; the viewer could be right on top of it. In that case, we must refine, because the projected error is infinite.

In Cesium, we use four planes to bound each tile, one at each horizontal boundary of the tile, plus a curved surface at the maximum height of the terrain in the tile. The two planes on the Western and Eastern edges of the tile tightly bound the tile, but planes on the Northern and Southern boundaries are approximations due to the surface normals at those boundaries not lying in a plane. The distance from the fifth, curved, surface is easily computed by ransforming the viewer position to a height above the ellipsoid and then subtracting the tile's maximum height.

We can not render a tile that is not yet loaded, nor can we render a tile where any of its four siblings in the quadtree are not loaded. If the parent cannot be rendered, either, we will continue up the quadtree until we find a suitable tile to render. As more data is loaded, the visual quality of the globe will improve, but at no point do we suspend rendering to wait for data to load.

Having selected the tiles to render this frame, it's now time to actually render them. In general, we issue one draw command per tile. Multiple color textures, a specular mask, and other effects are applied in a single pass. In some cases, a single draw command is executed multiple times per render frame, though, in order to enable correct depth testing of scene elements against terrain via multi-frustum rendering.

The shaders used in rendering will be discussed in the next section. First, let's discuss the tile load pipeline.

3.4 Life of a tile

Our terrain and imagery datasets are huge — much too big to fit in memory or even on a local disk - so a primary challenge of rendering them is loading and unloading subsets of data at appropriate times. As a web-based application, Cesium downloads terrain and imagery on demand over HTTP. It uses a tile load pipeline to manage the process of fully populating a selected tile with data, keeping much of the process asynchronous with rendering so that the frame rate remains relatively consistent even as tiles are downloaded and processed.

A tile starts its life as a *skeleton*. Tile skeletons know their location in the world, and their relationship to other tiles, but do not yet have the geometry and texture data necessary to render. During tile selection, we can't refine to a skeleton. Instead, we add the skeleton tile to the load queue and render the parent tile for this frame.

The load queue is a priority queue, implemented as a doubly-linked list. Tiles needed in the most recent frame are kept at the front of the queue, because those tiles are likely to be needed in the next frame as well. The tiles in the load queue that were requested in a single frame are further prioritized based on their level in the tile tree such that larger, lower-detail tiles are loaded before smaller, higher-detail ones, to maximize the average detail across the scene.

Once per frame, Cesium processes the tiles in the load queue, moving them through the following steps:

3.4.1 Create imagery skeletons

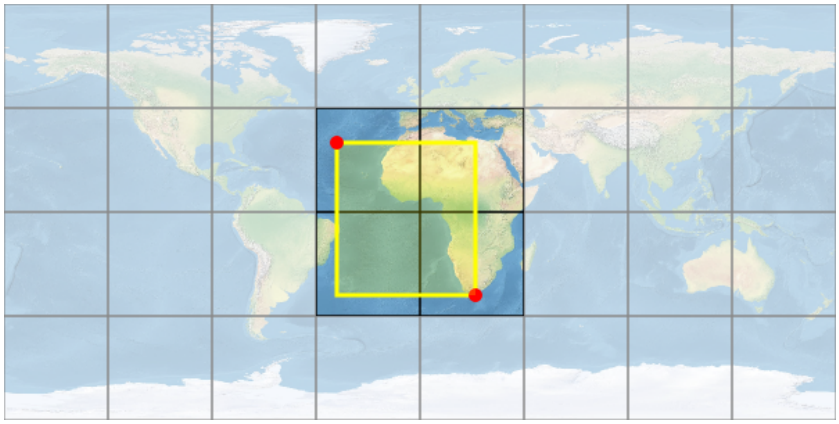
Cesium enables imagery from multiple sources to be overlaid on terrain and layered or alpha blended together. For example, a base map with worldwide imagery can be overlaid with a very high-resolution image for a small area, or a base layer can alpha blended with a rasterized road layer.

In support of this capability, the first task in the load pipeline is to attach *TileImagery* skeletons to the *Tile* for each active imagery layer. Like a *Tile* skeleton, a *TileImagery* skeleton does not yet have any renderable resources. However, it knows which imagery tile will be needed, the minimum and maximum texture coordinates over which the imagery tile applies, and how to scale and translate the imagery tile in order to map it onto the terrain surface.

Given a terrain tile and an imagery layer, Cesium first determines which quadtree level from the imagery layer applies to the terrain tile. As a first cut, we simply need to find the imagery level that has a texel spacing, in world coordinates, that is equal to or smaller than the geometric error of the terrain tile's level. For a good looking scene, however, we have found that it is important that the spacing between imagery texels once projected to the screen be smaller than the terrain screen-space error (SSE). In other words, while a

maximum SSE for terrain geometry of two pixels looks quite good, the scene will look blurry if each imagery texel is allowed to be mapped to two pixels. Cesium selects an imagery level with texel spacing less than or equal to half the geometric error of the terrain tile.

With the level selected, the next step is to identify which imagery tiles in the level overlap the terrain tile. For an imagery layer with a non-worldwide extent, there may not be any tiles that overlap a portion of the terrain tile, so we compute the rectangular extent that is the intersection between the terrain tile's extent and the imagery layer's extent. Then, we use the imagery layer's tiling scheme to find the tile coordinates of the intersection extent's northwest and southeast corners. As shown in the figure below, the result is a rectangular range of tile coordinates, and we create a *TileImagery* object for each tile in the range.



To find the imagery tiles that overlap a terrain tile (yellow), we determine the imagery tiles that contain the northwest and southeast corners (red dots). The containing tiles form the extremes of a rectangular range, and all imagery tiles in the range are rendered on the terrain tile.

Later, in the fragment shader for this terrain tile, we'll determine if the imagery tile applies to the current fragment by checking if its texture coordinates are within the texture coordinate extent for this imagery tile. If so, we'll transform the fragment's texture coordinates to imagery tile texture coordinates by multiplying by the TileImagery's scale property and then adding its translation property. These three properties, the texture coordinate extent, scale, and translation, are set during this step in the load pipeline using a computation on the extents of the terrain and imagery tile, shown in the listing below.

```
var terrainWidth = terrainExtent.east - terrainExtent.west;
var terrainHeight = terrainExtent.north - terrainExtent.south;


var scaleX = terrainWidth / (imageryExtent.east - imageryExtent.west);
var scaleY = terrainHeight / (imageryExtent.north - imageryExtent.south);
var translationX = scaleX * (terrainExtent.west - imageryExtent.west) / terrainWidth;
var translationY = scaleY * (terrainExtent.south - imageryExtent.south) / terrainHeight;
var minU = Math.min(1.0, (imageryExtent.west - tile.extent.west) /
  (tile.extent.east - tile.extent.west));
// maxU, minV, and maxV are similar to minU and not shown
```

Care must be taken to ensure that minU is 0.0 for an imagery tile that starts on the western edge of the terrain tile, and similar for the other edges, even in the face of small rounding errors. Otherwise, cracks will be visible between tiles even when skirts are used. In addition, for adjacent textures, maxU of one must equal minU of the next in order to avoid cracks in the middle of the tile.

3.4.2 Request

The next task is to request the tile terrain and imagery from the remote sources. Because Cesium runs in a web browser, we use browser-based APIs to download images and other resources asynchronously. In a native application, we would use one or more threads to retrieve resources.

We take care to avoid creating too many simultaneous requests. This is especially important in Cesium, which has a browser-imposed limit of six simultaneous requests per hostname. Because the tiles needed to render change as the camera moves, we may decide to start loading a particular tile, then before the request completes, find that it's no longer needed. Throttling requests allows us to re-order pending requests every frame in order to always keep the most important requests at the front, keeping the latency as small as possible.

Cesium benefits from its host web browser's automatic caching of downloaded resources. In a native application, we would manually keep a cache of downloaded terrain and imagery tiles on disk. Such tiles cannot, of course, be prepared for rendering as quickly as those that are already in memory, but they can be readied much more quickly than those that must be requested from a remote server.

When the requests complete, the tile transitions to the received state and is ready for the next step of the load process.

3.4.3 Transform

The transform step takes the raw data received from the remote server and transforms it into a form more amenable to rendering. For example, we turn terrain expressed as a heightmap into a triangle mesh, and add skirts around its perimeter to hide the cracks between tiles that can result when two tiles of different LODs are adjacent to each other.

Transforming a single tile is fast, but when the user moves the camera to a new part of the globe, the time to transform all the new tiles can significantly impact rendering performance. It's much better to continue to render low-detail tiles at interactive frame rates than to stall rendering while transforming a large set of new tiles.

In Cesium, we use a task processing system that uses Web Workers to interpret height maps and compute vertices asynchronously, transferring the resulting array back to the main

thread. Once the main thread receives the results, it transitions the tile to the transformed state and the tile is ready for the next step of the load process.

3.4.4 Create resources

Finally, WebGL resources, such as vertex buffers and textures, are created from the transformed data. For example, after the transform step, we have a TypedArray containing the interleaved vertex attributes for the tile, but that vertex data is not yet in a WebGL vertex buffer and it is not yet available to the GPU. During this step, we upload the vertex data to the GPU as a vertex buffer.

We take care to share WebGL resources whenever possible. A tile of imagery that overlaps multiple terrain tiles will only be loaded and uploaded to the GPU once. For heightmap-based terrain, all meshes across all levels of detail can share a single index buffer. This sharing is managed using reference counting, and yields a significant reduction in memory usage.

Ideally, resource creation would be managed by a thread separate from the rendering thread, and in a native application it probably would be. Doing so reduces memory copies and can enable driver optimizations. On the web, however, it is currently not possible to create WebGL resources in a Web Worker. We look forward to a future extension that allows this.

With its WebGL resources created, the tile is ready to be rendered, and will be added to the render list the next time the tile selection process selects the tile for rendering.

3.4.5 Replacement

As the user moves around the globe, a large number of tiles will be loaded. Once the viewer moves elsewhere, old tiles are no longer needed. To keep the memory used by tiles from growing unbounded, Cesium places all partially or fully loaded tiles in a replacement queue. Each time a tile is traversed or rendered it is moved to the head of the queue, so the tiles at the tail of the queue are the ones that have been used least recently. To save memory, tiles are unloaded from the tail of the queue.

