Under the Hood of Virtual Globes

Patrick Cozzi
Analytical Graphics, Inc.
University of Pennsylvania

Kevin Ring
Analytical Graphics, Inc.

Software for Space, Defense & Intelligence

COM.Geo 2011

UNIVERSITY of PENNSYLVANIA

Administrivia

- Download course slides:

http://www.virtualglobebook.com/

- Recording is OK
- Ask questions anytime
- Come and go as you please

We are informal

Course Goals

- Enjoy pretty pictures and demos
- To gain an appreciation for and understanding of graphics engines in virtual globes
- Useful for
  - Implementors
  - Integrators

- *This course is not*
  - A direct comparison of virtual globe engines
  - A tutorial on rendering effects like atmospheres and oceans - maybe next year :)

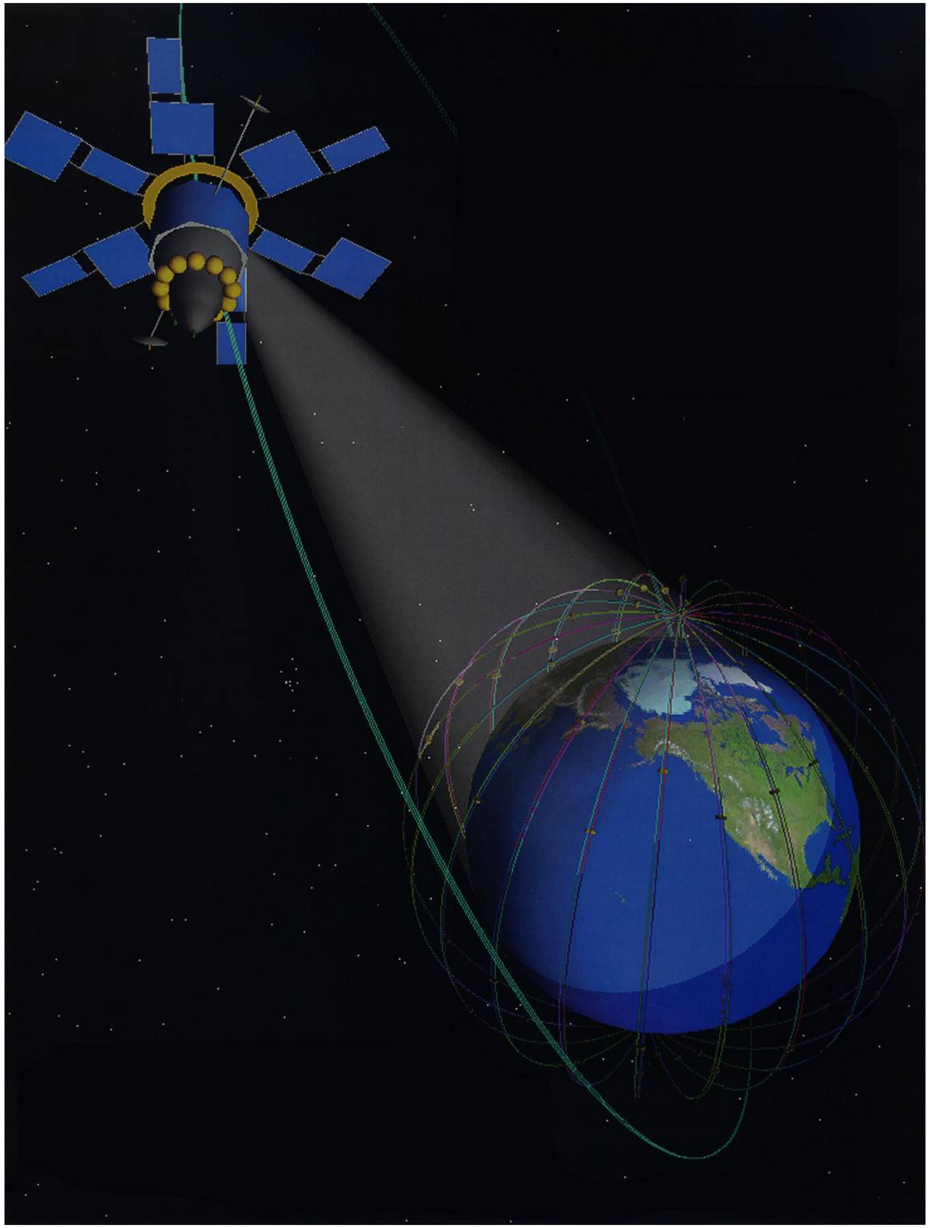
Course Overview

- Our Background
- Ellipsoids
- Precision
- Parallelism
- Terrain

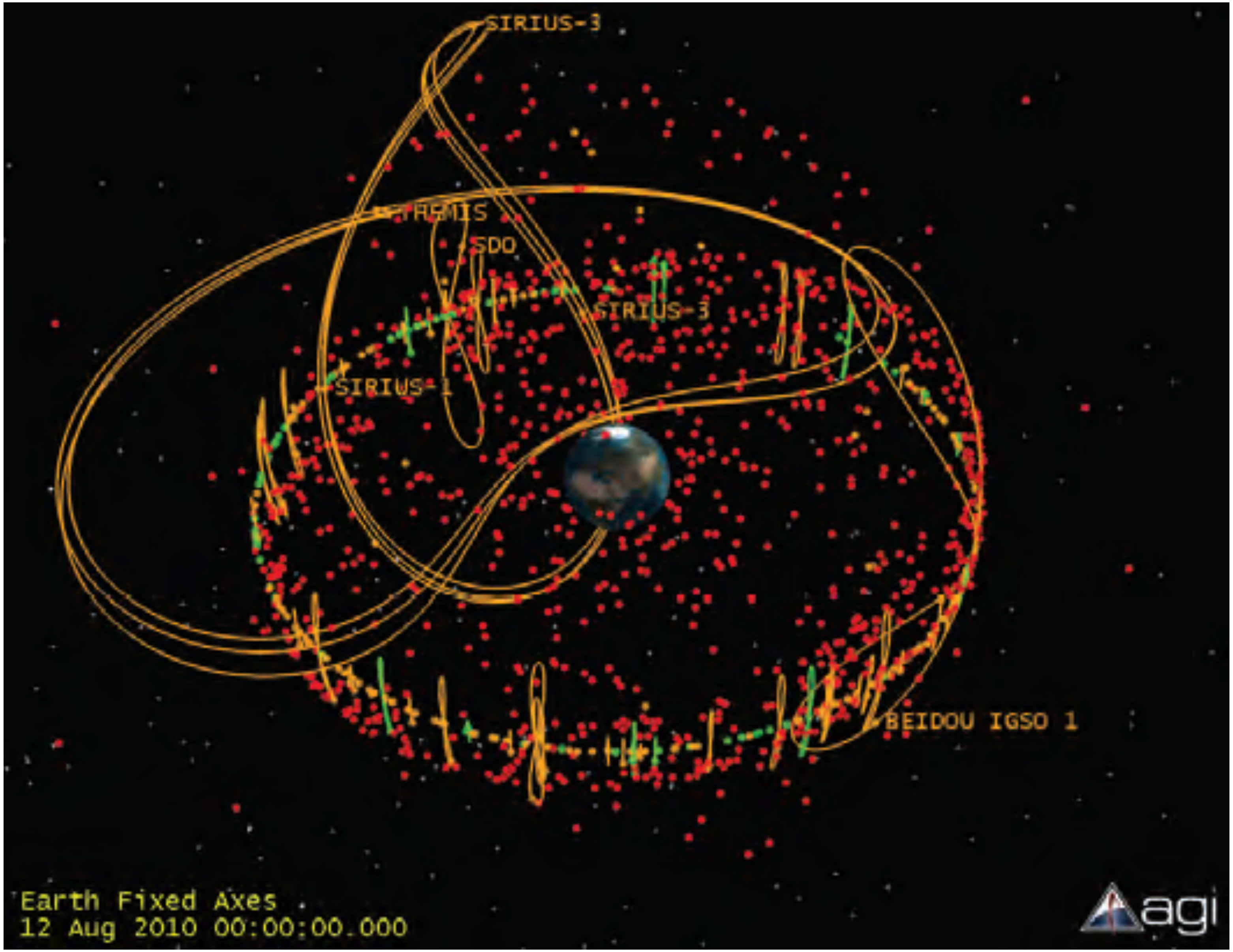
Our Background

STK

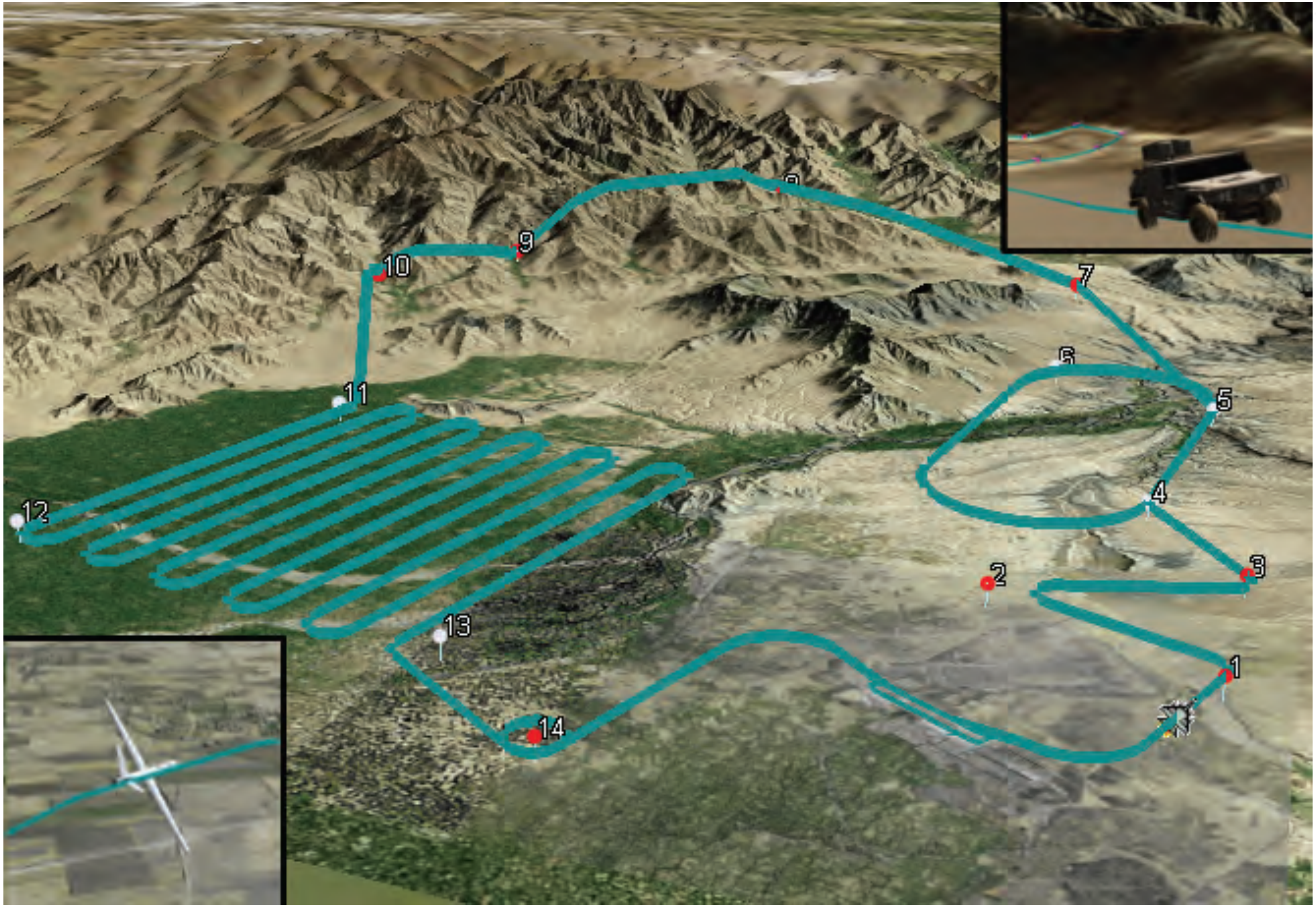
- First had a 3D, spinning globe in 1993
- STK/VO - Visualization Option
- Ran on high-end, SGI IRIX workstations
- Emphasis on space and **analytical accuracy**
- Less emphasis on terrain and imagery (!)



STK Today


SIRIUS-3
SDO
SIRIUS-3
SIRIUS-1
BEIDOU IGSO 1
Earth Fixed Axes
12 Aug 2010 00:00:00.000

STK Today

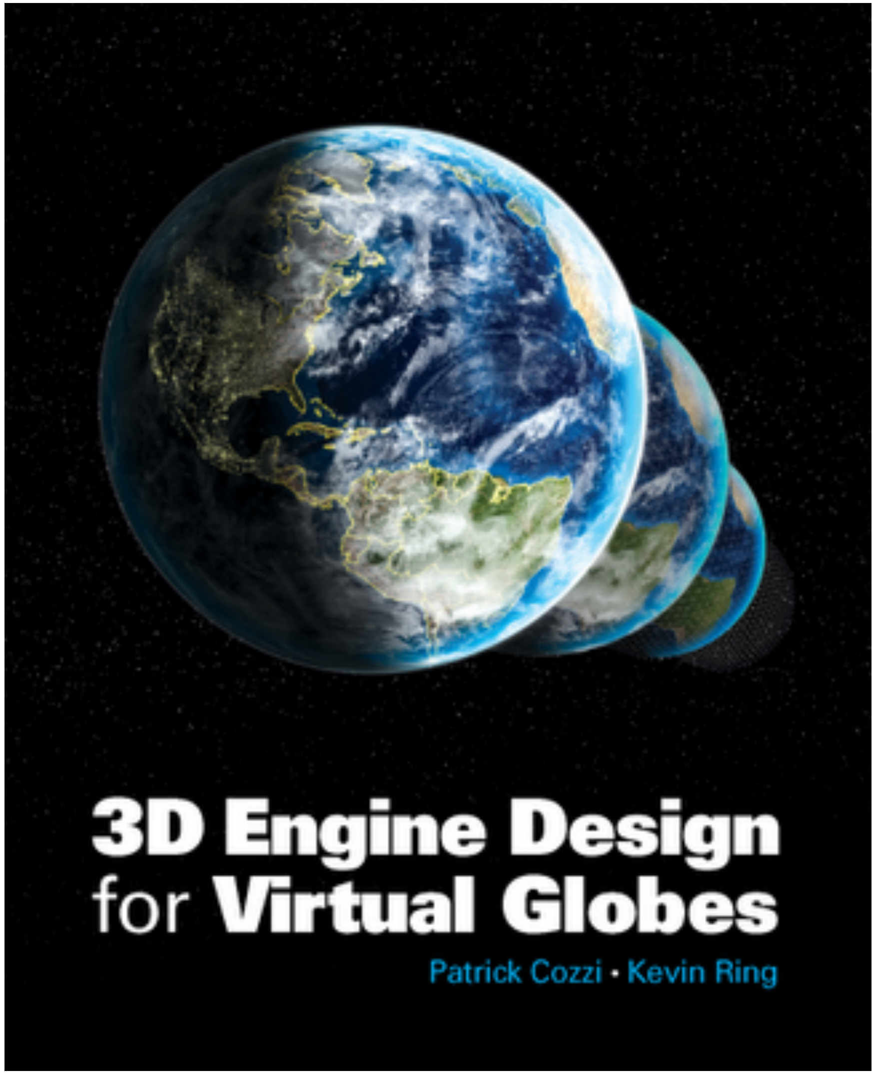


Our Book

This course is based on a subset of our upcoming book:

3D Engine Design for Virtual Globes

http://www.virtualglobebook.com/



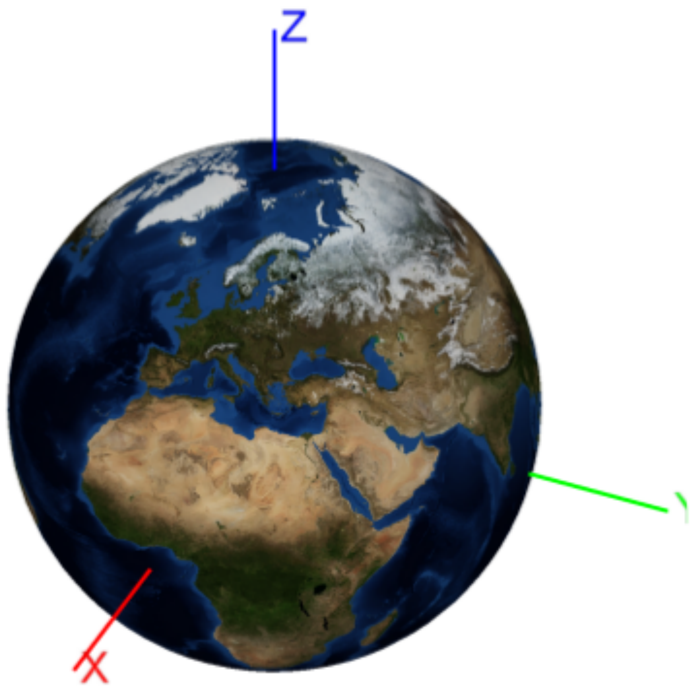
Foundations

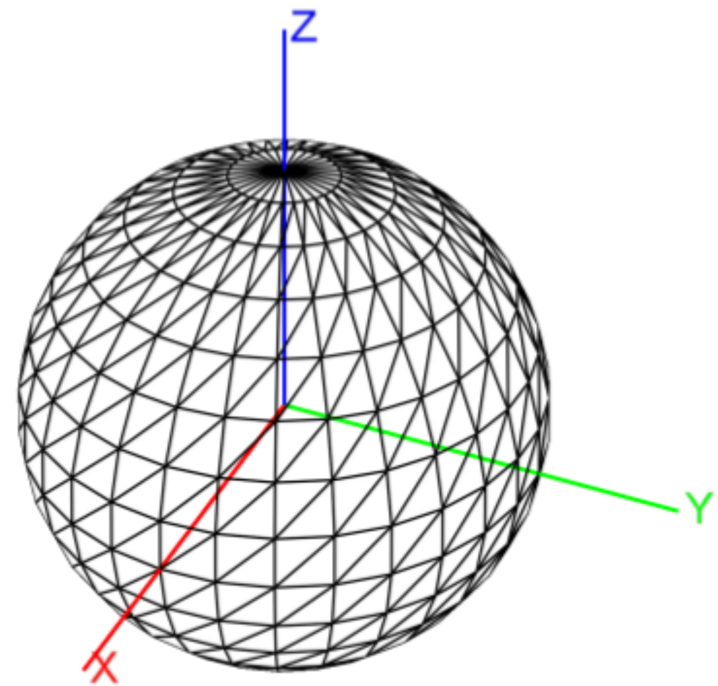
Geographic and Cartesian Coordinates

- Lots of geospatial data uses geographic coordinates (longitude, latitude, height)
  - KML, ESRI Shapefiles, etc.
- The video card wants Cartesian coordinates (x, y, z)
  - (long, lat, height) != (x, y, z)
  - What to do?

Geographic and Cartesian Coordinates

- Pick a Cartesian coordinate system, .e.g.,
    - *WGS84 Coordinate System*
        - *W*orld *G*eodetic *S*ystem 19*84*



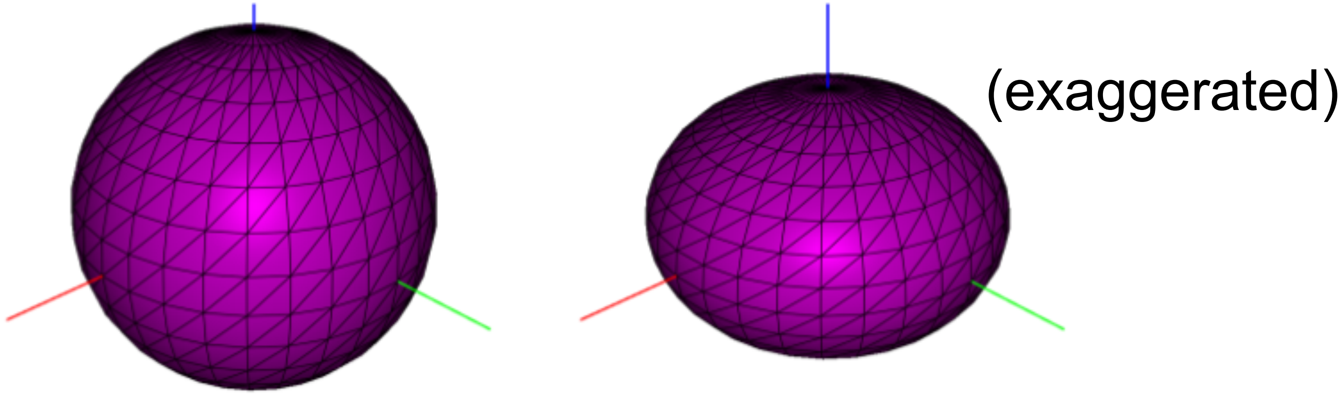


Geographic and Cartesian Coordinates

- Conversions between coordinate systems
  - Geographic → Cartesian
    - Simple and closed form
  - Geographic ← Cartesian
    - Simple and closed form when *height == 0*
    - General case is iterative (our algorithm, at least)
      - Converges quickly for Earth

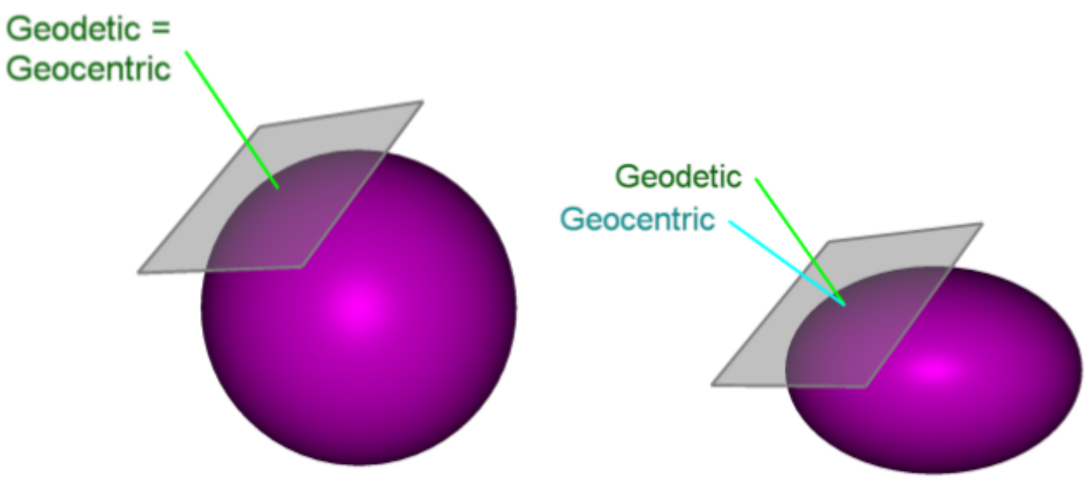
Ellipsoids

- *WGS84 Ellipsoid*
  - National Geospatial-Intelligence Agency's (NGA) latest model of Earth
  - Equatorial radius: 6,378,137 m
  - Polar radius: 6,356,752.3142 m
  - About 21,384 m longer at the equator than at the poles
    - Not too important for imagery on the globe
    - Important when positioning objects above the ground, e.g., aircraft, satellites, etc.

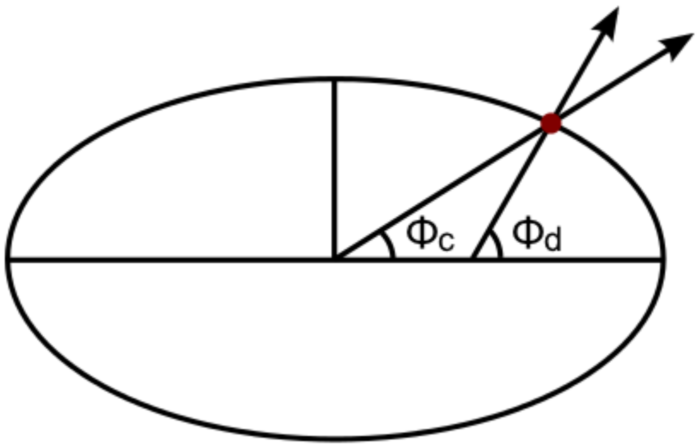


Ellipsoids

- *Geodetic* vs. *Geocentric* surface normals



- *Geodetic* vs. *Geocentric* latitude



Demos

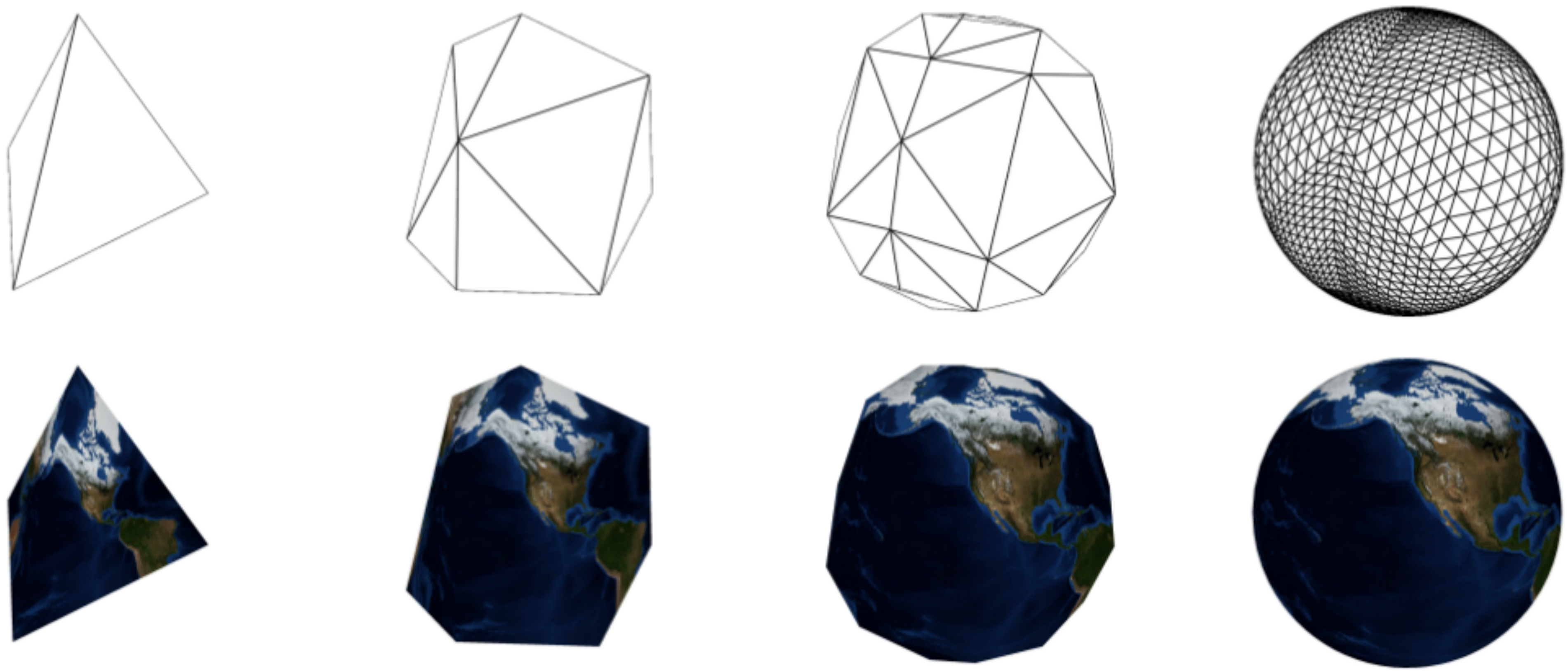
- Geodetic vs. Geocentric normals

Ellipsoid Representations

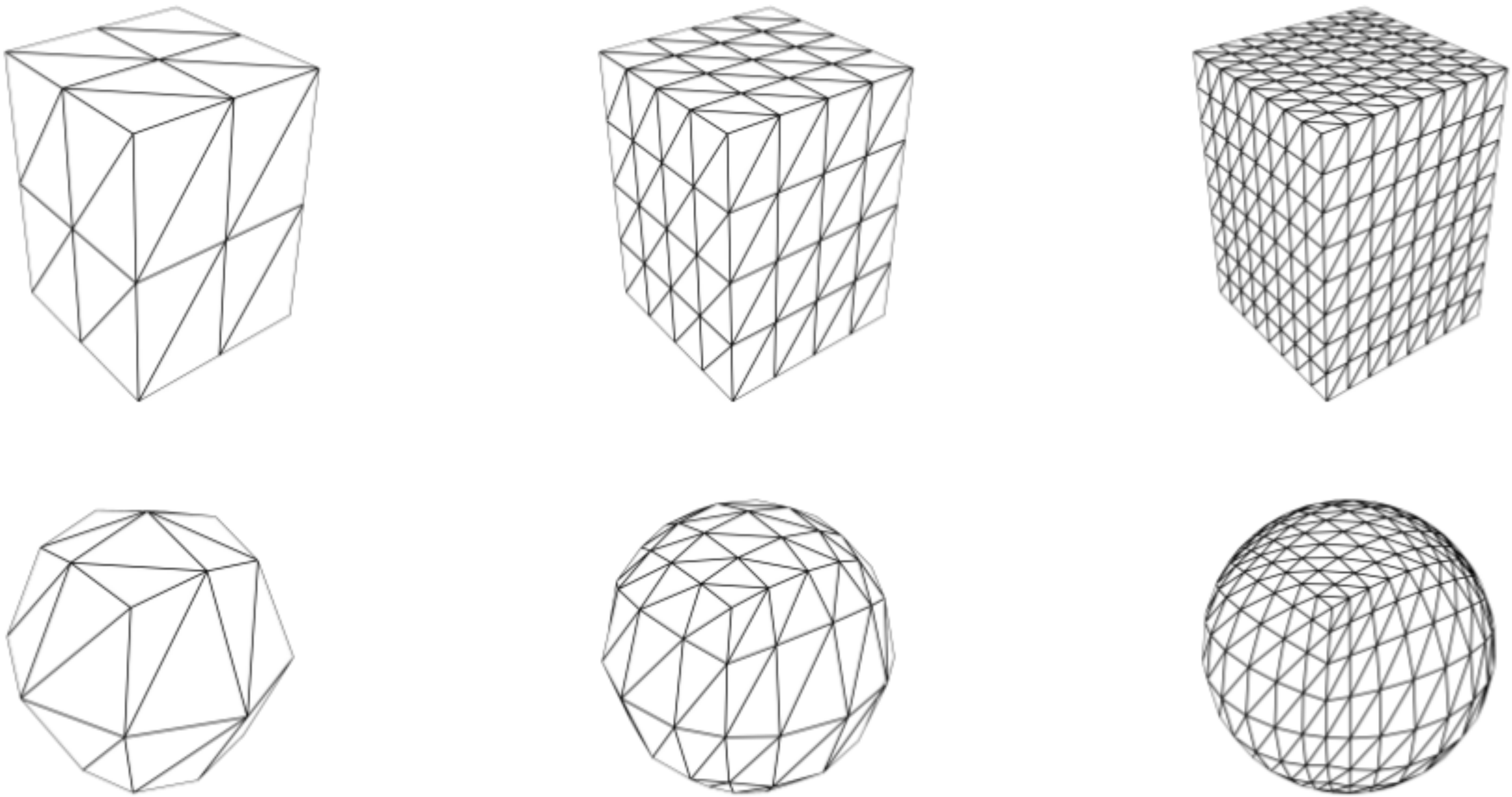
Ellipsoid Representations

- Our ellipsoid is defined by an equatorial radius and polar radius, but the video card wants triangles
- *Solution*: tessellation or ray casting