When we unload a tile, we put its WebGL resources, such as textures, back into a pool rather than destroying them outright. Later, when loading a new imagery tile, we can reuse an existing texture from the pool. This approach minimizes the number of calls to allocate WebGL resources.

3.5 Terrain and imagery shaders

Our vertex shader is straightforward. We multiply an RTC model-view-projection matrix by the input vertex position, expressed relative to a center-point somewhere in the tile. By using RTC rendering, we avoid jittering artifacts when zoomed in close to a tile.

The fragment shader is a bit more complicated, mostly due to the need to blend together imagery from multiple layers. We start by assuming the fragment is a constant color, perhaps a nice shade of Earthy blue. Then, in layer order, from bottom to top, each texture has the opportunity to modify that color. The color resulting from one texture is passed as the input to the next. The color output by the last texture is the input to the lighting equations. A single layer may have multiple textures. In that case, their relative order is arbitrary because multiple textures from the same layer do not overlap.

Because we allow different imagery layers to use different tiling schemes — from the terrain and from each other - it is possible and common that a given imagery texture will not completely overlap a tile. For each texture, we pass as packed uniforms the scaleX, scaleY, translationX, translationY, minU, minV, maxU, and maxV values computed during the "Create imagery skeletons" phase of the tile load pipeline. Collectively, these values control which portion of the current texture applies to which portion of the tile.

Tile texture coordinates are $(0.0, 0.0)$ in the southwest corner of the tile and $(1.0, 1.0)$ in the northeast corner. They are linear with longitude and latitude, which is consistent with a geographic projection. The minU, minV, maxU, and maxV values specify the rectangular portion of the tile that is overlapped by the imagery texture. If the current fragment's texture coordinates lie outside this range, this texture does not modify the color of the fragment.

```
if (tileTextureCoordinates.s < minU ||
    tileTextureCoordinates.s > maxU ||
    tileTextureCoordinates.t < minV ||
    tileTextureCoordinates.t > maxV)
{
    return previousColor;
}
```

Next, we compute the texture coordinates with which to sample this texture by applying the scale and translation computed during loading of the tile:

```
vec2 textureCoordinates = tileTextureCoordinates * scale +
translation;
```

After sampling the texture using these coordinates, we apply any adjustments to the sampled color, such as brightness, contrast, hue, saturation, gamma, or alpha. Finally, we alpha blend the sampled color with the previous color and return the new color:

```
return mix(previousColor, color.rgb, color.a);
```

Once the final color has been determined by applying all of the layers, we do one more texture lookup to determine if the fragment is on land or in water. If it's in water, we add a specular highlight and an animated wave effect, following the general approach used by

Jonas Wagner[3].

3.6 Horizon culling

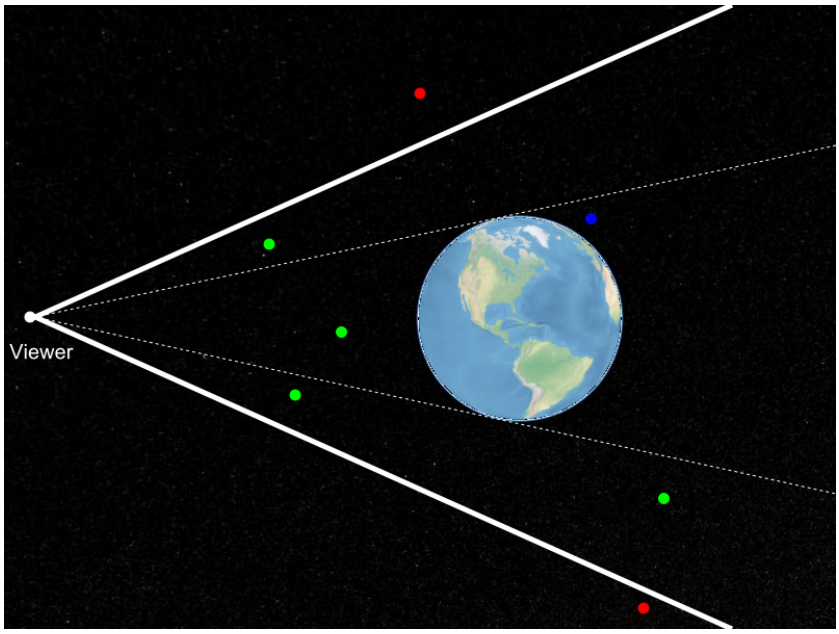
Horizon culling is a form of occlusion culling that is critical for any spherical or ellipsoidal massive world. It is critical because it allows us to determine that significant chunks of terrain are not visible to the viewer, and therefore to avoid rendering them. In the figure below, terrain tiles covering the entire Earth lie inside the view frustum. Over half of them, however, are below the horizon and do not need to be rendered.

The green points are visible to the viewer. The red points are not visible because they are outside the view frustum, which is represented as heavy white lines. The blue point is inside the view frustum, but it is not visible to viewer because it is occluded by the Earth. Horizon culling aims to separate the green points from the blue ones.

Cesium performs horizon culling of each terrain tile using a novel technique that requires only a handful of floating-point operations to test against an ellipsoidal model of the horizon. We achieve this by working in an ellipsoid-scaled space in which the ellipsoid is, very conveniently, represented as a unit sphere. To transform from a standard reference frame with its origin at the center of the ellipsoid and its axes aligned with the ellipsoid's axes to the ellipsoid-scaled space, we simply multiply each coordinate value by the inverse of the ellipsoid's radius along that axis.

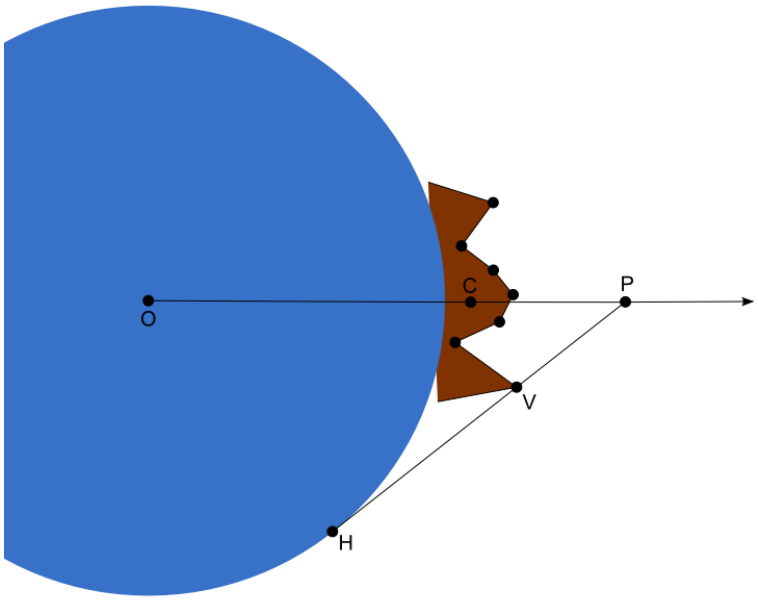
The basis for our horizon culling is a horizon occlusion point associated with each tile.

[3] http://29a.ch/2012/7/19/webgl-terrain-rendering-water-fog



The horizon occlusion point has the useful property that when the point is below the horizon, the entire tile is below the horizon as well. Thus, we can conservatively horizon cull the tile by simply testing the point against the horizon.

We compute the horizon occlusion point during the preprocessing step. First, we arbitrarily assume that the horizon occlusion point lies along a vector from the center of the ellipsoid to the center of the tile's bounding sphere. Then, for each vertex in the tile, we determine where on that center line the point is located.



In the figure above, the Earth ellipsoid is shown in blue and a terrain tile is shown in brown. For a given vertex V, the horizon occlusion point P we're looking for is the intersection of with the center line, . Point H is a point on the horizon from the perspective of point V. There are an infinite number of horizon points from the perspective of point V, forming a circle on a sphere, but only two of these horizon points form a vector through point V that intersects with the center line. One of these intersections will occur after point V and the other before, but we're only concerned with the intersection that occurs afterward, because that one will occur farther from the center of the ellipsoid.

A full derivation of the horizon occlusion point computation can be found on our blog[4]. It all boils down to computing the magnitude of point P along as follows:

```
var position = ...;
var scaledSpacePosition = ellipsoid.transformPositionToScaledSpace(position);
var magnitudeSquared = scaledSpacePosition.magnitudeSquared();
var magnitude = Math.sqrt(magnitudeSquared);
var direction = scaledSpacePosition.divideByScalar(magnitude);


var cosAlpha = direction.dot(scaledSpaceDirectionToPoint);
var sinAlpha = direction.cross(scaledSpaceDirectionToPoint).magnitude();
var cosBeta = 1.0 / magnitude;
var sinBeta = Math.sqrt(magnitudeSquared - 1.0) * cosBeta;


var magnitude = 1.0 / (cosAlpha * cosBeta - sinAlpha * sinBeta);
```

We repeat this computation for each vertex in the tile. The final magnitude is the greatest of the magnitudes computed for any vertex.

Later, during tile selection, we perform the following computation each time the camera moves:

```
var cv = ellipsoid.transformPositionToScaledSpace(cameraPosition, this._cameraPositionInScaledSpace);
var vhMagnitudeSquared = Cartesian3.magnitudeSquared(cv) - 1.0;


Cartesian3.clone(cameraPosition, this._cameraPosition);
this._cameraPositionInScaledSpace = cv;
this._distanceToLimbInScaledSpaceSquared = vhMagnitudeSquared;
```

And then the following for each tile:

```
var cv = this._cameraPositionInScaledSpace;
var vhMagnitudeSquared = this._distanceToLimbInScaledSpaceSquared;
var vt = Cartesian3.subtract(occludeeScaledSpacePosition, cv, scratchCartesian);
var vtDotVc = -vt.dot(cv);
var isOccluded = vtDotVc > vhMagnitudeSquared &&
                 vtDotVc * vtDotVc / vt.magnitudeSquared() > vhMagnitudeSquared;
```

If isOccluded is true, we do not render the tile.

[4] http://cesium.agi.com/blog.html

3.7 Map reprojection on the GPU

Cesium currently displays imagery referenced to either a WGS84 Geographic (EPSG:4326) projection or a Web Mercator (EPSG:3857) projection, the two most common map projections seen on the web.