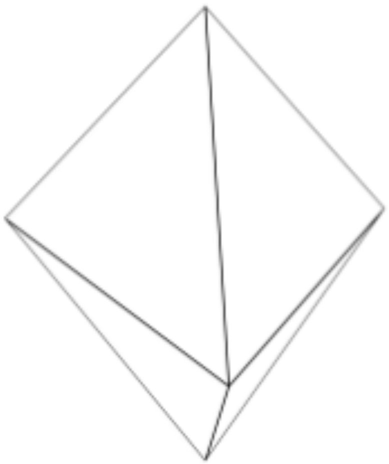
Platonic Solid Subdivision

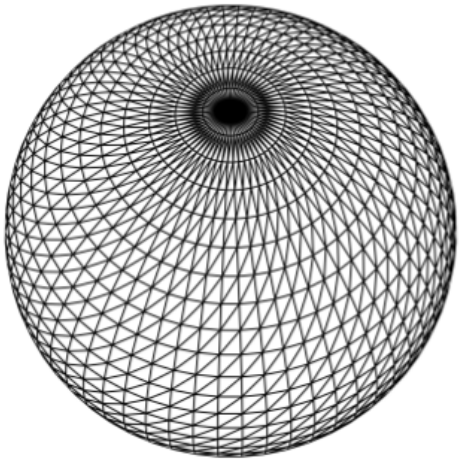


Cube-Map Tessellation



Geographic-Grid Tessellation





$$x = a \cos\theta \sin\phi,$$
$$y = b \sin\theta \sin\phi,$$
$$z = c \cos\phi.$$

Tessellation-Algorithm Comparisons

Algorithm	**Oversampling at Poles**	**Triangles Avoid Poles**	**Triangles Avoid IDL**	**Similar Shape Triangles**	**Aligned with Constant Latitude/Longitude Lines**
Subdivision Surfaces	No	No	No	Yes	No
Cube Map	No	No	Yes	No	No
Geographic Grid	Yes	Yes	Yes	No	Yes

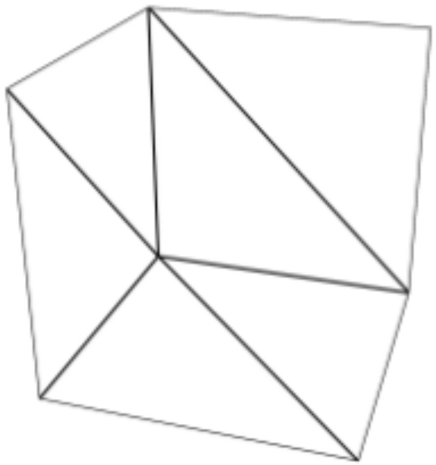
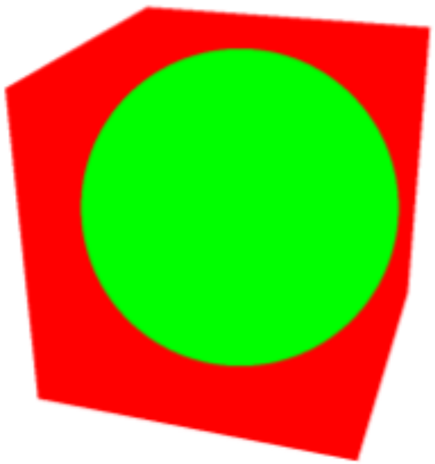
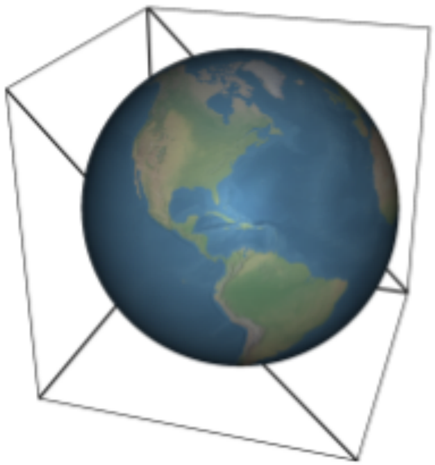
GPU Ray Casting

- Tessellation
  - Each algorithm has strengths and weaknesses
  - Needs *LOD* to balance triangle count vs. visual quality

- *Rasterization*: triangles → pixels
- *Ray tracing*: what triangles, or objects, affect a pixel?

- Ray cast ellipsoid's implicit surface
  - Infinite level of detail
  - No triangles - no problems at poles or IDL
  - Trivial memory requirements

GPU Ray Casting

- *Downside*: GPU 32-bit precision
  - Speaking of precision...

Demos

- GPU Ray Casting

High Precision Rendering

High Precision Rendering

- *Rendering precision*: a difference between virtual globes and most game engines. How do we support:
  - Large WGS84 coordinates with 32-bit GPUs?
    - Vertex transform precision
  - Long view distances with a non-linear depth distribution?
    - *Depth buffer precision*

- *Disclaimer*: Nowadays not all GPUs are 32-bit

Demo

- *Jittering* caused by vertex transform precision
  - http://blogs.agi.com/insight3d/index.php/2008/09/03/precisions-precisions/

Vertex Transform Precision

- CPUs: 64-bit
- Many GPUs: 32-bit

- Cause of jittering: insufficient precision in 32-bit floating-point represents for large values like 6,378,137.
- IEEE-754 rules of thumb
  - 32-bit: 7 accurate decimal digits
  - 64-bit: 16 accurate decimal digits

Vertex Transform Precision

- Gaps between representable floating-point values

```
float f = 6378137.0f;      // 6378137.0
float f1 = 6378137.1f;     // 6378137.0
float f2 = 6378137.2f;     // 6378137.0
float f25 = 6378137.25f;   // 6378137.0
float f26 = 6378137.26f;   // 6378137.5
float f3 = 6378137.3f;     // 6378137.5
float f4 = 6378137.4f;     // 6378137.5
float f5 = 6378137.5f;     // 6378137.5
```

- Gap increases as values get further away from zero

```
float f = 17000000.0f;   // 17000000.0
float f1 = 17000001.0f;  // 17000000.0
float f2 = 17000002.0f;  // 17000002.0
```

Vertex Transform Precision

- Example matrix-vector multiply done in vertex shader:

$$
\begin{aligned}
p_{\text{clip}} &= \mathbf{MVP} * p_{\text{WGS84}} \\
&= \begin{pmatrix} 2.17f & 1.77f & 0.00f & -13,844,653.0f \\ 0.76f & -0.94f & 3.53f & -4,867,969.0f \\ -0.60f & 0.73f & 0.32f & 3,810,548.25f \\ -0.60f & 0.73f & 0.32f & 3,810,548.25f \end{pmatrix} \begin{pmatrix} 6,378,137.0f \\ 0.0f \\ 0.0f \\ 1.0f \end{pmatrix} \\
&= \begin{pmatrix} (2.17f)(6,378,137.0f) + -13,844,653.0f \\ (0.76f)(6,378,137.0f) + -4,867,969.0f \\ (-0.60f)(6,378,137.0f) + 3,810,548.25f \\ (-0.60f)(6,378,137.0f) + 3,810,548.25f \end{pmatrix}.
\end{aligned}
$$

Vertex Transform Precision

- Example matrix-vector multiply done in vertex shader:

Large translation

Large WGS84 position

$$
\begin{aligned}
p_{\mathrm{clip}} &= \mathbf{MVP} * p_{\mathrm{WGS84}} \\
&= \begin{pmatrix} 2.17f & 1.77f & 0.00f & -13,844,653.0f \\ 0.76f & -0.94f & 3.53f & -4,867,969.0f \\ -0.60f & 0.73f & 0.32f & 3,810,548.25f \\ -0.60f & 0.73f & 0.32f & 3,810,548.25f \end{pmatrix} \begin{pmatrix} 6,378,137.0f \\ 0.0f \\ 0.0f \\ 1.0f \end{pmatrix} \\
&= \begin{pmatrix} (2.17f)(6,378,137.0f) + -13,844,653.0f \\ (0.76f)(6,378,137.0f) + -4,867,969.0f \\ (-0.60f)(6,378,137.0f) + 3,810,548.25f \\ (-0.60f)(6,378,137.0f) + 3,810,548.25f \end{pmatrix}.
\end{aligned}
$$

- Jitter at 800 m view distance, but not 100,000 m. *Why?*

Vertex Transform Precision

Solutions

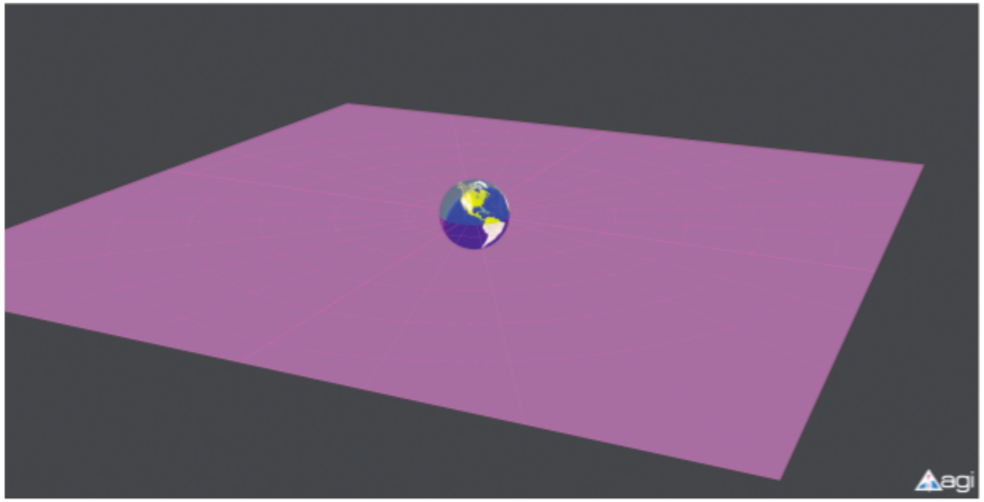
- Scaling coordinates doesn't help. *Why?*
- Use the CPU's double precision or emulate it on the GPU

Render Relative to Center (RTC)

$$\mathbf{MV_{RTC}} = \begin{pmatrix} MV_{00} & MV_{01} & MV_{02} & \text{center}_{\text{eye}}x \\ MV_{10} & MV_{11} & MV_{12} & \text{center}_{\text{eye}}y \\ MV_{20} & MV_{21} & MV_{22} & \text{center}_{\text{eye}}z \\ MV_{30} & MV_{31} & MV_{32} & MV_{33} \end{pmatrix}$$

Render Relative to Center (RTC)

- 1 cm accuracy for radius up to 131,071 m
- So, how do you render this?



Render Relative to Eye (RTE)

$$\mathbf{MV_{RTE}} = \begin{pmatrix} MV_{00} & MV_{01} & MV_{02} & 0 \\ MV_{10} & MV_{11} & MV_{12} & 0 \\ MV_{20} & MV_{21} & MV_{22} & 0 \\ MV_{30} & MV_{31} & MV_{32} & MV_{33} \end{pmatrix}$$

$$p_{\mathrm{RTE}} = p_{\mathrm{WGS84}} - \mathrm{eye}_{\mathrm{WGS84}}$$

- Per-vertex on the CPU or on the GPU with emulated double precision in the vertex shader

See [Ohlarik08]

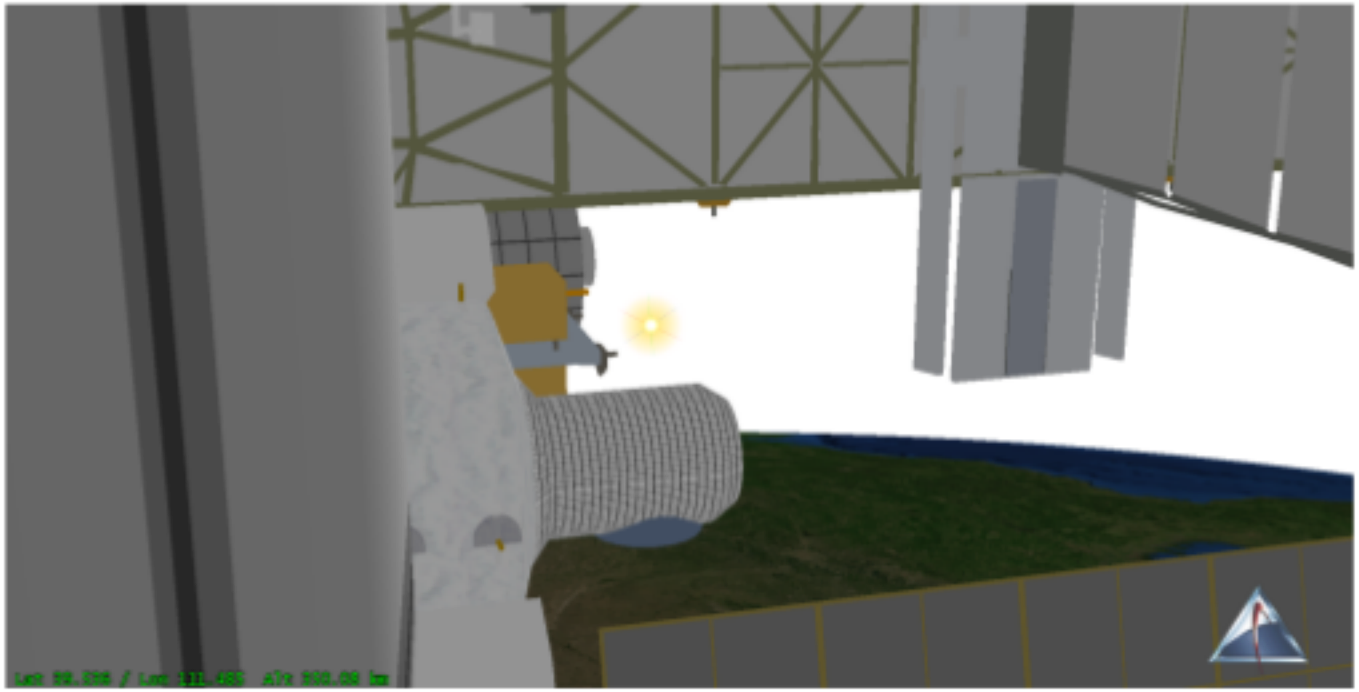
Depth Buffer Precision

- How can we render very close and very far objects in the same scene?

Image courtesy of Brano Kemen, Outerra

Depth Buffer Precision

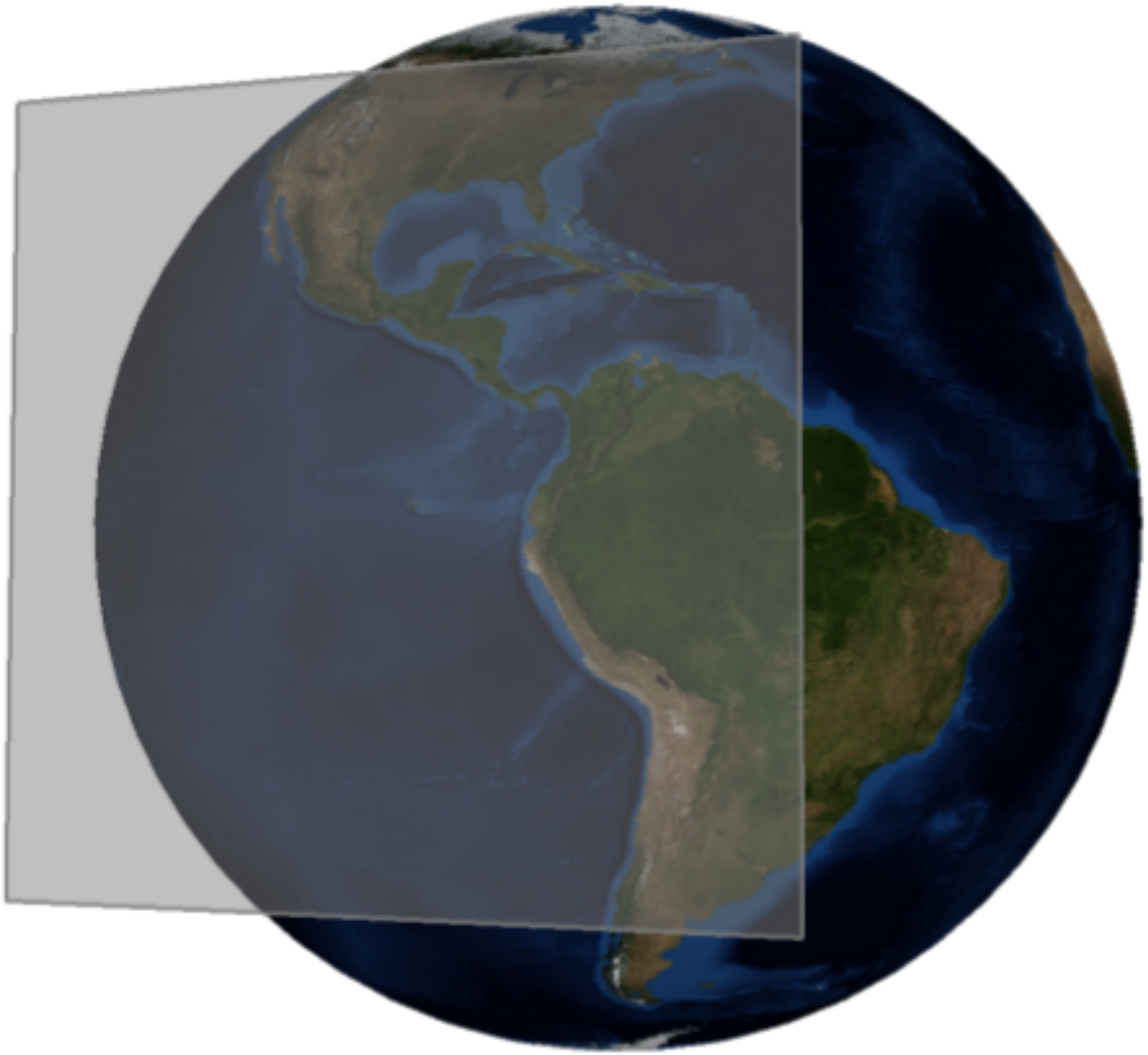
- How can we render very close and very far objects in the same scene?