As mentioned previously, terrain vertices include texture coordinates that assume a Geographic projection. In other words, the texture coordinates, regardless of the original map projection of the terrain data, are a function of the latitude and longitude of the vertex as a fraction of the total latitude and longitude spanned by the terrain tile. The GPU's interpolation of the vertex texture coordinates across fragments gives us reasonable, if not 100% accurate, Geographic texture coordinates for the fragments.

If the source imagery tile is in a Web Mercator projection, however, using these Geographic texture coordinates will badly distort the image for tiles that cover a large spatial area because the texels in such an image have a non-linear mapping to latitude. Instead, we reproject Web Mercator imagery tiles to Geographic on the GPU by rendering them to a framebuffer with a color attachment and doing the reprojection in the fragment shader. We found this to be much lighter on memory than including multiple sets of texture coordinates for the different projections, and more performant than reprojecting the imagery on the CPU.

Transforming Geographic texture coordinates to Web Mercator in order to sample a source Web Mercator texture is straightforward. Given the latitude in radians, the Web Mercator Y-coordinate of the southern edge of the tile, and the latitude of the current fragment, the Web Mercator vertical texture coordinate is:

```
mercatorV = 0.5 * log((1.0 + sin(latitude)) / (1.0 - sin(latitude)) - southMercatorY;
```

The difficulty, as is often the case when rendering a world as big as the Earth, is in dealing with the GPU's limited floating-point precision. In particular, the subtraction of southMercatorY is problematic because both the subtrahend and the minuend are of similar magnitude, so a great deal of precision is lost in the process.

We deal with this problem in two ways. First, we perform the subtraction using simulated double-precision using the DSFUN90 algorithm. Second, for spatially small tiles, we abandon the reprojection and simply use the Web Mercator image as if it were already Geographic. This is acceptable because, over small distances, the transformation amounts to much less than a texel of difference in the texture coordinates. This is shown in the figure below.

While this approach works extremely well on desktop and laptop systems, Cesium is also intended to run on mobile devices using the increasingly high-quality WebGL implementations available in Mozilla Firefox and Google Chrome for Android. Most of these devices have reduced precision available to their fragment shaders, so cutting off the reprojection at level 11 is not soon enough. The result is smeared-looking textures at medium zoom levels.

Our solution is to move the texture coordinate computation to the vertex shader instead

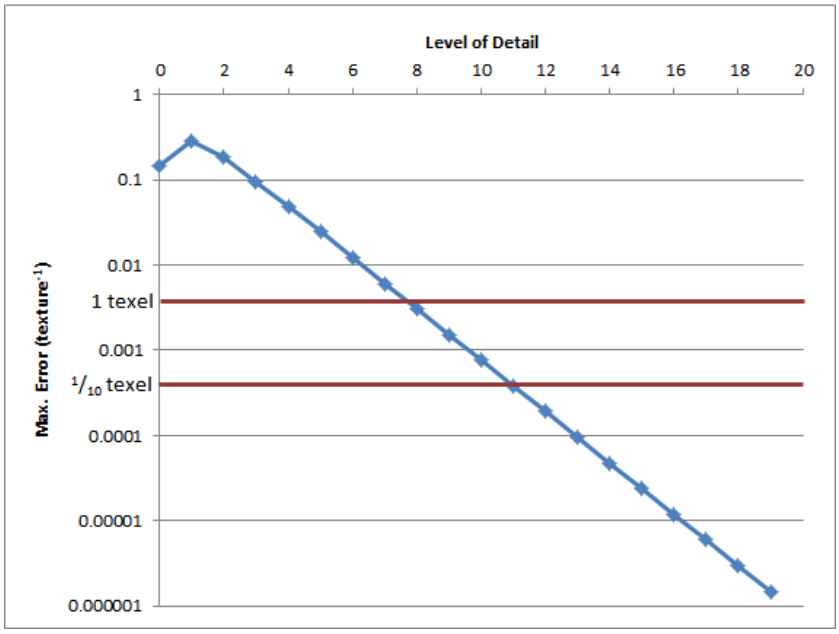


Figure 3.1: If we skip reprojection and assume a Mercator image is the same as a Geographic image, we effectively introduce error in the texture coordinates. This figure shows the maximum magnitude of that error at any latitude for the first 20 levels of the tile quadtree. Beginning at level 8, the error is smaller than a texel of a 256x256 texture. At level 11, the error is smaller than 1/10 of a texel, which we consider acceptable.

of doing it in the fragment shader. This requires many more vertices — approximately one per fragment rather than four total — in order to avoid errors introduced by interpolation of the texture coordinates during rasterization. Fortunately, the vertex buffer is shared and unmodified for all reprojections. In addition, our fragment shader becomes trivial with this modified approach, so overall the performance is very similar to the original approach on desktop systems and has the nice benefit of producing correct results on mobile devices.

3.8 Acknowledgements

Thanks to Scott Hunter for much of the content of the "Life of a Tile" section. Thanks to Frank Stoner for deriving the scaled-space formulation of the horizon culling technique, and for patiently explaining it to me.

Chapter 4

Populating a Massive Game World

4.1 Introduction

In March 2010 Just Cause 2 was released. The game was generally very well received, praised for its vast open landscapes, the player freedom, and the seemingly never-ending amount of things to blow up. To this day Just Cause 2 still stands out among large games.

But filling a game world of over 1,000 km^2 with meaningful content was challenging. The original Just Cause had a game world of the same size, but it had generally been perceived as rather empty, so we had to find better ways to produce content. While we ultimately reached our goals and shipped a high-quality game, getting there was not all a success story. It is also a story about poor tools, broken builds, horrible turnaround times, and the efforts to salvage the situation.

4.2 Build system

4.2.1 Issues with the content build system

Our content build system had grown organically since the birth of the company. No one had a complete picture of everything it did, and it was simply a collection of loose compilers. The dependencies were unclear. Many compilers took results from other compiled data as input. Since already compiled data from previous compiles typically was around this usually worked, but could fail on clean builds, or even content was upgraded. Sometimes there was even dependencies on data from another platform. For instance, it was found at one point that doing a clean PS3 build required not only that the PS3 platform was built repeatedly something like 3 to 5 times for all changes to propagated properly to all dependencies, but it also required a complete build of Win32 data.

As a result of the poor to non-existing dependency management, our content builds were frequently broken. Sometimes the compile seemingly succeeded, but data was broken. Sometimes the build failed despite seemingly valid content. On top of this, the data building process was slow, in the order of many hours. Consequently the turn-around time for data builds tended to be overnight. If a check-in broke something, we would not know until the next day when the nightly build was broken. And sometimes fixing the fault only resulted in revealing the next broken piece of content. Occasionally a company-wide content check-in stop had to be enforced to bring the data back to a working condition.

4.2.2 Untangling the mess

In parallel with the completion of Just Cause 2 the engine team took a holistic approach on fixing the content build system. A framework was built for compilers and a dependency walker that could figure out exactly what needed to be rebuilt and in which order. All compilers were analyzed for whether they were still needed or could be removed, and what their true dependencies were. Any sort of dependency loops were eliminated. With this in place we could do proper incremental builds, which had the greatest impact on turn-around times. But a lot of effort was also spent on removing the worst bottlenecks in the compilers to further reduce build times. Many compilers were rewritten from scratch.

The perhaps greatest challenge was to restore trust in the pipeline. If people are doing complete rebuilds because they don't trust an incremental build to do the right thing, then that's a great loss for productivity. A build error should indicate broken data, and if something fails at runtime after a successful build, it should be assumed the code is to blame, not the data. To ensure that the incremental build was working properly we set up build servers to regularly do complete rebuilds and compare the results to the latest incremental build. Whenever there was a mismatch the problem was promptly investigated and a proper solution devised.

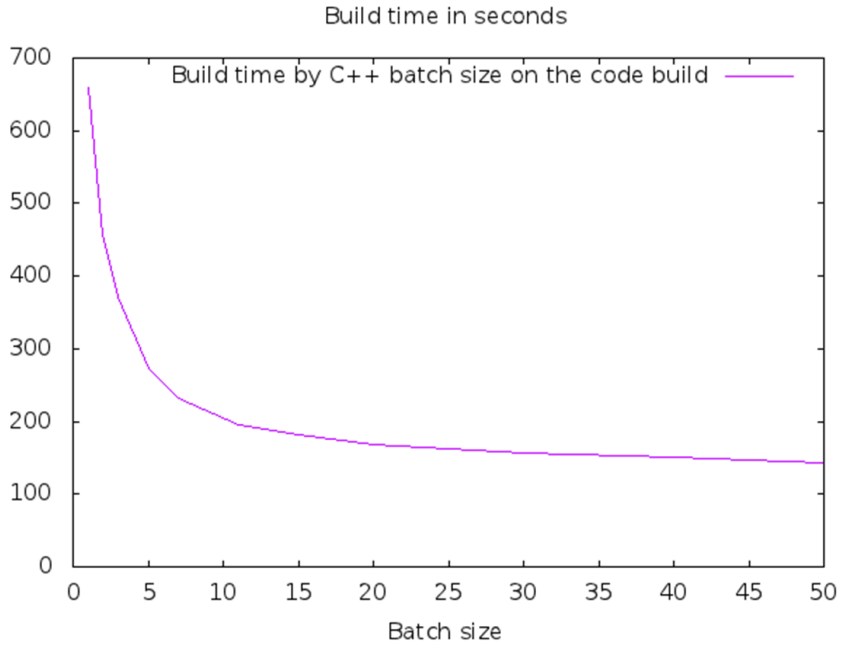
During the Just Cause 2 project missing data was replaced with placeholder content at runtime. Missing models became pink boxes and missing textures were replaced with a pink texture. This way the build could still run with broken data and hopefully be able to spot broken data in game. This may seem like a sensible approach, but we have since abandoned this idea. Instead we have worked hard to ensure that we avoid the problem to begin with, by having the build system fail the build and have the content creator fix the problem before it is ever checked in. The core philosophy is that if the content build succeeds, the resulting data should be complete and working.

4.2.3 Code build system

A big problem is most large productions, whether it is a large game or any other sort of big code-base, is that compiling and linking the executable can take a considerable amount

of time. Like at most game studios, our game is written primarily in C/C++, which is the best choice for runtime performance, but somewhat problematic for build times. In the early days we used third party tools like Incredibuild, which alleviated the problem somewhat by distributing the build cost across all available machines in the studio, but it could still take half an hour to do a clean rebuild of the game. This was of course a big problem for productivity.

Our solution is not necessarily novel, but has worked out great for us. We built a custom build system that batched up *.cpp files, by default in chunks of 30 files. Basically it creates a new set of source files, containing nothing but #includes of the original source files. This batching vastly reduces the amount of duplicated compiles, which is the fundamental flaw of the C/C++ model. The number of translation units is drastically reduced, which also speeds up the linking process, which often tends to be a significant portion of the build time. Figure **??** shows the impact of batching on build performance. The build time was cut down to a fraction of the original.



As an added bonus, with more source baked into the same translation unit we get smaller executables and faster code . The result is similar to what you get from "Whole Program Optimization", enabling for instance cross-module inlining, except for us it is not across the whole program, but only across a batch. But since batches are parsed out from the source directory tree, related files are typically batched together and we get much of the same benefits.

On the downside, batching cpp files comes with a set of problems, primarily that code "leaks" across units. This means that if only one file in a batch includes a header, it will be visible to all source files. Similarly, statements like "using namespace Graphics;" in a

global scope will also leak. The result of this is that code that is broken sometimes actually compiles, and only once a new file is added to the project and the batch arrangement shifts does the problem potentially emerge. To deal with this problem, our auto-build servers also regularly does unbatched builds to catch such errors. Typically the solution is to just include the right header. In rare cases the batching has created hard to understand compile error due to leaking, but overall the time lost to such debugging is vastly outweighed by the gain in far faster build times.

Other things we have done over the last few years is to separate central systems into shared libraries. Originally the project was essentially a large collection of source code, and the engine was not really separated from the game. While there is more to be done in this area, the current situation is much improved, with many core system separated into their own modules, allowing easy sharing of functionality across projects and also speeding up compilation by pulling those out of the project code base. Add hardware upgrades on top of that, including SSD drives in all developer machines, we can now do a complete rebuild of a current project in about 2.5 minutes.

4.3 Content Production

4.3.1 Modularization

Filling a world of this size with content required us to work in a modular fashion. Content was created to be used repeatedly across the world. This means creating reusable chunks that can be placed in the world and easily combined without too much customizations. Other than collections of models it also included light sources, occlusion culling boxes and similar data. The content was arranged such that the entity could be updated and all instances inherited the new settings, unless they had been locally overridden. This allowed updated to art to propagate to the whole world, while maintaining the possibility to do local customizations.

Artists are typically more passionate about art than performance, so placement of occluder boxes for our occlusion culling system was often relatively poor or non-existant at first. However, this setup allowed occluder boxes to be placed into entities and thus automatically placed in all locations that used it.

Some content is generated completely procedurally. For instance light poles and other props around roads.

4.3.2 Landscape authoring

The landscape is divided into axis-aligned patches of a fixed size of 512m $\times$ 512m. We refer to these as stream patches as they are stored and streamed like this at runtime. A

stream patch is a data container that holds not just the terrain vertex data and textures, the landscape physics representation, as well as vegetation and various meta-data.

Each patch is stored separately on disk and in source control, which allows multiple artists to work on the landscape separately without collisions, as long as they work in different areas of the world. Our terrain is for the most part artist authored, using our in-house editor, but we do support importing and exporting subsets of the data for editing in external tools or crafting an initial terrain procedurally.

At runtime the terrain comes in two different representations, one fixed-size height-map for the physics, and a graphical representation that has varying density depending on topographical complexity, proximity to water, and other factors. The graphics representation also has an elaborate LOD system. Editing the terrain is done at a fixed resolution, using our custom editor, and a terrain compiler then computes the final in-game terrain. While the editor uses the full game engine to render the terrain, the intermediate results in the editor might differ somewhat compared to the in-game results after compilation and optimization. For this reason, and practicality in general, we have tools for anchoring entities in the game to the terrain. This means that you can place items on a newly edited terrain and even if the compiler triangulates the area in a somewhat different fashion, the results will still be good. In particular, there will not be any items hanging in thin air (which was otherwise common in the early days of Just Cause 2), or worse, buried underground.

The vegetation is a combination of procedural and artist driven. Most of the vegetation is procedurally placed. Normally it is not particularly important exactly where trees are in a forest, so the system does the vast amount of vegetation placement automatically. You care about the overall density and composition of trees. Artists set up properties that define the vegetation in an area, and the system takes factors such as the slope, elevation and climate zone (desert, jungle, artic etc.) into account and places trees and plants in a randomized but deterministic fashion. We only store a minimum amount of data to reproduce the placement and most placement is done with a seeded random function at runtime. Naturally we need some level of artist control over where vegetation is placed. For instance, we do not want trees to land inside of buildings, or in the middle of a road. Artists can set the density of the vegetation in an area and naturally also completely remove vegetation where it is undesired. The system also automatically removes vegetation under roads.

Sometimes artists need fine control over vegetation. For instance, you may need a tree at a particular spot within a location. For this we have a manual vegetation placement system, allowing the artists to place trees and plants much like any other game object.

For larger vegetation, such as trees and plants, there are four different systems for rendering. Close to the camera trees are drawn as full-blown models. These render like any other models and have several LOD steps. Beyond some distance the models are replaced by simple impostors. At this distance an entire patch, with hundreds of trees, is drawn in a single draw-call. Further out the impostors are rendered as a textured mesh layered on top of the terrain. We sometimes refer to this as the "forest carpet". Finally, to get a proper

silhouette at distant mountains we also have something we call “forest fins”. These

4.4 Bringing it to life

An important aspect of making a large game world is to make it come to life. Filling it with content is very important, but it is not everything. There should also be things going on at idle times, or in- between locations or outside of missions, and you need to preserve a sense of a consistent and contiguous world even though you cannot keep everything in memory, draw all objects or simulate every entity in the world. For this we have a number of systems in place, both for the visual richness and for simulating an alive world.

4.4.1 World simulation

We use a world simulation system that always makes sure that something is going on in the world. The system is driven by the context in which the player is. If you are somewhere along a populated area, say in a village, there will be civilians walking around. Roads have a moderate flow of assorted vehicles passing by. Once in a while an airplane will fly over. In the water you will see boats. There is an important gameplay aspect of this too. It gives you an steady supply of vehicles that you can use. This is especially important if you end up crashing somewhere in the ocean. Swimming back to the shore is probably not going to be very exciting for the player. So we specifically guide the system to have a boat of some kind passing by relatively close soon thereafter that the player can hijack and get back into action.

There is all sorts of animal life as well. There is always a handful of birds around, flies, fish in the sea, and even scorpions. Except for the scorpions maybe, these are extremely simple models. Birds are only a handful of polygons actually and costs nearly nothing to render. But they have a large impact on bringing life to the world.

Another important element is the rolling day-night cycle. An interesting part of how this works in Just Cause 2 is that time is not linear. The time cycle has been tweaked with various considerations in mind, both visual and gameplay. It turns out most players enjoy daytime more than nighttime. This is besides the fact that visibility is better during daytime, even though we spent a lot of effort trying to highlight enemies at night through rim-lighting and other tweaks without breaking the illusion of the scene being dark or enemies popping out unnaturally from the environment. Nighttime is also visually less pleasing in general, except perhaps in the city (which looks great when it lights up at night). So we let the player enjoy the daytime for a longer time than they have to endure nighttime. But we also highlight visually pleasing moments like sunset and sunrise by staying longer there, but once sun is below the horizon, we fast-forward to full night.

4.4.2 Landmarks

We maintain a very large draw distance of 50,000m, which is actually enough to see any part of the world from anywhere, ignoring occlusion of course. Standing on the top of the highest mountain, or flying an airplane, you can see the entire world. Naturally can we only keep the closest locations streamed in at any point. However, it is important both for gameplay and visual richness that we are able to identify key locations even at a very large distance. If you are standing on the top of a mountain, you should be able to see all the interesting places around you, even if they are miles and miles away. This encourages the player to go explore those areas and makes it less necessary to interrupt core gameplay by opening up the map. It is more enjoyable for the player to have a visual target to aim for. For this we created a landmark system.

Models are streamed in and out as needed. They come in different LODs for performance. The landmark system takes over once the regular models in a location gets unloaded. This replaces the entire location with a simple model capturing the essence of the location. Since they are to be viewed from a rather large distance, the models are very low-res in terms of polygon count and texture resolution. The landmark models are loaded at all times and are (ignoring occlusion) visible at all times. Refer to Figures 4.1 and 4.2 for an example of the iconic Mile High Club location visible over 25km.

4.4.3 Lighting

For lighting we had two different solutions for scale. The main system was for relatively close distances where we needed proper lighting. In Just Cause 2 this was handled with a world-space axis aligned tiled light indexing system. The details of this technique has been previously covered in GPU Pro [1]. For our current lighting solution, which is based on Clustered Deferred Shading, please refer to the Advances in Real-Time Rendering in Games course [2]. In both the old and the new system we work on two different scales. The full lighting is done up to a certain distance. In Just Cause 2 the limit was 300m. At the time of this writing, we are using 500m. As lights approach this distance they begin to fade out and smoothly transitions to a distant light system taking over the rest of the range.

The distant light system simulates lighting from static light sources. No actual lighting is computed per se, it basically just highlights the light sources. At data compilation stage all the static light sources in the world are assembled and split up into a grid structure. The grid is used for frustum culling. Each grid cell is drawn with a single draw call. A very compact vertex buffer encodes all lights with only a position, radius and color. The lights are then rendered as simple point sprites. This highlights important locations, especially the city and villages, and makes them look fully lit from a distance. This is not only very cheap to render, but also very effective to bring a huge world to life. In Figure 4.3, note how all the interesting locations are clearly visible at night, giving the player a clue where to go next, in addition to a great visual effect. In Figure 4.4, note how the city stands out and looks alive.

Figure 4.1: Landmark visibility

Figure 4.2: Landmark distance

Figure 4.3: Distant lights highlighting the interesting locations

Figure 4.4: City highlighted by the distant light system

4.5 Summary

While in no way exhaustive, this article touches on some of the issues and some of the solutions developed at Avalanche Studios for creating a vast game world.

4.6 References

[1] Persson, E., 2010. Making it large, beautiful, fast and consistent — Lessons learned developing Just Cause 2. GPU Pro.