Depth Buffer Precision

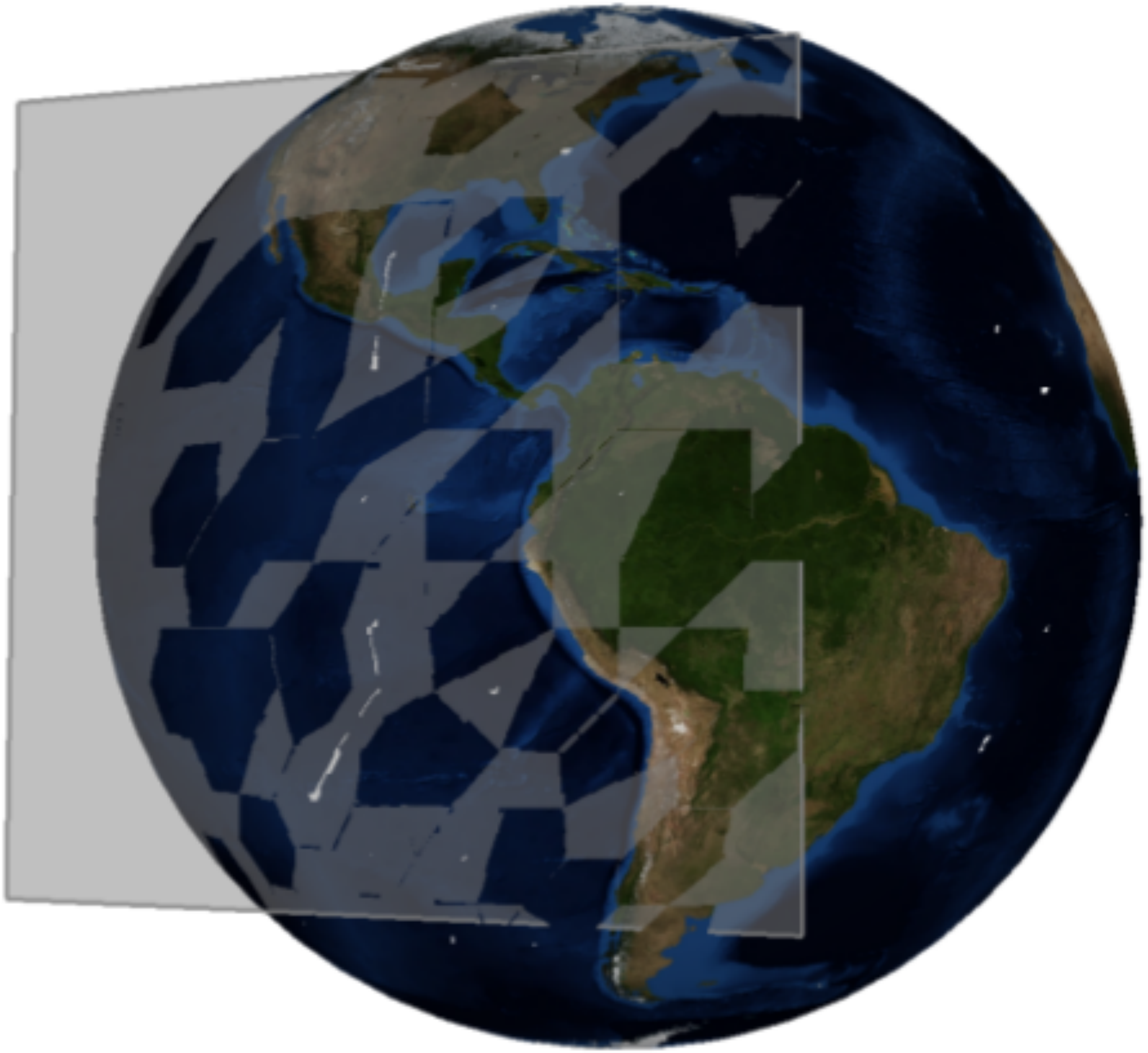
- Ideally, we want:
  - *near* = 0.00000001   // very near zero
  - *far* = ∞ // very far away
- Let's try...

Depth Buffer Precision



near = 35 m
far = 27,000,000 m

Depth Buffer Precision



near = 1 m

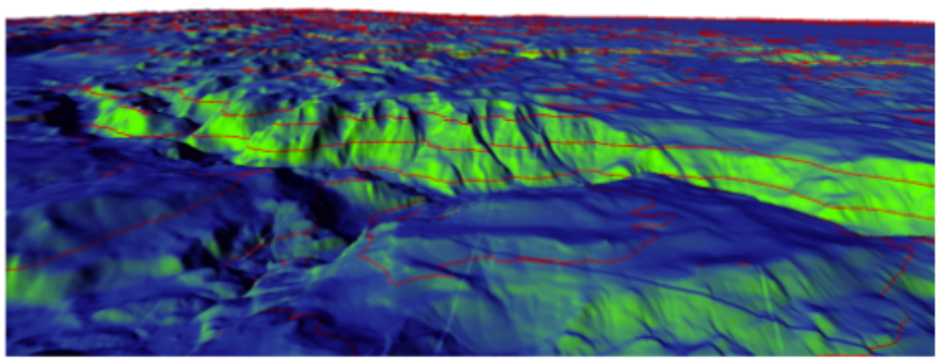
far = 27,000,000 m

Demo

- Depth Buffer Precision

Depth Buffer Precision

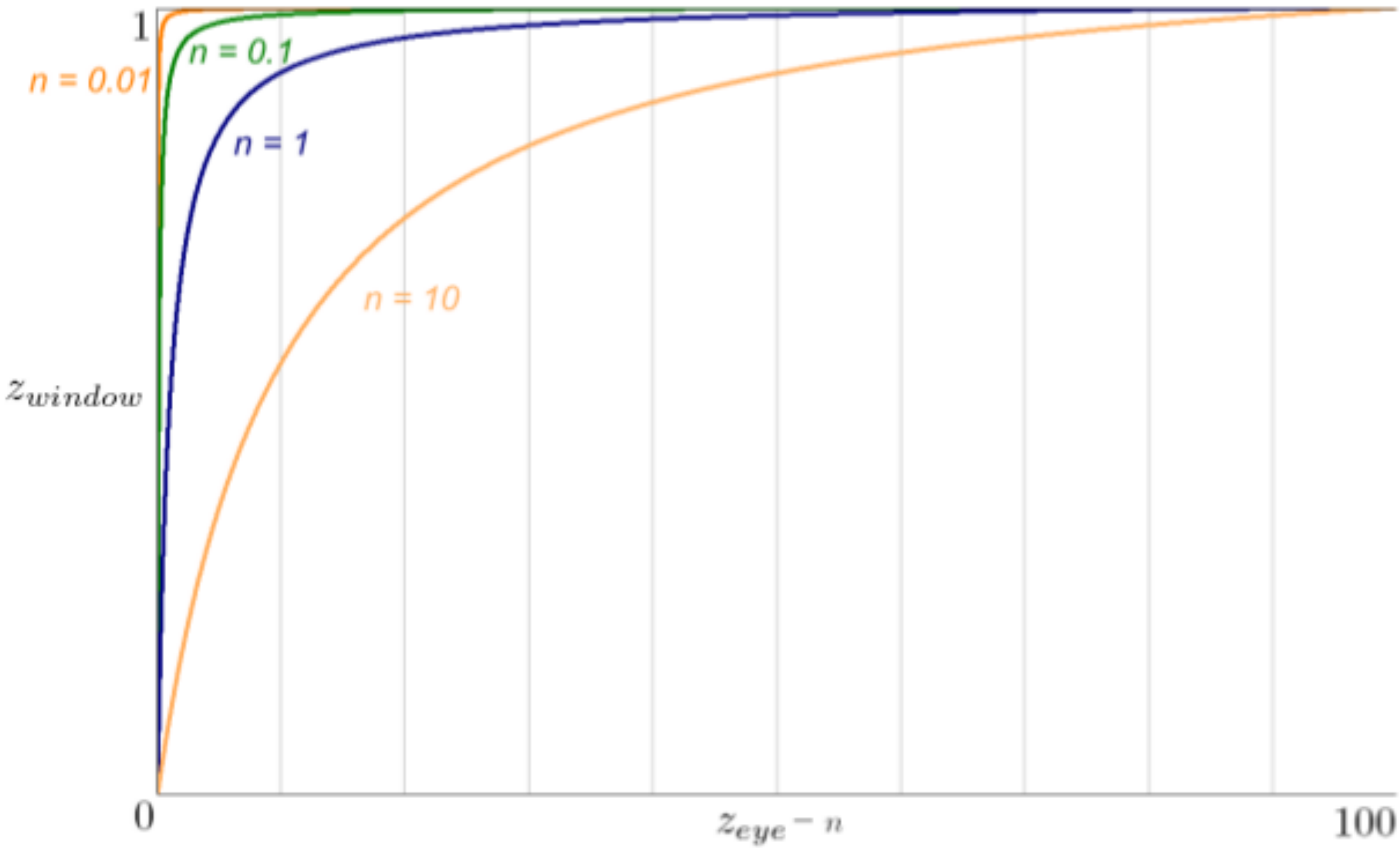
- Near-to-far ratio impacts depth buffer precision



Depth Buffer Precision

$$z_{\text{window}} = \left(\frac{f+n}{f-n} + \frac{2fn}{z_{\text{eye}}(f-n)}\right)\left(\frac{1}{2}\right) + \frac{1}{2}$$

$$= \frac{\left(\frac{f+n}{f-n} + \frac{2fn}{z_{\text{eye}}(f-n)}\right) + 1}{2}$$



Depth Buffer Precision

Basic Solutions

- Push near plane out as far as possible?
- Push far plane out as far as possible?
- Use *fog* or *blending* in the distance?
- Remove distant objects?
- *Complementary Depth Buffering*

Logarithmic Depth Buffer

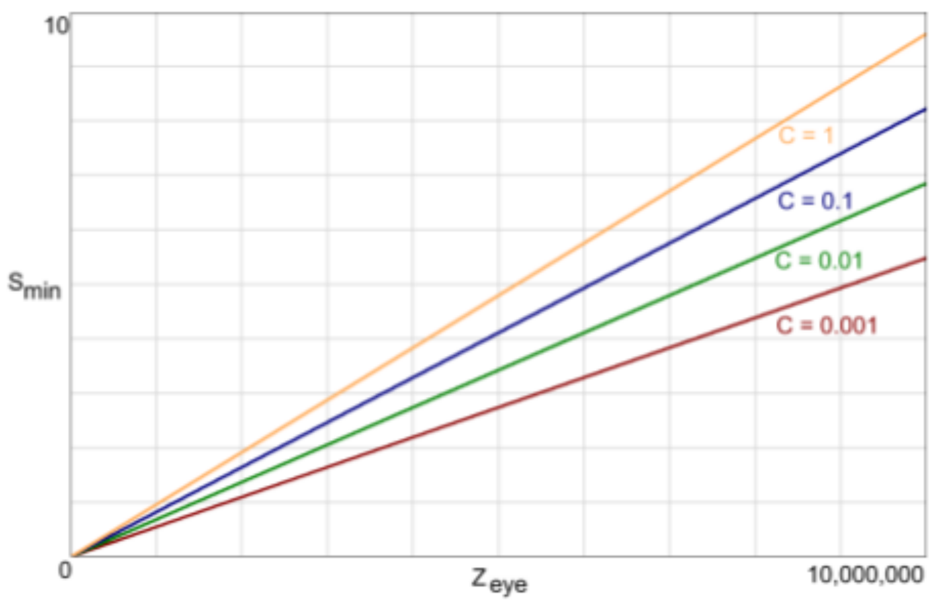
- Logarithmic distribution for z_{screen}
- Trades close object precision for distant object precision
- Use a vertex shader or fragment shader

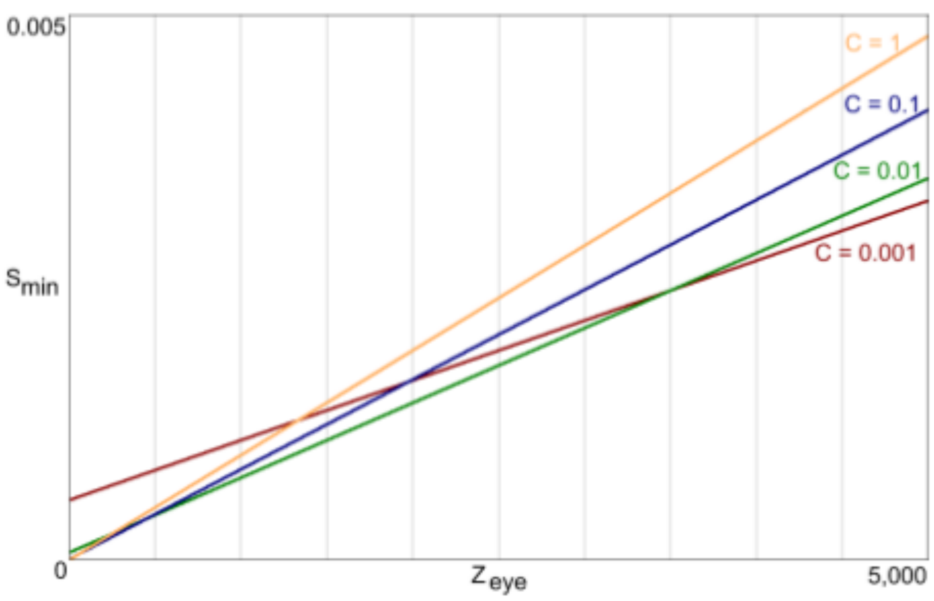
$$z_{\text{clip}} = \frac{2\ln(Cz_{\text{clip}} + 1)}{\ln(Cf + 1)} - 1$$

- *C* determines the resolution near the viewer...