[2] Persson, E., 2013. Practical Clustered Shading. Advances in Real-Time Rendering in Games. http://advances.realtimerendering.com/ (to appear)

Chapter 5

Hardware Virtual Texturing

Partially Resident Textures provide direct hardware support for the majority of all tasks present in the virtual texturing pipeline. Application developers are no longer required to deal with managing of the page table, address translation and/or figuring out which texture types need to be supported. The responsibility of managing the virtual nature of a texture is moved toward the hardware and the driver.

Partially Resident Textures are supported in all AMD Radeon HD 7xxx GPUs. The functionality is exposed to application developers via the AMD_sparse_texture OpenGL extension.

PRT support in hardware relies on 3 core components:

- HW Virtual Memory subsystem
- Page Residency information propagation
- Driver stack support for efficient mapping/unmapping

5.1 Virtual Memory

Memory addresses used to fetch texture data on Radeon 7xxx GPUs are virtual. When a shader attempts to fetch a texel from a texture using UV coordinates, a dedicated GPU block first computes the virtual memory address of the texel (or a block of texels if filtering is used). The address computation depends on the texture type, format, UV coordinate values, desired mipmap level, offset, internal texture tiling, etc. The virtual memory address is then fed to the virtual memory subsystem. The VM subsystem performs the virtual-to-physical address translation and then initiates a read operation from the physical memory. When the read completes, the texel data is returned to the shader.

The virtual-to-physical address translation inside the VM subsystem leverages a dedicated hardware page table. This is in stark contrast to software virtual texturing techniques in which the page table is just another texture managed by the application. A dedicated hardware page table provides several benefits when compared to a software one.

Unified format When dealing with software (texture) page tables, the application must decide on its size, format, etc. None of this is required with hardware page tables as they have a unified format and support all texture types, formats and sizes. The advantage of this is that applications can use different formats for different purposes, without having to rewrite their address encoding schemes.

The only downside of the unified format is that texture tiles might have different dimensions based on what type/format they are using. In the current hardware, the page size is fixed to 64kB, which means that a 32-bit RGBA8 texture will have tile dimensions of 128 $\times$ 128 texels. PRT tile dimensions for uncompressed 2D textures are listed in Table 6.1.

Texture BPP	PRT Tile Width	PRT Tile Height
128	64	64
64	128	64
32	128	128
16	256	128
8	256	256

Figure 5.1: PRT tile dimensions for uncompressed 2D textures.

Filtering Hardware page tables provide support for all texture filtering modes. With software page tables, certain filtering modes (e.g. trilinear or anisotropic filtering) are very difficult to implement in a robust fashion. As discussed in the previous chapter, this is because the physical texture coordinates are not contiguous across page boundaries. The hardware is not aware of any page boundaries and therefore cannot filter across pages. On the other hand, PRT-enabled hardware supports filtering across page boundaries without issues.

Two-level structure Software page tables used in virtual texturing are traditionally only one-level (in the virtual memory terminology, they only contain the PTEs). This impacts their sizes and does not allow for any kind of compression. Hardware page tables can be two-level (they contain both PDEs and PTEs), which decreases their memory footprint if the virtual address space is only sparsely populated.

Address translation performance Sampling from a virtual texture with software page tables amounts to a texture fetch using virtual texture coordinates followed by another

texture fetch using physical texture coordinates. Both of these require roundtrips between the shader and memory, which can incur significant performance penalties should cache trashing occur. With hardware page tables, the address translation happens as a part of the texture fetch that uses virtual texture coordinates directly, diminishing the bandwidth requirements by 50% in the general case.

Simplified programming When dealing with software page tables, the application attempting to map/unmap a page needs to take care of the page table updates. Hardware page tables are programmed by the SW driver stack when the client application commits/clears individual texture tiles. The application is only required to specify which parts of the texture should be resident for the upcoming commands.

Caching efficiency Hardware page tables can take advantage of special HW caches to speed up lookups and the virtual-to-physical address translation. This is not possible with software page tables, as they go down the traditional texture fetch hardware path.

Consider the following fragment shader that performs sampling using a software (texture) page table:

```
uniform sampler2D samplerPageTable;          // page table
uniform sampler2D samplerPhysTexture;        // physical texture

in vec4 virtUV;                              // virtual texture coordinates
out vec4 color;                              // output color

vec2 getPhysUV(vec4 pte);                    // translation function

void main()
{
    vec4 pte = texture(samplerPageTable, virtUV.xy);        // 1
    vec2 physUV = getPhysUV(pte);                           // 2
    color = texture(samplerPhysTexture, physUV.xy);         // 3
}
```

Figure 5.2 illustrates what happens inside the hardware during software-based virtual-to-physical address translation operation. In the first texture fetch invocation (line 1), the virtual texture coordinates are used to look up the PTE (page table entry). The PTE is an application-specific data structure stored in the page table texture. In the next step (line 2), the PTE is converted to physical texture coordinates, again by an application-specific function that deals with the encoding scheme, filtering, formats, etc. Finally (line 3), the physical texture coordinates are used to fetch the texture data.

From the hardware's point of view, both texture fetches (lines 1 and 3) are pretty much identical except that they are accessing different textures. One important detail to notice is that both texture fetches are dependent, i.e. the second one cannot be launched before

the first one completes. Dependent texture fetches are generally not a good approach when high-performance is desirable.

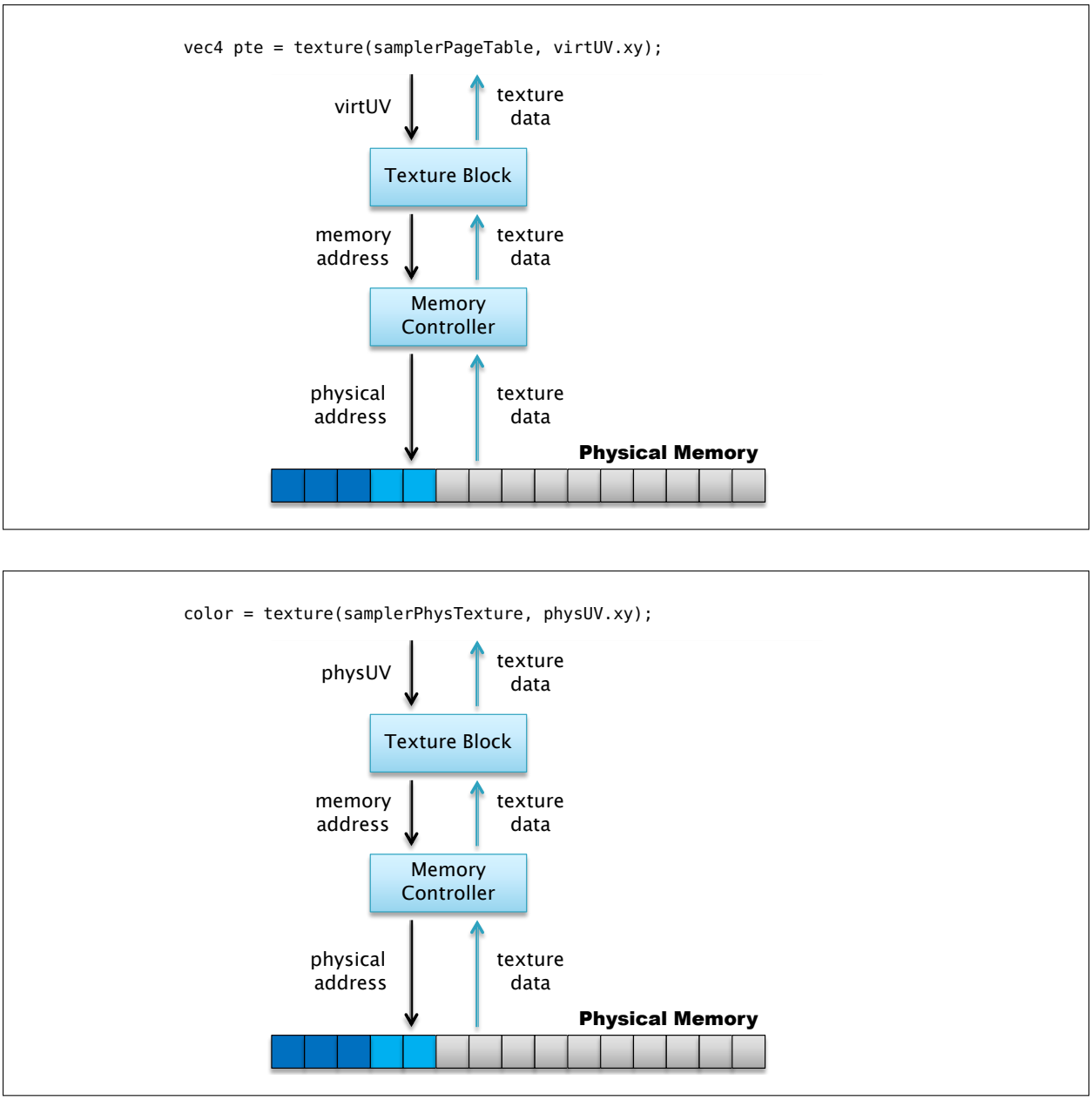


Figure 5.2: Virtual-to-physical address translation using a software (texture) page table and two dependent texture fetches.

Now consider a different fragment shader that takes advantage of a PRT-enabled texture fetch (the sparseTexture() texture sampling instruction is the new instruction introduced in the AMD_sparse_texture OpenGL extension — details in Chapter ??):

```
uniform sampler2D samplerVirtTexture;

in vec4 virtUV;                              // virtual texture coordinates
out vec4 color;                              // output color

void main()
{
    // sparse texture fetch
    int code = sparseTexture(samplerVirtTexture, virtUV.xy, color);
}
```

The shader no longer uses two dependent texture fetches, but instead, the virtual-to-physical address translation is performed directly in hardware based on the virtual texture coordinates passed to the sampling function. The hardware function is depicted in Figure 5.3.

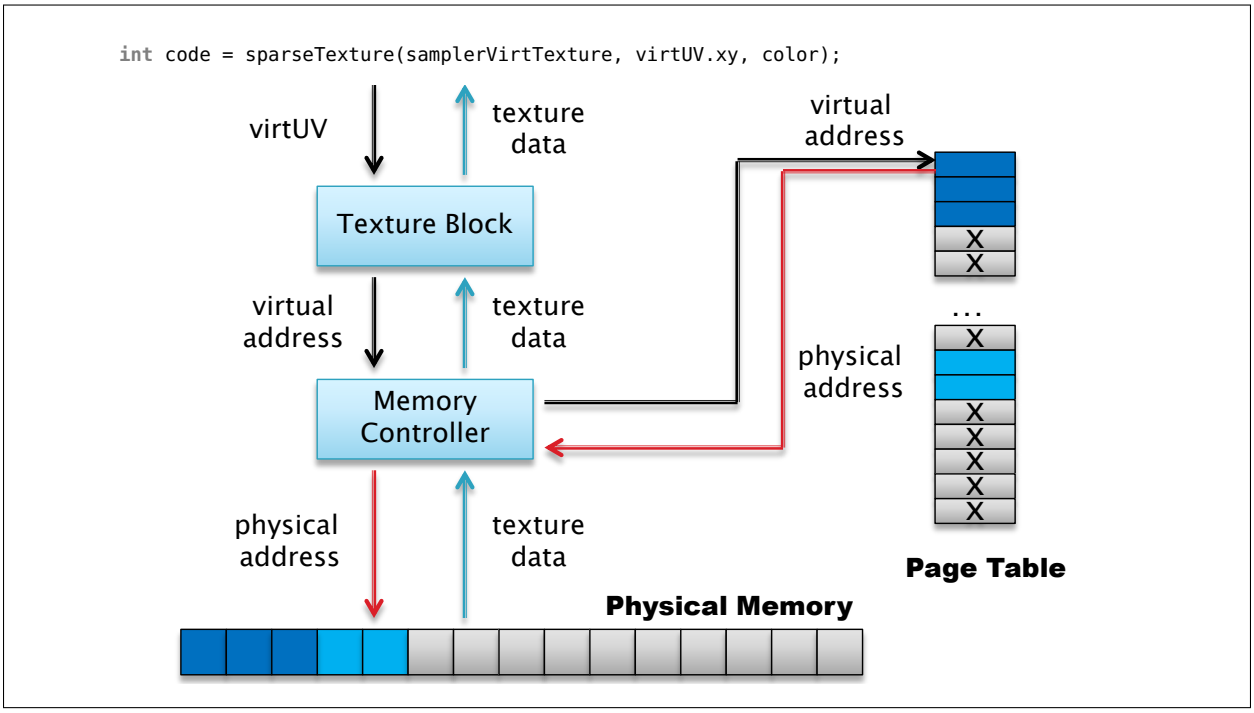


Figure 5.3: Virtual-to-physical address translation using a hardware page table.

5.2 Page/Tile Residency Information

The previous section introduced the concept of hardware page tables and discussed their advantages in the context of fetching data from pages that we know are resident in GPU memory. However, a completely different class of algorithms can be based on the idea of determining page residency information at runtime.

Page fault is a virtual memory event that occurs when the client attempts to translate a virtual address that does not have an entry in the page table (i.e. the page is not mapped to any physical address). In virtual texturing systems, it is very convenient when a shader is able to determine page residency status in an efficient manner.

Querying page residency status from software page tables is straightforward — the shader performs a texture fetch from the page table texture (as in Figure 5.2) and then tests the resulting address for validity (an application specific value can be stored in the page table to indicate unmapped access). The cost of the texture fetch is equal to the cost of any other texture fetch.

With PRTs, the hardware directly supports propagating of the page residency information from the page table to the shader core. In other words, if the shader attempts to read from an unmapped virtual address, the hardware will report failure without having to perform a read from the texture memory. The return code is referred to as a *NACK* in the rest of this document. The sequence of events is illustrated in Figure 5.4.

5.3 Sparse Texture Use Cases & Future Development

In the first part of this chapter, we cover some use cases of PRTs in the real world and demonstrate techniques that may be implemented using the PRT feature. Use cases are enumerated below. In a second part of this chapter, we address some current limitations of the approach implemented in AMD's hardware, and some thoughts on future directions.

5.3.1 Very Large Texture Arrays

First, we discuss the use of very large texture arrays as an application managed cache of textures that may be used to virtually eliminate texture binds in a real-time application. Under this scheme, one, very large texture array is allocated for each class of texture (say, diffuse albedo, specular coefficients, normal maps, etc.). These array textures are bound and left bound for the lifetime of the application. Each material in a scene is assigned a slice of the array. On current hardware, we are able to support more than 8,000 slices in a single texture array, allowing more than 8,000 unique materials to be represented in a single array.

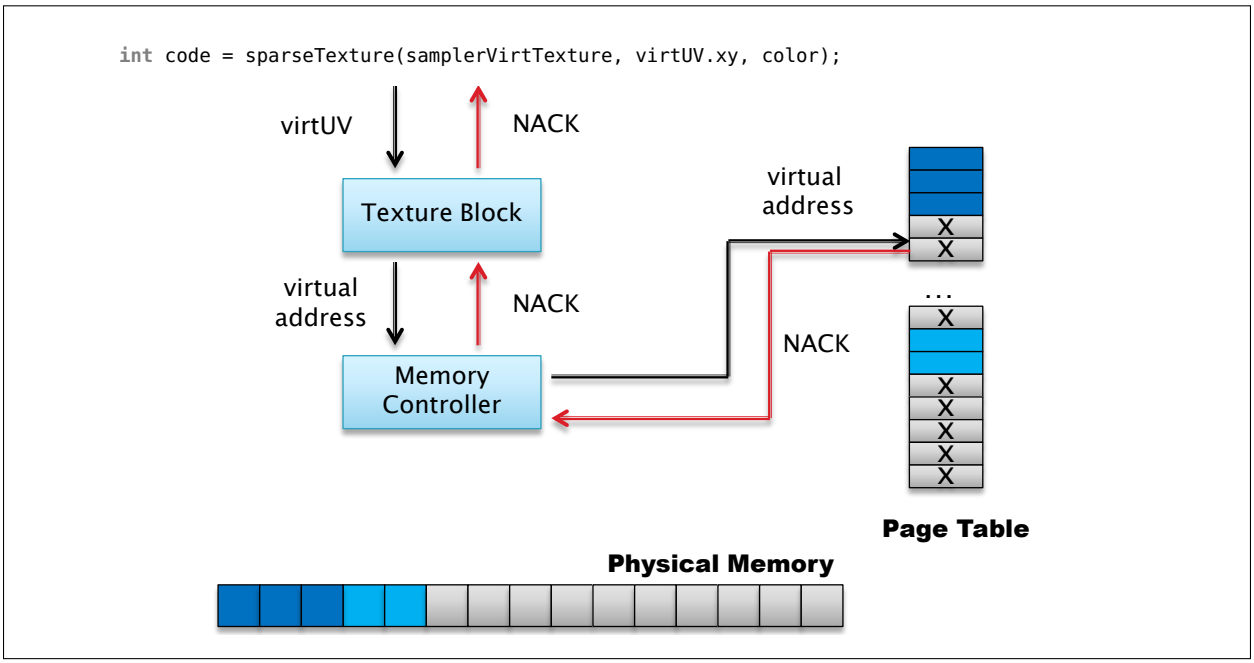


Figure 5.4: NACK propagation.

Of course, a moderate sized array texture (of the order of 2K × 2K texels) with 8,000 slices consumes more than 10s of gigabytes of address space and so it is impossible to ensure that all of the texture data is resident at all times.

However, assuming that the live data set for a single rendering call can be made resident (i.e., it fits in GPU memory), several advantages arise from using texture arrays with sparse textures. The first of these is that the layout of textures in memory is consistent between materials. All materials have access to their diffuse albedo, specular coefficient and normal map textures if they have them. For those materials that do not have some of those components, then those slices of the array may be left non-resident without consuming precious physical memory — only virtual memory is reserved. This simplifies shader development as it allows texture layout to be declared boiler-plate style.

The second, and perhaps more important aspect to this approach is that the texture array can be considered an application-controlled cache. When a material is about to be rendered, the application must ensure that the relevant slices of the appropriate texture arrays are present in GPU memory. However, there is no need to bind new textures as all of the texture data is actually part of the same set of array textures. As materials are rendered, new slices of the array are uploaded to the GPU as needed and then left resident. If the same slice is needed again, then it is already resident and no texture upload or rebind operation is necessary. If, during texture upload, an out-of-memory error is detected, slices that are no longer needed may be discarded and a new attempt to make pages resident made. If, during

the rendering of a single frame, all textures needed fit into GPU memory, then nothing is discarded, everything remains resident and no paging is necessary on the next frame. Thus, the subsequent frame may be rendered with no texture binds at all.

Once the need for binding textures between draw commands is eliminated, several common optimizations found in modern realtime graphics engines become redundant. For example, engines often sort or bin geometry in order to reduce state changes. As a change in texture is no longer considered a state change, this sorting becomes less important. As another example, large, complex models consisting of surfaces with many materials are often broken into several smaller parts for rendering. As all of the texture data for these parts can now be made resident simultaneously, this can be avoided by simply attaching a per-chunk material ID to what would previously have been separate drawing commands. Other graphics features such as instancing become more applicable here.

5.3.2 Incomplete Mip-map Chains

A second technique that becomes possible with sparse textures is the use of incomplete mip-map chains. These may be used for procedurally generated textures or streaming texture data from networks, optical drives or other slow media. Under such circumstances, a minimum level of detail is made resident before scene rendering begins. This level can be chosen by the developer, but due to artifacts of the PRT implementation, is likely to be a minimum of 64KB per texture. This data may, perhaps, be kept closer to the engine in the form of a decompressed base-level texture set, or a set of texture data that is downloaded first. During rendering, a record is made of which textures are actually necessary during scene traversal. This could be done on the CPU based on some simple CPU-based rendering, through GPU assisted techniques such as occlusion queries, or entirely on the GPU by writing texture access data into images in GPU memory. The application then periodically examines the list of live textures and brings them into GPU memory on demand.

On the shader side, an attempt is made to fetch the textures that are required to render the scene. If the necessary textures are not resident in GPU memory, a signal is returned to the shader to indicate so, and the shader begins traversing the mip-map pyramid until a resident texel is found. Because the application made all of the lowest resolution mip-map levels resident during initialization, it is guaranteed that some reasonable texture data is found during this pyramid walk. Over the next few frames, texture data becomes resident – either by loading it from the slow resource, or by generating it on the fly using the CPU or even the GPU itself. Non-resident textures are displayed as blurry, downsampled versions of their higher resolution counterparts at first, and over the course of one or more frames, become sharper. Because no physical address space is required for non-resident texture data, the largest resolution layer of the mip-map pyramid need not even exist if it is known a-priori that it will never be accessed by the texture. The same algorithm follows, though; traverse the mip-map pyramid, starting from the desired LoD until a resident texel is found.