See [Kemen09]

Logarithmic Depth Buffer



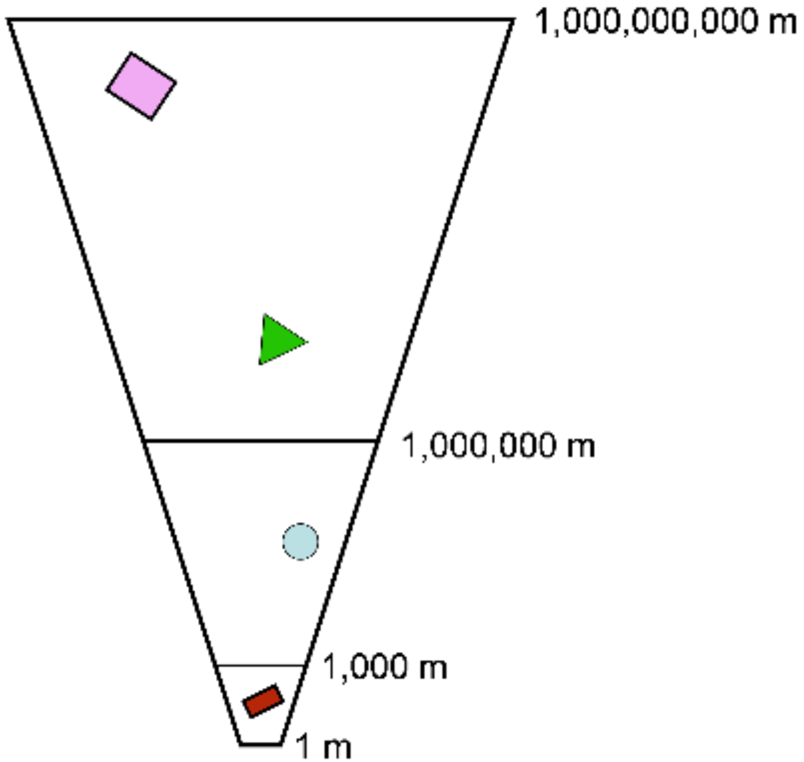


Demo

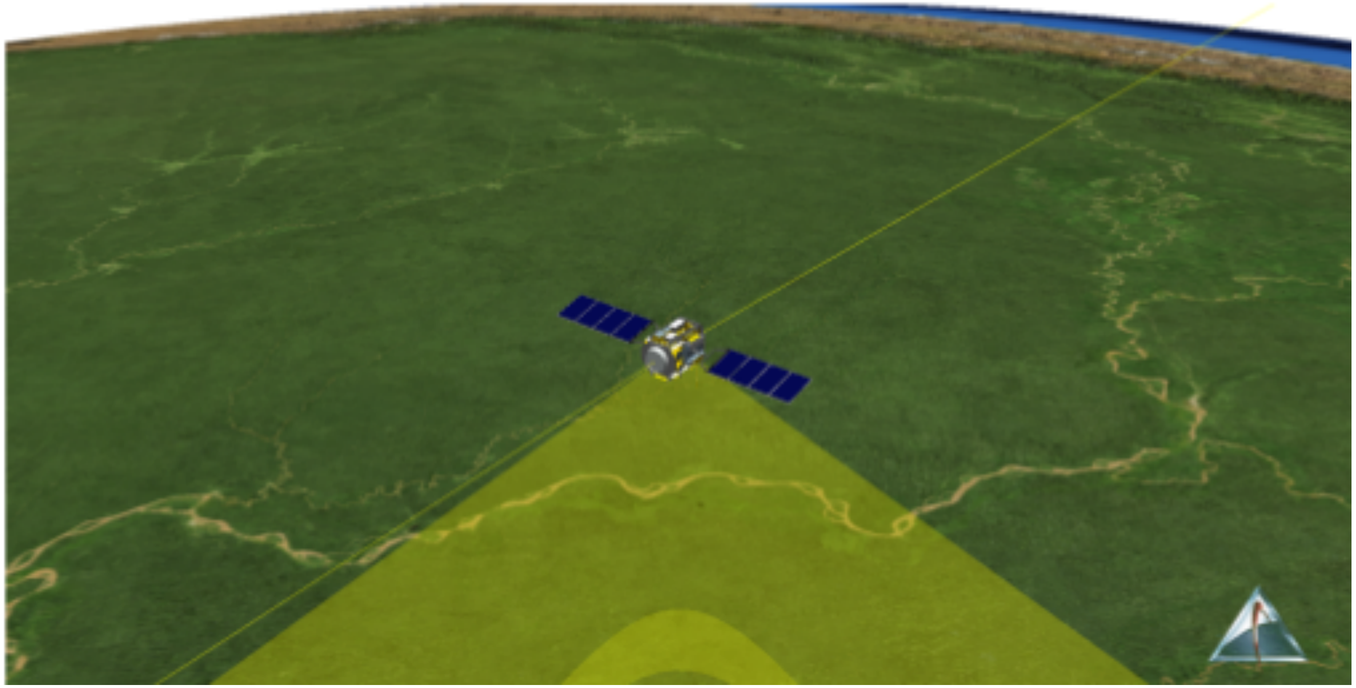
- Logarithmic Depth Buffer

Rendering with Multiple Frustums

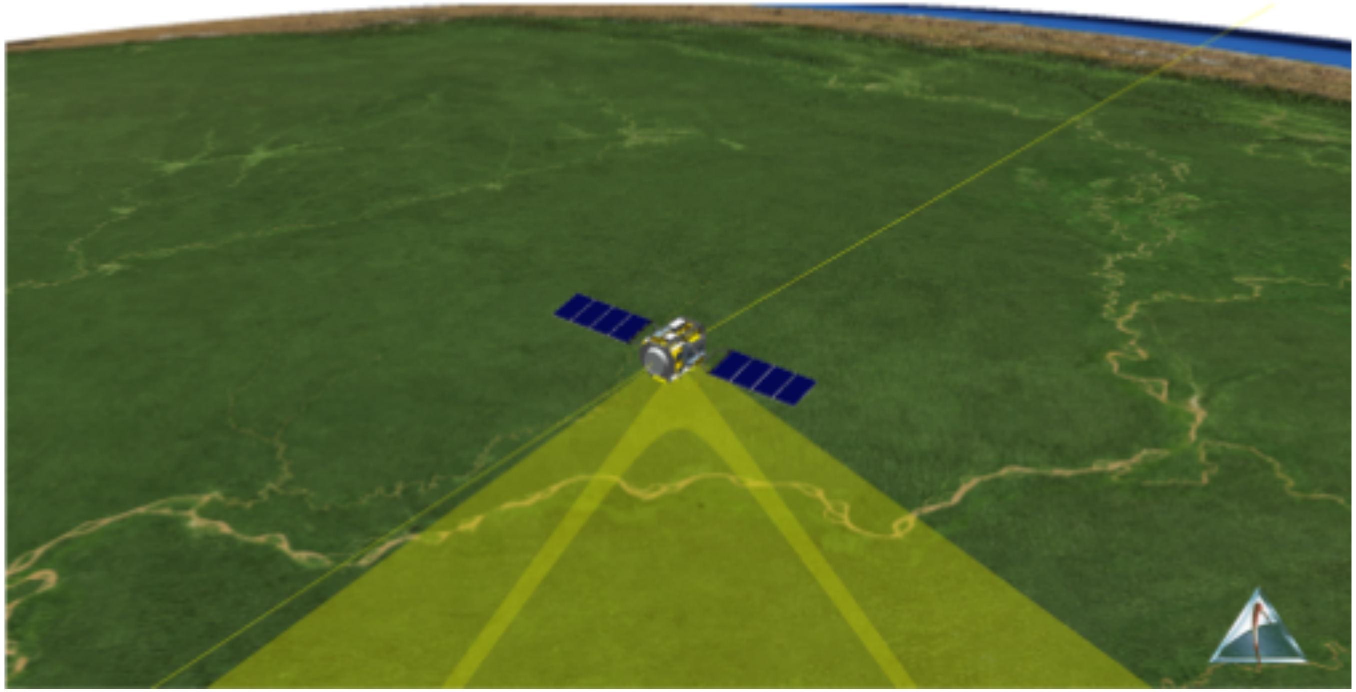
- Maintain near-to-far ratio of 1000 by using multiple view frustums rendered back to front
- Virtually infinite precision
- Performance Implications
  - Redundant computations
  - Culling
  - Careful batching
  - Early-z
- Visual artifacts...



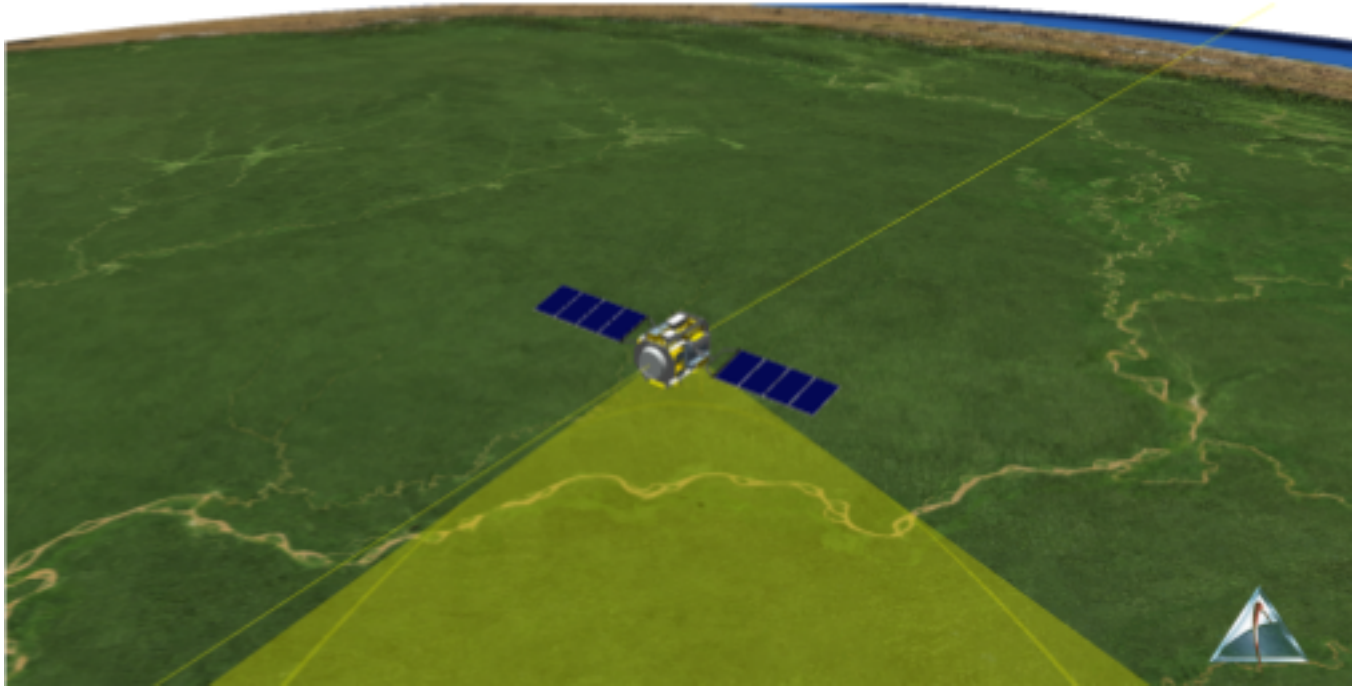
Rendering with Multiple Frustums



Rendering with Multiple Frustums



Rendering with Multiple Frustums



Parallelism

Parallelism Everywhere

CPUs

- Instruction-level parallelism
  - Pipelining
  - Superscalar / Out-of-order execution
- Data-level Parallelism
  - SIMD operations
- Thread-level Parallelism
  - Hyper-threading
  - Multicore

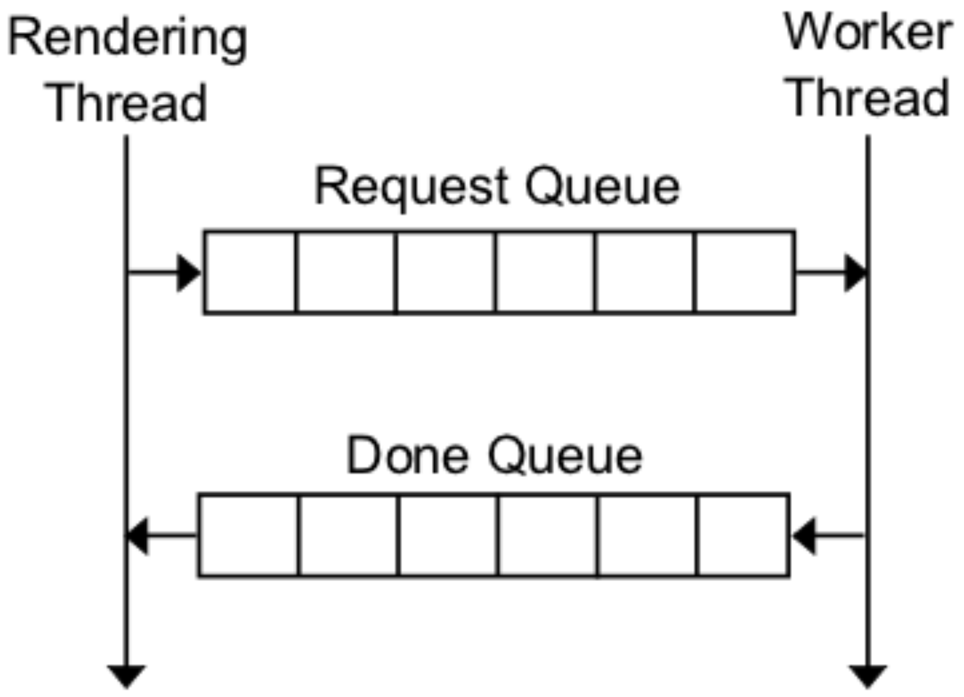
GPUs

- Rasterization is embarrassingly parallel
- SIMT

Parallelism

- Virtual globes use lots of data:
    - Terrain
    - Imagery
    - Vector data
- Rendering preparation requires
    - I/O
    - CPU-intensive processing
        - Triangulation
        - Texture-atlas packing
        - LOD creation
        - decompression
- What happens if preparation and rendering occur in the same thread?

Coarse-Grained Threads



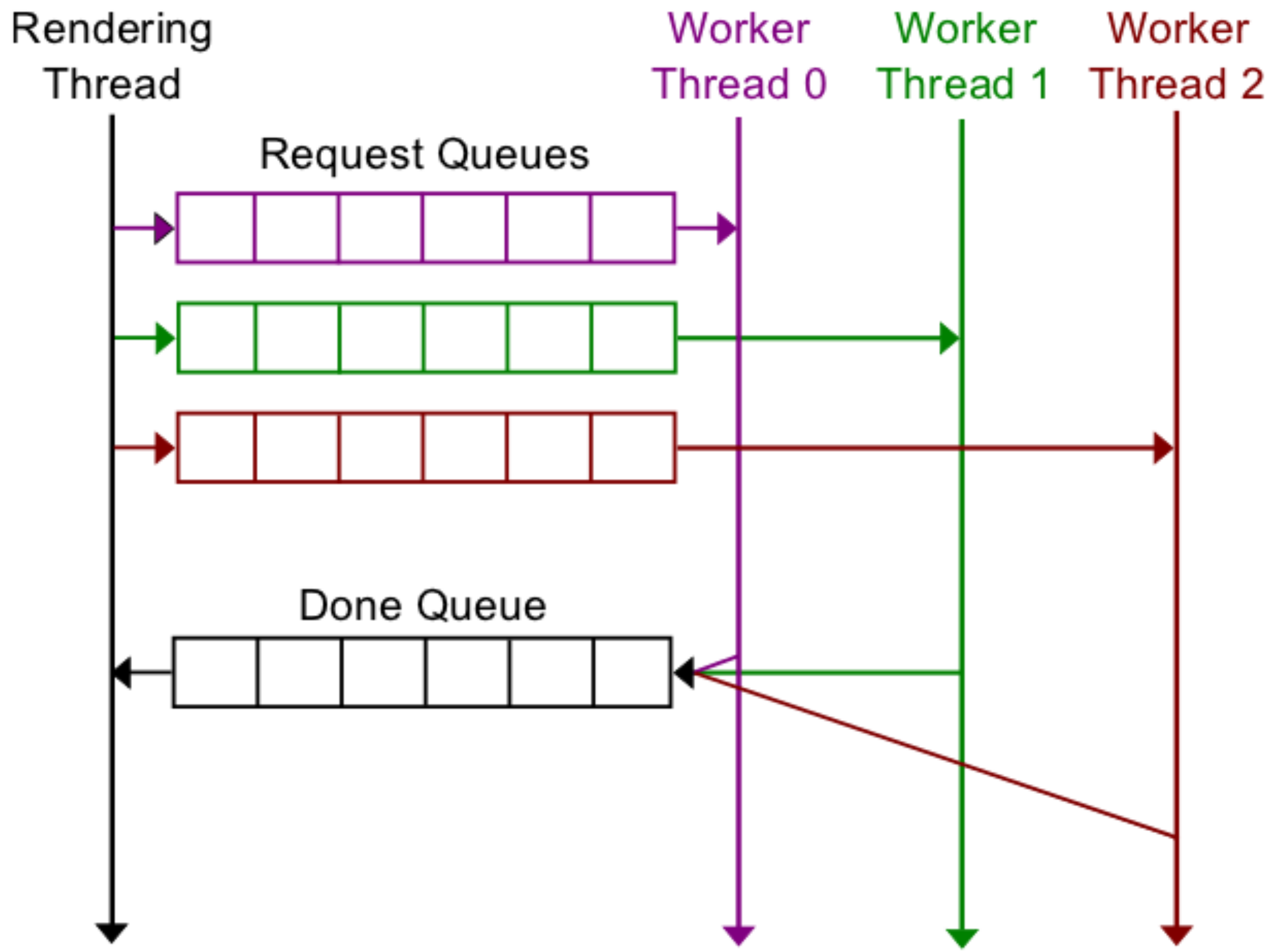
Message queues are great for communicating between threads