5.3.3 Truly Sparse Textures

Truly sparse textures are another excellent use case for PRT. For example, consider a traditional texture atlas. In general, tools must find a balance between tightly packing atlas components in order to conserve empty space, and leaving enough space between those components to avoid bleeding during the generation of the mip-map chain. With PRT, this is not as necessary. Large regions of unused space may be left empty between components of a texture atlas. Any 64KB chunk of texture can be ignored as it will not be allocated in physical storage. Large, irregular shapes may be created in the atlas without worrying about filling the voids in the convex or even hollow outlines. Sparsity is even more relevant in 3D and volumetric data-sets. A 3D scan of a large volume can often consume many gigabytes of storage, but contain large homogeneous regions and even voids. By using a PRT to store these types of texture, larger volumes that would previously have been impossible to render without complex shader driven page tables may be simply treated as large contiguous textures. Those regions that are completely empty may be left entirely un-allocated. For those regions where lower frequency or even single-valued data is acceptable, the very lowest level of the 3D mip-map pyramid may be used. Use in ray-marching or slice-based rendering algorithms of these apparently complete data sets is then trivial.

5.4 Current Limitations and Thoughts on the Future

The PRT feature we are shipping in hardware is certainly very powerful, but does not address all the wants or needs of the current SVT community. In particular, the maximum texture size has not changed - it is 16K $\times$ 16K $\times$ 8K texels. The limit lies in the precision of the representation of texture coordinates with enough sub-texel resolution for artifact-free linear sampling. To some degree, this may be easy to lift, but we are seeing requests from developers to go as high as 1M $\times$ 1M or more in a single texture. This presents significant architectural challenges and may or may not be feasible in the near term.

It is also easy to see that with large textures and high precision texel formats, we start to exhaust even the virtual address space of the GPU. The largest possible texture is 16K $\times$ 16K $\times$ 8K $\times$ 16 bytes per texel. This amounts to 32 terabytes of linear address space. This far exceeds the addressable space available to the GPU, irrespective or residency. Furthermore, as it is backed by the virtual memory subsystem, page table entries need to be allocated for those pages referenced by sparse textures. The approximate overhead of the page tables for a virtual allocation on current-generation hardware is 0.02% of the virtual allocation size. This does not seem like much and for traditional uses of virtual memory, it is not. However, when we consider ideas such as allocation of a single texture which consumes a terabyte of virtual address space, this overhead is 20GB — much larger than will fit into the GPU's physical memory. To address this, we need to consider approaches such as non-resident page tables and page table compression.

There are several use cases for PRT that seem reasonable but that come with subtle complexities that prevent their clean implementation. One such complexity is in the use of PRTs as renderable surfaces. Currently, we support rendering to PRTs as color surfaces. Writes to un-mapped regions of the surface are simply dropped. However, supporting PRTs as depth or stencil buffers becomes complex. For example, what is the expected behavior of performing depth or stencil testing against a non-resident portion of the depth or stencil buffer? Also, supporting rendering to MSAA surfaces is not well supported. Because of the way compression works for multisampled surfaces, it is possible for a single pixel in a color surface to be both resident and non-resident simultaneously, depending on how many edges cut that pixel. For this reason, we do not expose depth, stencil or MSAA surfaces as renderable on current generation hardware.

The operating system is another component in the virtual memory subsystem which must be considered. Under our current architecture, a single virtual allocation may be backed by multiple physical allocations. Our driver stack is responsible for virtual address space allocations whereas the operating system is responsible for the allocation of physical address space. The driver informs the operating system how much physical memory is available and the operating system creates allocations from these pools. During rendering, the operating system can ask the driver to page physical allocations in and out of the GPU memory. The driver does this using DMA and updates the page tables to keep GPU virtual addresses pointing at the right place. During rendering, the driver tells the operating system which allocations are referenced by the application at any given point in the submission stream and the operating system responds by issuing paging requests to make sure they are resident. When there is a 1-to-1 (or even a many-to-1) correspondence between virtual and physical allocations, this works well. However, when a large texture is slowly made resident over time, the list of physical allocations referenced by a single large virtual allocation can become very long. This presents some performance challenges that real-world use will likely show us in the near term and will need to be addressed.

Chapter 6

High Quality Software and Hardware Virtual Textures

6.1 High Quality Software Virtual Textures

Modern simulations increasingly require the display of very large, uniquely textured worlds at interactive rates. In large outdoor environments and also high detail indoor environments, like those displayed in the computer game RAGE, the unique texture detail requires significant storage and bandwidth. Virtual textures reduce the cost of unique texture data by providing a sparse representation which does not require all of the data to be present for rendering, while leaving the majority of the texture data in highly compressed form on secondary storage.

A virtual texture is divided into small pages that are loaded into a pool of resident physical pages as required for rendering. In RAGE these small pages are square blocks of 128 x 128 texels and the pool with physical pages is a fully resident texture that is logically subdivided into such square blocks of texels. While a virtual texture can be very large (say a million pages) and is never fully resident in video memory, the texture that holds the pool of physical pages is fully resident but much smaller (typically only 4096 x 4096 texels or 1024 pages). Virtual texture pages are mapped to physical texture pages, and during rendering virtual addresses need to be translated to physical ones.

Virtual textures differ from other forms of virtual memory because first, it is possible to fall back to slightly blurrier data without stalling execution, and second, lossy compression of the data is perfectly acceptable for most uses. Implementations of software virtual textures exploit these key differences between virtual textures and other forms of virtual memory to maintain performance and reduce memory requirements at the cost of quality. Implementing virtual textures without special hardware support is challenging and inevitably comes down to finding the right trade between performance, memory requirements, and quality.

While the implementation of software virtual textures in RAGE emphasized performance, the visual fidelity of the virtual textures in RAGE can be improved in several ways that trade performance and memory for quality.

6.1.1 Explicit Page Table LOD

The high performance software virtual textures implemented in RAGE use page table textures to perform the virtual to physical translation. Such a page table texture must be point-sampled to retrieve individual page mappings without mangling the data by blending between adjacent but independent page mappings. Using the texture hardware to point-sample a page table texture does not necessarily result in a mapping to a physical texture page with the appropriate texture detail for the anisotropic texture fetch that follows. Without any adjustments, the page table lookup returns a mapping to a physical texture page that is typically too coarse and provides too little detail. To provide more detail the page table lookup can be biased with the base-two- logarithm of the maximum anisotropic footprint. This results in the page table lookup returning a mapping to a texture page with enough detail for the anisotropic filter to work well on surfaces at an oblique angle to the viewer where the sampled footprint is maximized (anisotropic). However, this can cause noticeable shimmering or aliasing on surfaces that are orthogonal to the view direction where the sampled footprint is minimal (isotropic). To improve the quality, the correct LOD for the page table texture lookup can be calculated in the fragment program based on the anisotropy. The calculated LOD can then be explicitly passed to the page table texture lookup. The calculation of the page table LOD can be found below. Calculating the page table LOD for every fragment adds significant fragment program complexity.

```
const float maxAniso = 4;
const float maxAnisoLog2 = log2( maxAniso );
const float virtPagesWide = 1024;
const float pageWidth = 128;
const float pageBorder = 4;
const float virtTexelsWide = virtPagesWide * ( pageWidth - 2 * pageBorder );

vec2 texcoords = virtCoords.xy * virtTexelsWide;

vec2 dx = dFdx( texcoords );
vec2 dy = dFdy( texcoords );

float px = dot( dx, dx );
float py = dot( dy, dy );

float maxLod = 0.5 * log2( max( px, py ) );  // log2(sqrt()) = 0.5*log2()
float minLod = 0.5 * log2( min( px, py ) );

float anisoLOD = maxLod - min( maxLod - minLod, maxAnisoLog2 );
```

In RAGE texture feedback is rendered to a separate buffer that, for the virtual texture pages used in the current scene, stores the virtual page coordinates (x,y), desired mip level, and virtual texture ID (to allow multiple virtual textures). This feedback data is then used to make those virtual texture pages resident that are needed to render the current scene. Interestingly, the calculation of the desired mip level for texture feedback is equivalent to the calculation of the page table LOD. While in RAGE the feedback was rendered in a separate

pass, the feedback can also be generated during normal rendering by using multiple render targets. This allows the texture LOD to be calculated once after which it can be used for both texture feedback and the page table texture lookup.

6.1.2 Tri-Linear Filtering and LOD Clamping

It will happen sometimes that, despite all efforts, a significant latency will be incurred between the time that a texture page is needed and the time that the data for it is available. This can result in an unpleasant "pop" when the desired LOD for a page is off by more than one level and the right LOD suddenly becomes available.

If tri-linear filtering with two virtual to physical translations is employed, then it is possible to include some delay in the transition from coarser to finer mip level when the desired level is not sufficiently close to the currently displayed level. Tri-linear filtering with two virtual to physical translations is expensive because it requires double the number of page table and physical texture lookups. Tri-linear filtering by blending between two anisotropic texture lookups does, however, provide a noticeable quality improvement.

Gradually blending in finer detail requires a minimum LOD texture with one texel per virtual page. The texels of the minimum LOD texture are gradually adjusted to reveal more and more detail after a new texture page has been made resident. The minimum LOD texture is used to clamp the texture lookup in the fragment program, forcing a gradual transition to finer detail as opposed to a sudden change which is perceived as a “pop”.

```
uniform sampler2D pageTable;          // RGBA-FP32 - { scaleS, scaleT, biasS, biasT }
uniform sampler2D minLodTexture;      // R-8       - { minimum-LOD }
uniform sampler2D physicalTexture;    // RGBA-8    - { red, green, blue, alpha }

in vec4 virtCoords;   // virtual texture coordinates
out vec4 color;       // output color

void main()
{

    const float maxAniso = 4;
    const float maxAnisoLog2 = log2( maxAniso );
    const float virtPagesWide = 1024;
    const float pageWidth = 128;
    const float pageBorder = 4;
    const float virtTexelsWide = virtPagesWide * ( pageWidth - 2 * pageBorder );

    vec2 texcoords = virtCoords.xy * virtTexelsWide;

    vec2 dx = dFdx( texcoords );
    vec2 dy = dFdy( texcoords );

    float px = dot( dx, dx );
    float py = dot( dy, dy );

    float maxLod = 0.5 * log2( max( px, py ) );  // log2(sqrt()) = 0.5*log2()
    float minLod = 0.5 * log2( min( px, py ) );

    float anisoLOD = maxLod - min( maxLod - minLod, maxAnisoLog2 );

    const float maxVirtMipLevels = 16;
    float clampLod = texture( minLodTexture, virtCoords.xy ).x * maxVirtMipLevels;
    anisoLOD = max( anisoLOD, clampLod );
```

```
        vec4 scaleBias1 = textureLod( pageTable, virtCoords.xy, anisoLOD - 0.5 );
        vec4 scaleBias2 = textureLod( pageTable, virtCoords.xy, anisoLOD + 0.5 );

        vec2 physCoords1 = virtCoords.xy * scaleBias1.xy + scaleBias1.zw;
        vec2 physCoords2 = virtCoords.xy * scaleBias2.xy + scaleBias2.zw;

        vec4 color1 = texture( physicalTexture, physCoords1 );
        vec4 color2 = texture( physicalTexture, physCoords2 );

        float trilinearFraction = fract( anisoLOD );

        color = mix( color1, color2, trilinearFraction );

    }
```

The above code shows the complete fragment program that combines the explicit page table LOD calculation from the previous section (in blue), clamping of the page table LOD using a minimum LOD texture (in red), and tri-linear filtering using two page table lookups and two physical texture lookups (in green).

6.1.3 Texture Upsampling

A common approach to increase the perceived detail on textured surfaces is to add detail textures. A detail texture is a small texture with high frequency data that can be easily tiled many times over many surfaces. Detail textures are blended over regular textures to give textured surfaces a more detailed appearance without increasing the texture resolution. Not only are detail textures yet another specialized form of texture compression, detail textures also have several limitations and disadvantages such as: local modulation, limited variety, creation cost, selection cost and run-time cost.

Instead of using manually created and applied detail textures, the texture detail on rendered surfaces can also be programmatically enhanced. Normally the texture hardware uses a bilinear filter for texture magnification during rendering. This filter is implemented in graphics hardware and is very fast. Unfortunately, bilinear filtering tends to produce interpolation artifacts such as blurring and edge halos. More advanced upsampling filters and texture enhancement algorithms can be implemented in a fragment program. However, programmatic upsampling and enhancement of texture data in a fragment program is usually very costly.

Virtual textures make it possible to upsample and enhance existing texture data without increasing the per fragment rendering cost. A virtual texture can be extended to have many more mip levels for which actual source data does not exist. A texture page for which no original source data is available can then be generated from a coarser parent page that does have source data. As opposed to upsampling the texture data for every rendered fragment, the cost is amortized by only upsampling and enhancing texture data once, as the view approaches a surface and the texture pages for that surface are made resident. Various interesting upsampling algorithms can be used to generate additional detail.

6.1.4 Direct Texture Access

On systems with direct access to texture memory, the physical texture pages and page table textures can be updated asynchronously to the GPU. However, on systems where the only access to video memory is through an API like OpenGL or DirectX, the physical texture and page table texture updates cannot generally overlap with rendering and face significant API overhead. For instance, in RAGE more than 6 milliseconds of CPU time may be spent uploading and/or copying texture data through the graphics driver.

For improved memory access patterns and consequently improved performance, textures are usually stored in tiled formats. Tiling of textures improves the spatial locality of the texture data for typical texture sampling patterns during rendering. Texture tiling is one of the reasons direct access to texture memory is not generally supported. By allowing direct texture access, the texture needs to be either stored in a linear (non-tiled) format or the application needs to be aware of the particular tiled format being used. More importantly by allowing direct texture access it is no longer possible to change the tiled format underneath an application. Being able to change the tiling formats for existing applications can be important to achieve performance gains by developing new tiling formats that are designed specifically for the memory architecture of new graphics hardware. However, while these performance gains may be very important for some applications, other applications may not benefit as much. For instance, the virtual textures in RAGE do not appear to benefit as much from different tiled formats. In particular when the texture data is stored in a block compression format such as DXT/S3TC/ETC, the benefits appear to be small because the block compression formats already improve the spatial locality by encoding small blocks of texels. Sampling of page table textures during rendering always exhibits very good locality and storing them in a linear format has good performance. At the same time, not allowing direct texture access and forcing texture updates through the driver causes significant overhead.

Direct access to tiled textures is possible as long as the application is aware of the particular tiled format that is being used. For instance, in RAGE the texture transcoders can be easily modified to write directly to textures in a tiled format as long as the tiled format is clearly specified and does not change after the release of the game. While there may be a noticeable performance delta between linear and tiled textures, there is usually a much smaller performance delta between different tiled formats. In other words, having a couple of standardized tiled texture formats seems particularly useful.

6.2 Hardware Virtual Textures

When RAGE first shipped there was no special hardware support for virtual textures. The virtual to physical address translation had to be implemented in a fragment program through page table and mapping textures. The latest AMD graphics hardware, however, supports

hardware virtual textures also known as Partially Resident Textures (PRTs). Instead of using page table and mapping textures the hardware can perform the virtual to physical translation using the page tables of the underlying virtual memory system. Taking advantage of this special hardware does require some changes to the virtual texture system in RAGE.

In RAGE, multiple virtual textures can be mapped to the same pool with physical pages or a single virtual texture can be mapped to multiple pools with physical pages. When using PRTs, a virtual texture is not mapped to one of the existing physical pages pools. Instead, a new physical pages pool is allocated for each virtual texture. Such a private pool does not use a regular small texture but instead the pool is implemented as a PRT with the same size as the virtual texture. Instead of mapping and unmapping texture pages using physical coordinates, pages are mapped and unmapped using virtual coordinates. The hardware then takes care of the virtual to physical address translation and there is no need for page table textures. This is an elegant solution that requires minimal changes throughout the RAGE virtual texture system. Nevertheless, various changes are necessary to properly support PRTs.

6.2.1 PRT Page Management

In RAGE physical texture pages are allocated from fixed size physical textures. These physical textures tend to fill up quickly and once all pages from a physical texture have been allocated, an existing page will have to be unmapped and freed before a new texture page can be allocated. The page that is unmapped and freed may have been used for a different virtual texture when multiple virtual textures map to the same physical texture.

A hardware virtual texture in the form of a PRT can have as many pages resident as there is physical memory available. When a PRT page is mapped the memory allocation will either succeed or fail based on whether or not physical memory is available. To make sure the virtual texture pages for a particular scene can be made resident, it is desirable to track all the resident texture pages and unmap and free those texture pages that are no longer needed. To allow a PRT page to be mapped without running out of physical memory, a page management system must be implemented that can track and if necessary unmap and free any PRT page from any virtual texture. With such a page management system in place, allocating a physical page for one PRT may cause a page from another PRT to be unmapped and freed.

6.2.2 PRT Size Limitation

While a virtual texture in RAGE can be up to 128k x 128k, the maximum size of a PRT is only 16k x 16k. As a result, a large virtual texture cannot be mapped directly to a PRT. A virtual texture that is larger than 16k x 16k needs to somehow map to multiple PRTs. One solution is to use a Partially Resident Texture Array. This allows a virtual

texture to be logically broken up into 16k x 16k tiles that map to the PRTs stored in an array. In a fragment program, the array index and the texture coordinates within the PRT can be calculated from the texture coordinates that address the full virtual texture through multiplication by the size of the full virtual texture divided by 16k. The integer parts can then be used to calculate the array index and the fractional parts are the texture coordinates within the PRT.

To allow proper texture filtering without seams, this does, however, require that no texture island crosses a 16k boundary on the full virtual texture. These texture islands are chunks of geometry that are contiguous in texture space, often called "charts" in an "atlas" in literature. Texture islands larger than 16k x 16k will also have to be broken up into multiple islands. The algorithm that creates the layout of a virtual texture has to be modified to split up islands that are too large, and to keep texture islands from crossing 16k boundaries on the full virtual texture. Splitting up texture islands larger than 16k x 16k is not always easy without introducing noticeable seams. Fortunately, few texture islands are actually larger than 16k x 16k. Making sure that texture islands never cross a 16k boundary on the virtual texture is much easier and can, for instance, be done by pre-allocating the texels on the 16k boundaries such that those texels cannot be allocated for any of the actual texture islands.

6.2.3 Compressed PRT Pages

PRT texture pages do not have a fixed size in texels. Instead PRT texture pages always have a fixed size in memory which on current AMD graphics hardware is 64 kB. As a result, the size in texels of a texture page varies based on the texture format and compression.

Format	Tile Width	Tile Height
uncompressed RGBA-8	128	128
DXT5/BC3 compressed	256	256
DXT1/BC1 compressed	512	256

Table 6.1: Tile dimensions for compressed 2D textures.

To keep things simple, uncompressed RGBA-8 texture pages can be used for a first pass implementation of PRTs. To support DXT compression, multiple texture pages on disk have to be transcoded and uploaded simultaneously.

6.2.4 Borderless PRT Pages

One of the unfortunate complexities of software virtual textures without special graphics hardware support is that the texture unit, being unaware of the actual texture pages, cannot filter across page boundaries. Texture pages that are adjacent in virtual texture space do

not necessarily map to physical pages that are next to each other, let alone close to each other in the physical texture. In order to properly support hardware bi-linear filtering of software virtual textures, each physical texture page must have a border of texels around it. Hardware accelerated anisotropic filtering of software virtual textures can be supported if the page border is wider than one texel.

With virtual textures supported in hardware in the form of PRTs there is no need for page borders. However, the virtual texture pages in RAGE are stored on disk with page borders. The texture pages in RAGE are 128 x 128 texels with a 120 x 120 payload surrounded by a 4 texel border. One option is to upsample the center 120 x 120 to 128 x 128 when a page is transcoded from highly compressed form on disk to a format the GPU can use directly for rendering. Unfortunately this causes noticeable blurring due to the non-integer upsampling ratio. For the best quality the virtual textures will have to be reformatted to strip the borders and to re- subdivide the virtual texture into texture pages with a full 128 x 128 payload.