Coarse-Grained Threads

- Doesn't fully utilize
  - A second core
  - I/O bandwidth
- Responsiveness is not ideal

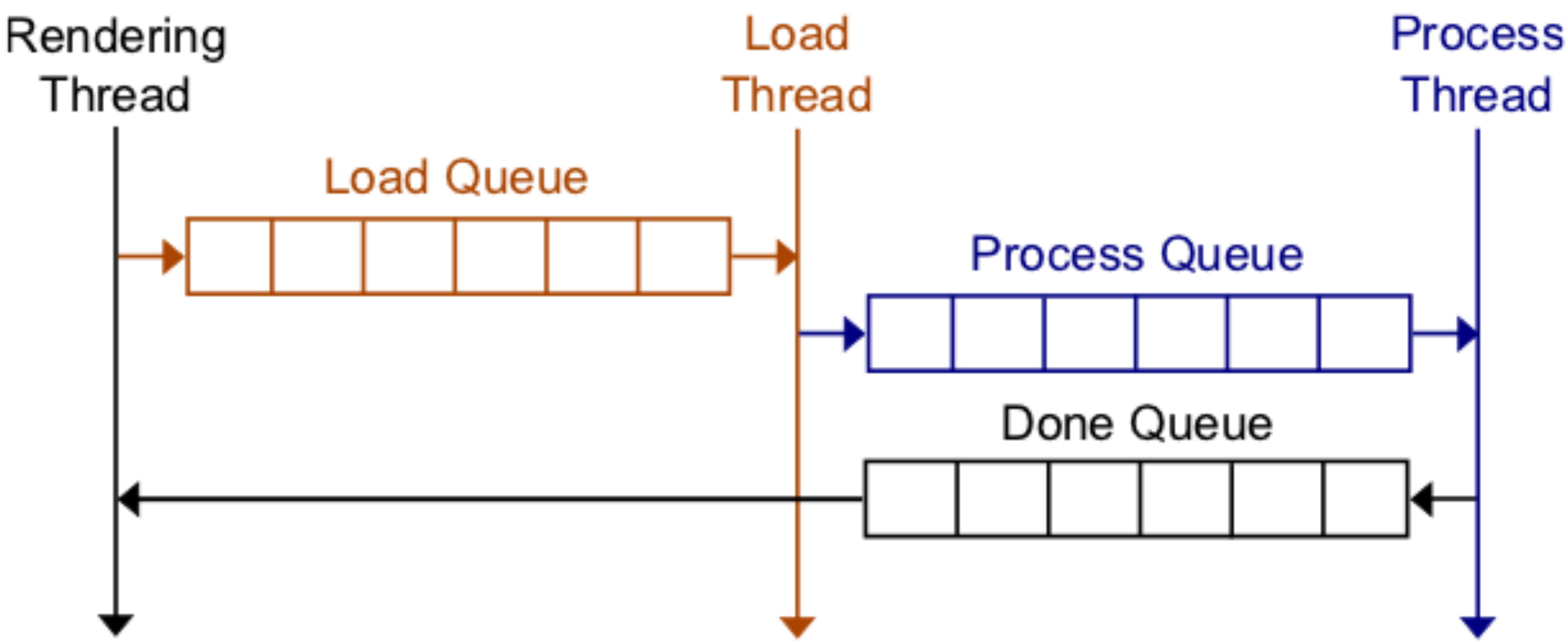
- Use multiple workers...

Multiple Worker Threads



How many threads?
How to schedule?

Fine-Grained Threads



- Attempting better throughput but longer latency
- How many threads?

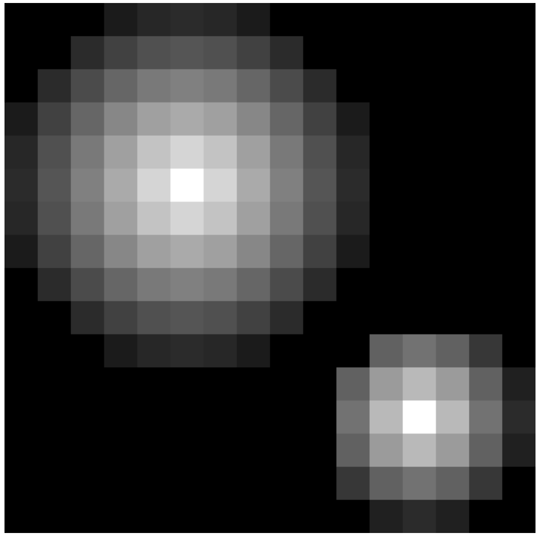
Other Architectures

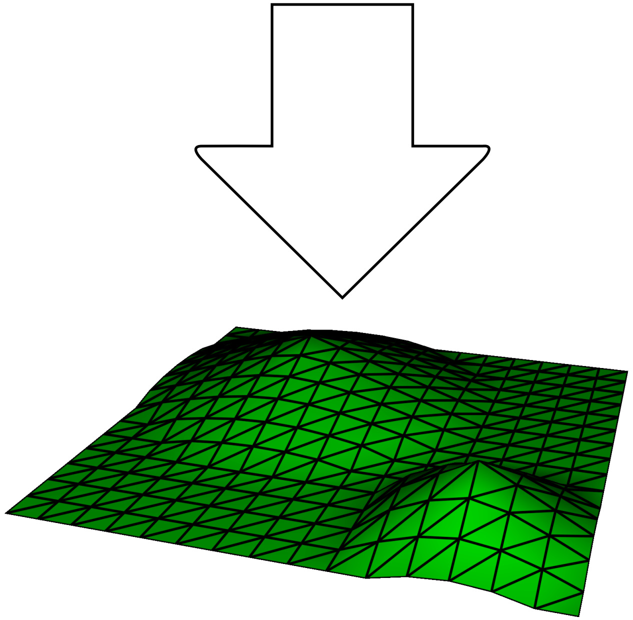
- Asynchronous I/O
- Parallel Job Systems

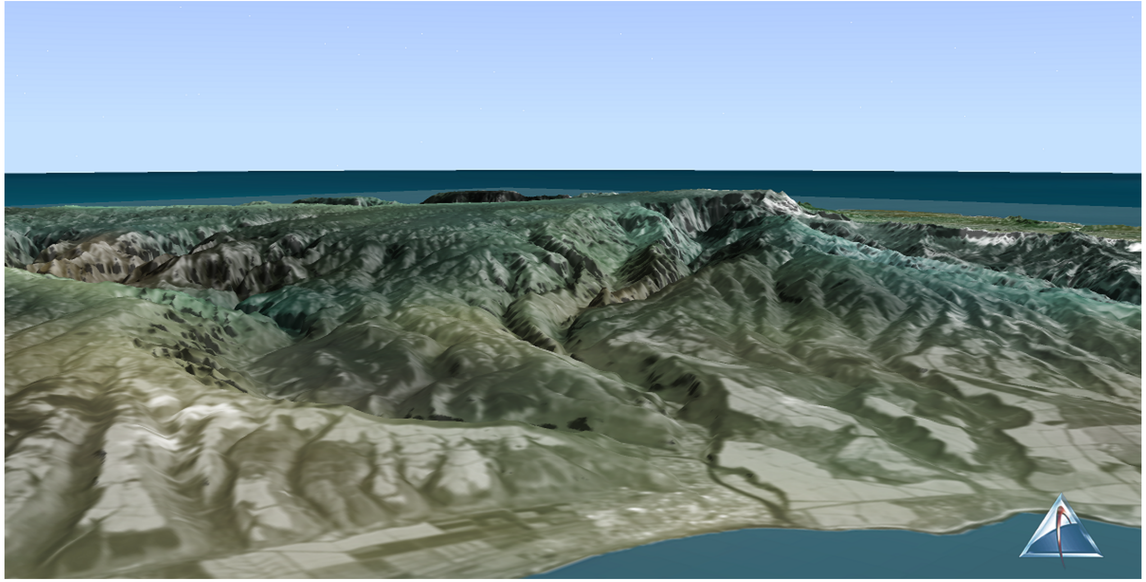
Terrain

Terrain Representations - Height Map

- Grayscale image
- Intensity of pixel represents the height at that position
- Extruded pixels are called posts
- Most widely used representation in virtual globes
- No cliffs or caves

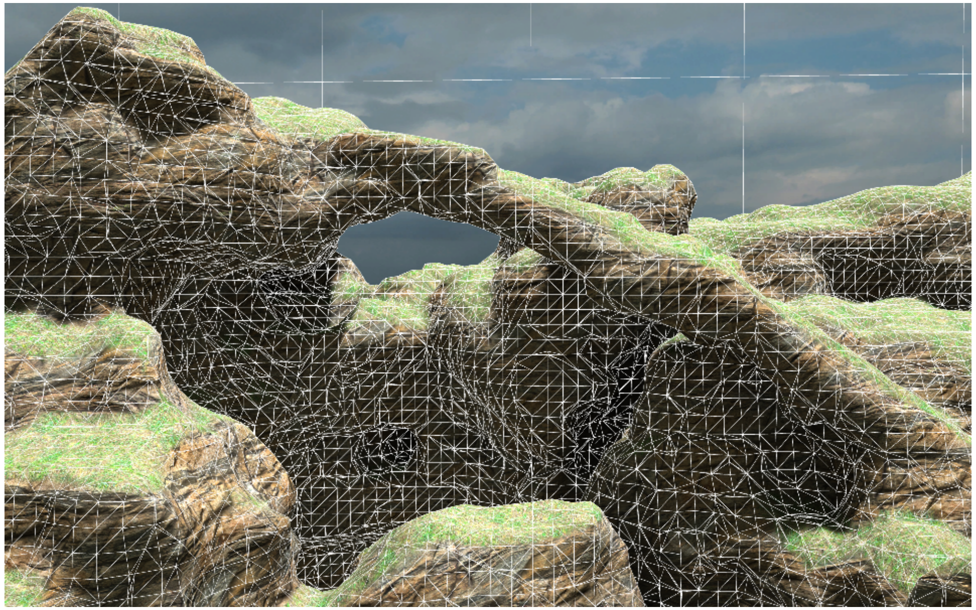






Terrain Representations - Voxels

- The 3D extension of a pixel
- Cliffs and caves are possible and common
- Rendered using raycasting or triangulated using marching cubes
- Often represented using hierarchical data structures like sparse octrees
- Uncommon in virtual globes



Images courtesy Eric Lengyel, Terathon Software

Terrain Representations - Implicit Surface

- Minimal data
- Terrain generated from an implicit function
    - Density function, fractal
- Entire real-world terrain as implicit surface not really feasible
- Virtual globes can use fractal fine detail to great effect

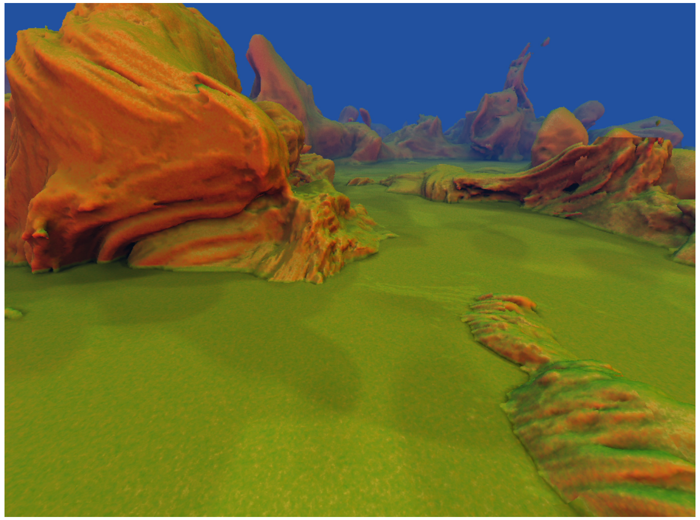
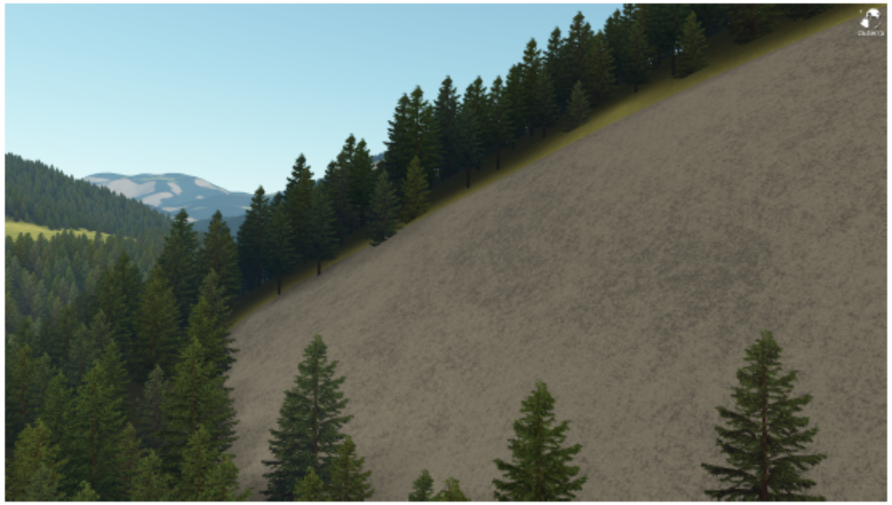
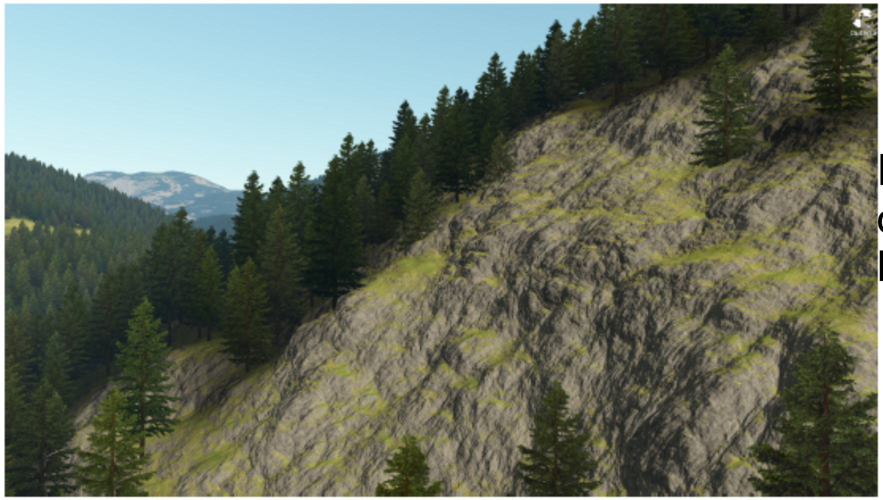


Image courtesy of NVIDIA Corporation and Ryan Geiss

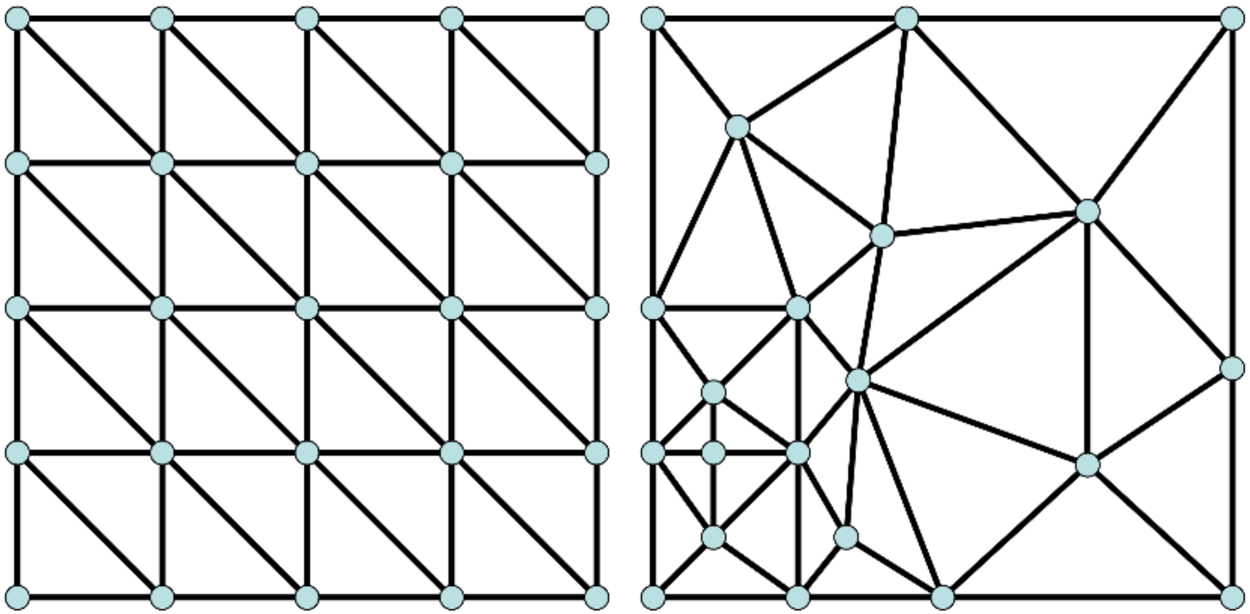




Images courtesy of Brano Kemen, Outerra

Terrain Representations - TIN

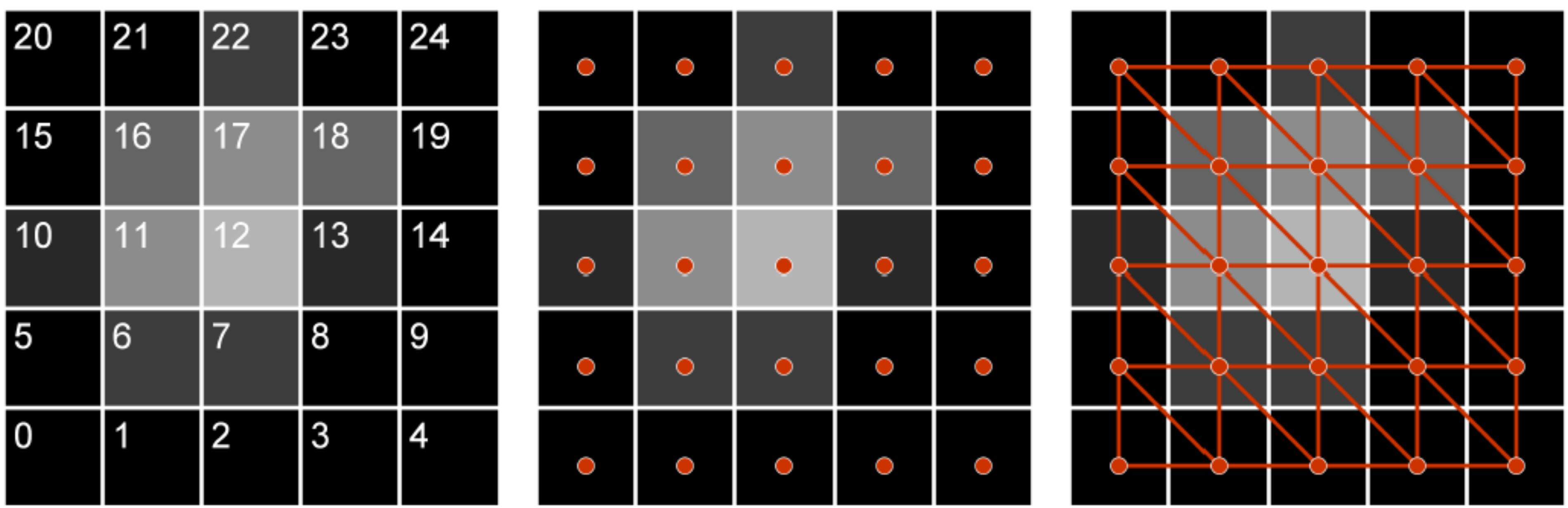
- Triangular Irregular Network
- Basically just a triangle mesh!
- Large triangles cover flat regions
- Small triangles represent fine features



Rendering Height Maps

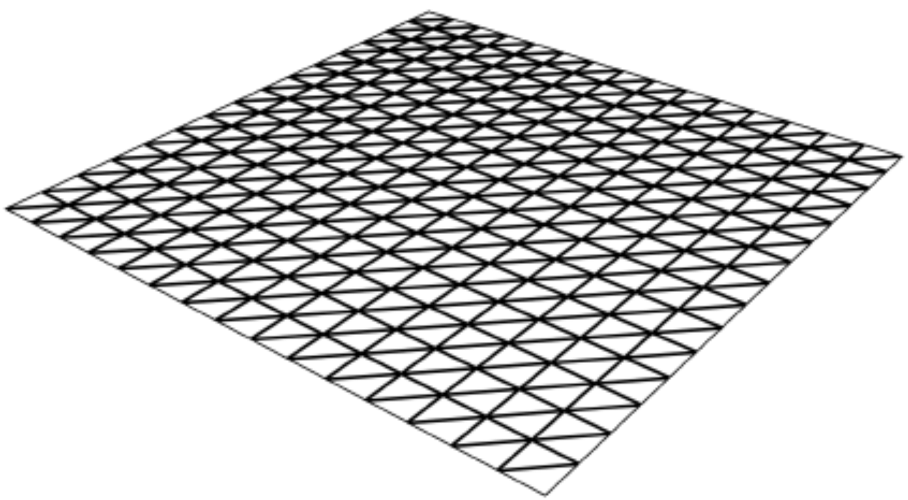
Create a Triangle Mesh on the CPU

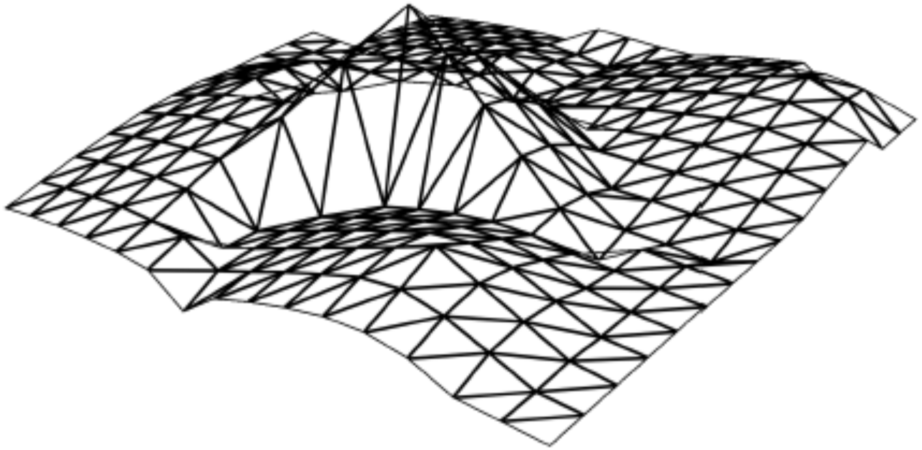
- Create a vertex at each pixel location
- Connect surrounding vertices with triangle edges
- Reduce memory by sharing index buffer between tiles



Rendering Height Maps
Vertex-Shader Displacement Mapping

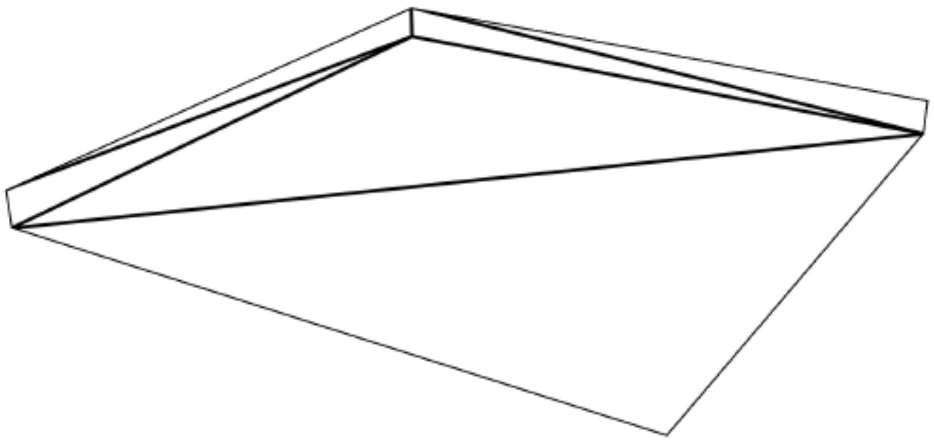
- Create a planar triangle mesh
- Pass the height map to the vertex shader as a texture
- Vertex shader samples the texture and displaces the vertex
- Much lower memory usage
- What about terrain on a globe?

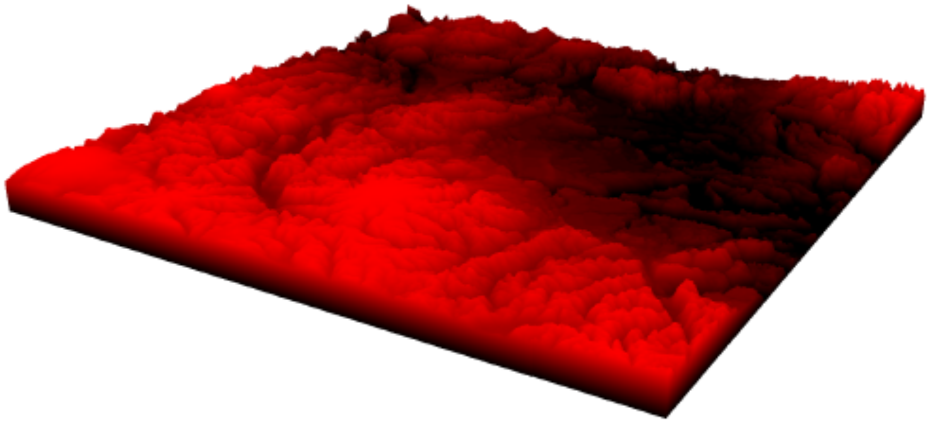




Rendering Height Maps
GPU Ray Casting

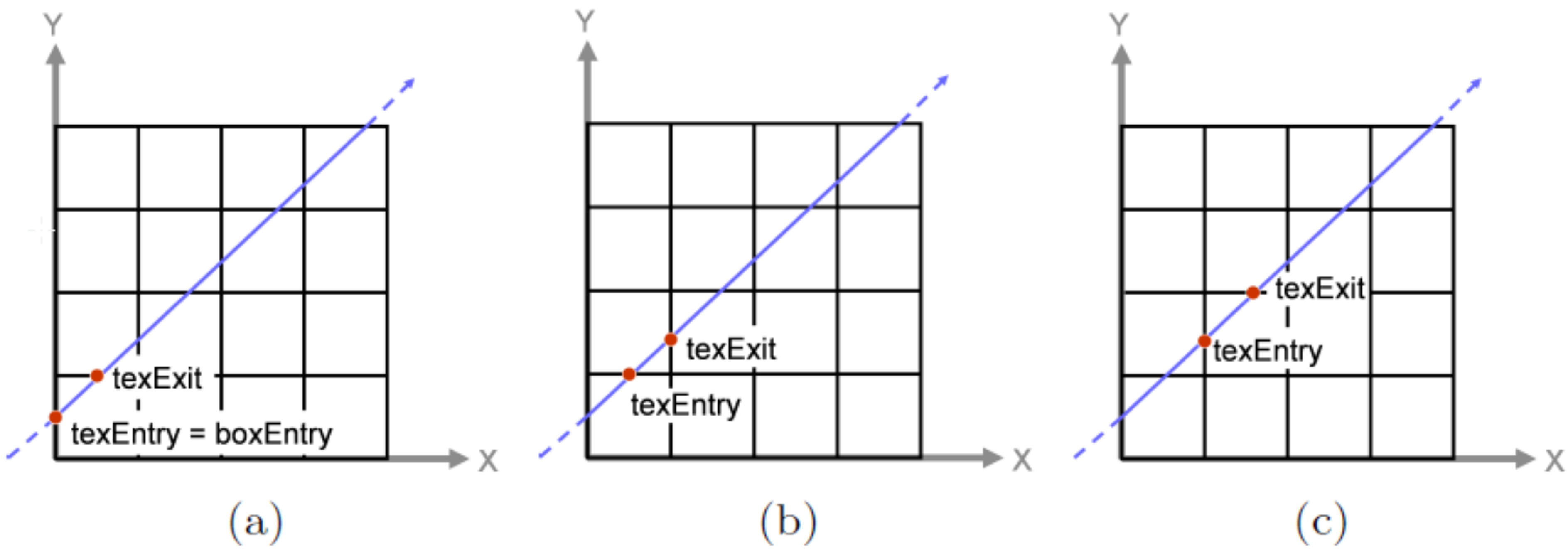
- Render a simple axis-aligned bounding box (AABB) with front-face culling enabled
- Vertex Shader: pass AABB exit point to fragment shader
- Fragment Shader: compute AABB entry point, step along ray





Rendering Height Maps
GPU Ray Casting (continued)

- Compute entry and exit of each height map texel
- If the height of the ray is below the texel height, intersection occurs, so shade the fragment
- If all ray entry/exit points are above the terrain, discard the fragment



Shading Terrain
Color Maps

- Often derived from real satellite imagery
- Texture coordinates used in the fragment shader to sample a color

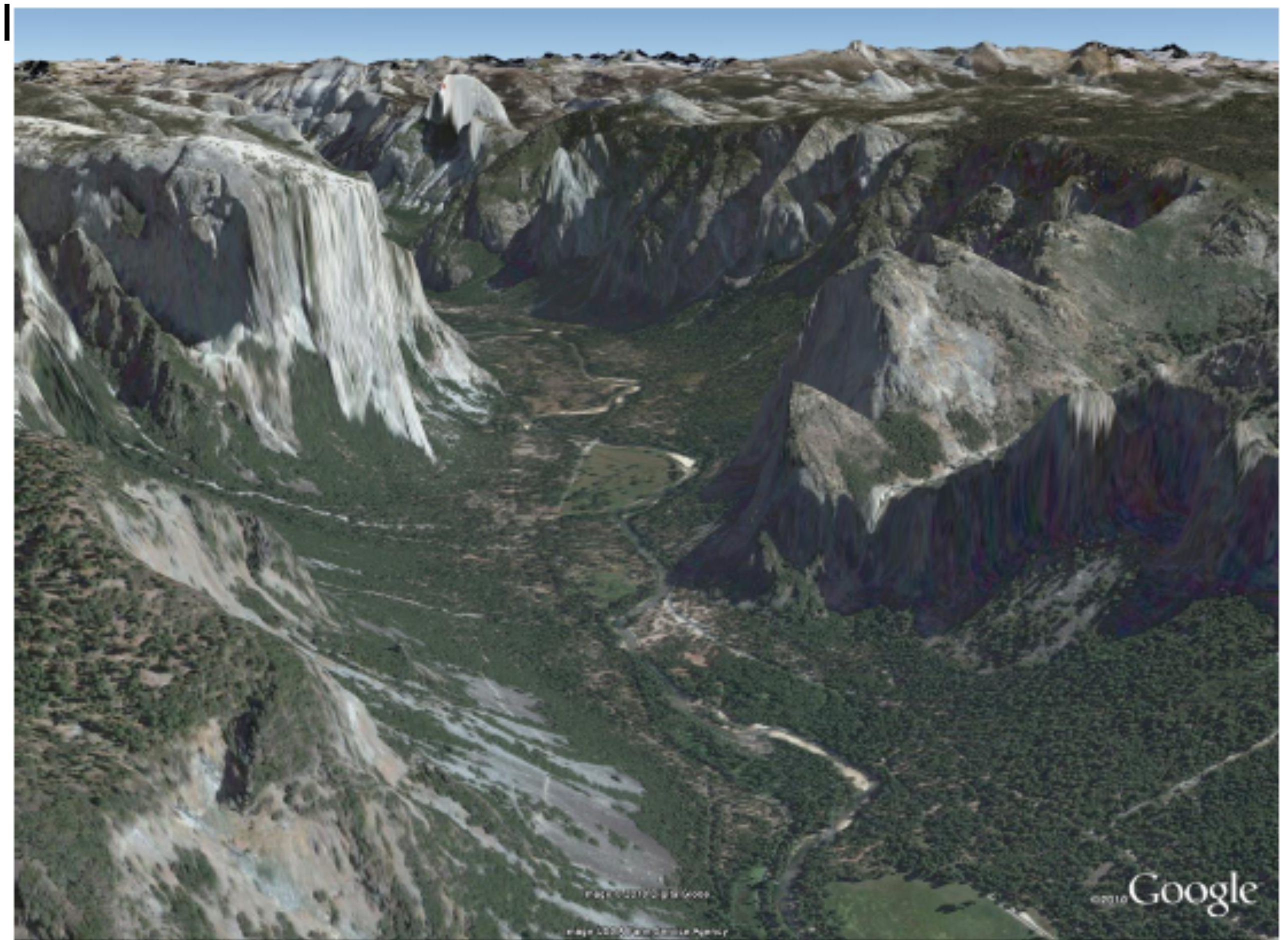
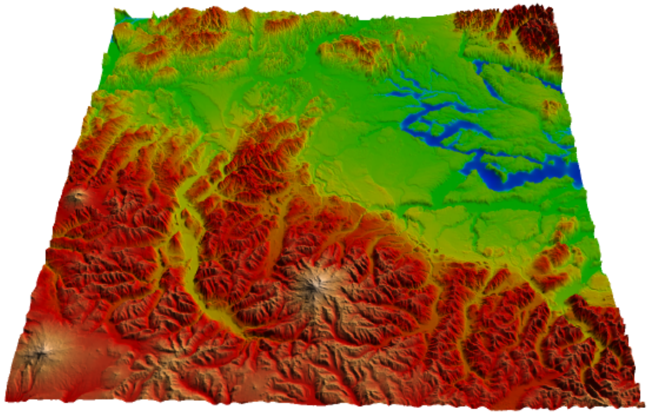


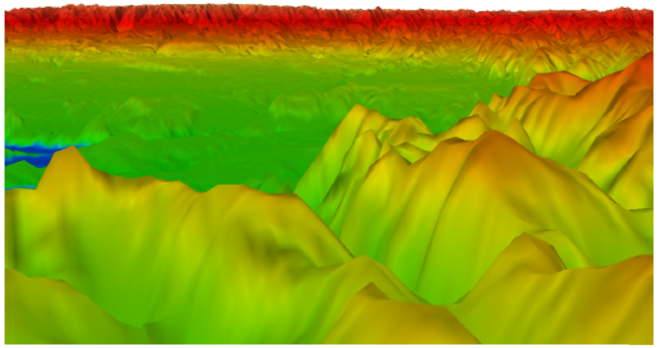
Image (C) USDA Farm Service Agency and DigitalGlobe. Taken using Google Earth.

Shading Terrain
Normals and Lighting

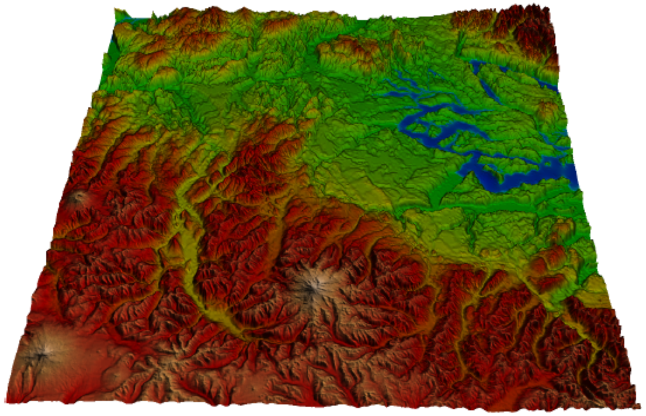
- Compute normals and shade based on realistic sun position
- With vertex displacement or GPU raycasting, normals can easily be computed in a shader

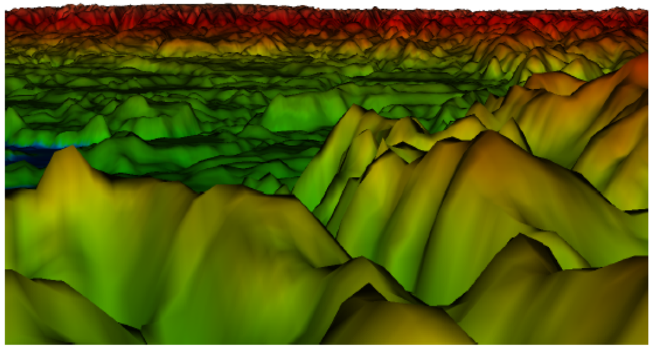
Color map only





With Lighting





Shading Terrain

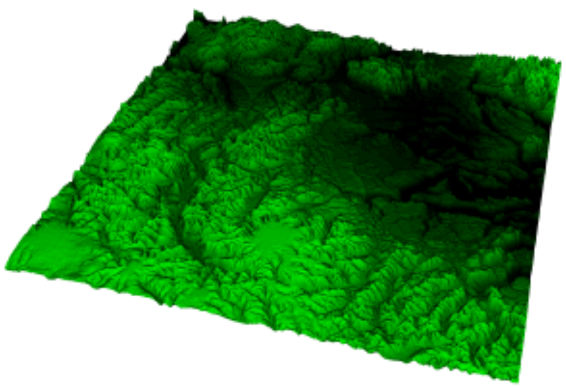
Procedural Shading by Slope/Height

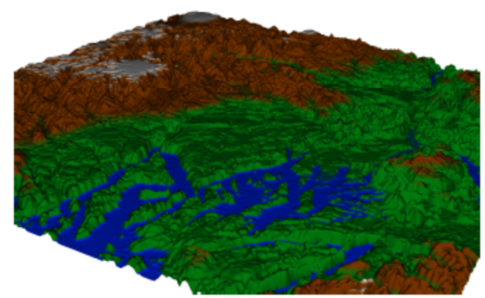
Intensit
y

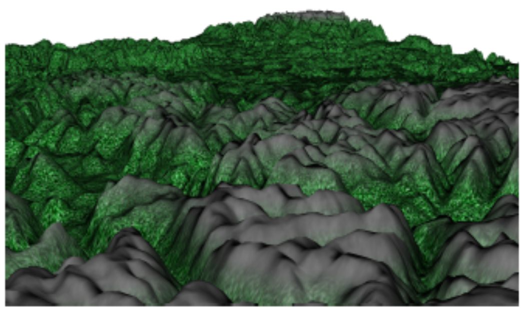
Color Ramp

Blend Ramp

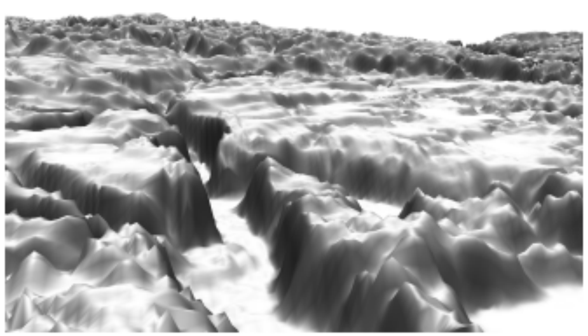
Height

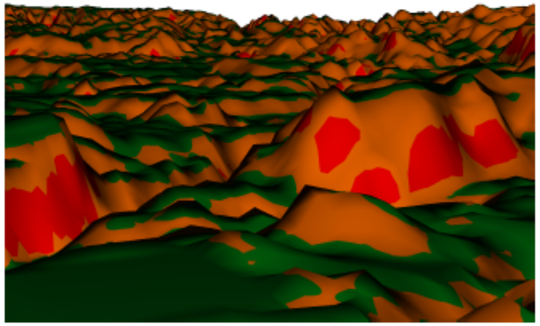


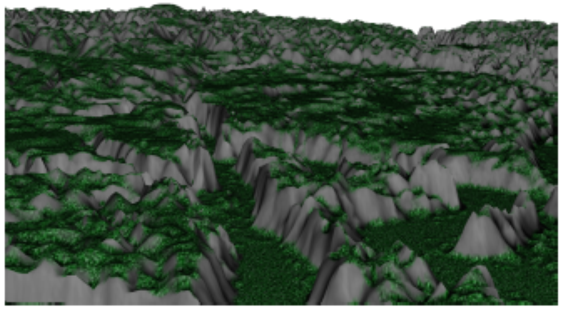




Slope







What about **planet-sized** terrains?

Popular virtual globes measure their terrain and imagery data in *terabytes*!

How do we fit that on a machine? Nevermind in GPU memory...

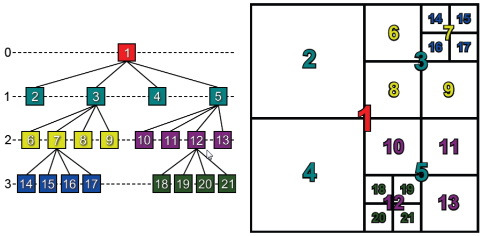
And worse, how in the world do we render it?

Today's GPUs can render hundreds of millions of triangles per second... but we're a long way from trillions per frame!

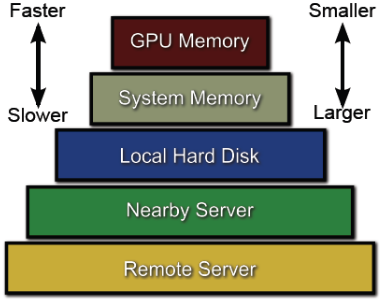
Terrain Rendering Facts

Virtual globe terrain datasets:

- Consist of far too many triangles to render with brute-force techniques.
  - ➡ We need terrain level of detail (LOD).



- Are much larger than available memory.
  - ➡ We need out-of-core (OOC) rendering.



Level of Detail

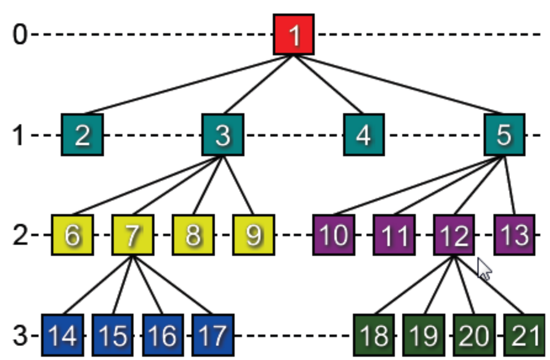
- LOD algorithms reduce an object's complexity when it contributes less to the scene
  - For example, objects in the distance are rendered with less geometry and lower resolution textures than nearby objects

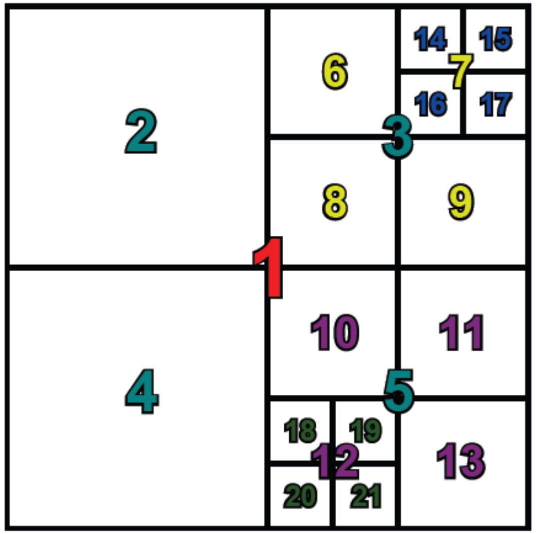
- ***Generation*** creates different versions of a model
- ***Selection*** chooses the appropriate version of the model to render
- ***Switching*** changes from one version of a model to another

- Three broad types of LOD: *Discrete*, *Continuous*, and **Hierarchical**

Hierarchical Level of Detail (HLOD)

- Operates on chunks, patches, or tiles of terrain geometry
  - Levels of detail are generated, selected, and switched at this granularity.
- Chunks are organized in a hierarchical data structure such as a quadtree or octree
  - Higher-resolution chunks are logically children of lower-resolution chunks.





Why Hierarchical Level of Detail?

- Modern terrain-rendering algorithms focus on:
  - Reducing the processing done by the CPU
  - Reducing the quantity of data sent to the GPU
  - NOT on achieving an optimally small number of triangles

- In HLOD:
  - The CPU only needs to select and switch chunks, not individual triangles.
  - Chunks are valid for a wide range of views, so less overall data is sent to the GPU

- *Plus: HLOD integrates naturally with out-of-core rendering (more on that later)*

Hierarchical LOD Switching

How do we decide when to switch between chunks of different detail in an HLOD scheme?

Goal: Render with the simplest LOD possible while still rendering a scene that looks good.

But how do we determine whether an LOD will provide a scene that looks good?

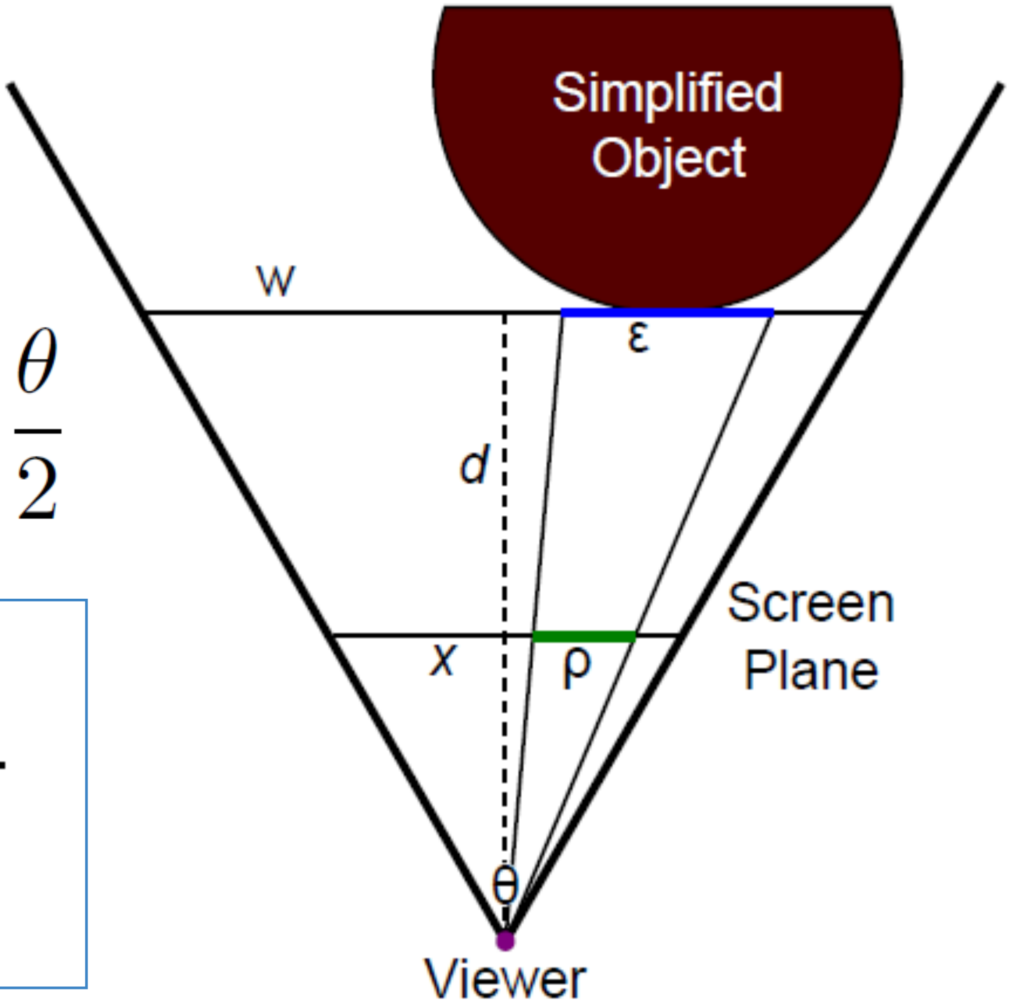
Screen-Space Error

The number of pixels of difference that would result from rendering a lower-detail version of an object rather than a higher-detail version.

$$\frac{\epsilon}{w} = \frac{\rho}{x}$$

$$\rho = \frac{\epsilon x}{w} \qquad w = 2d \tan \frac{\theta}{2}$$

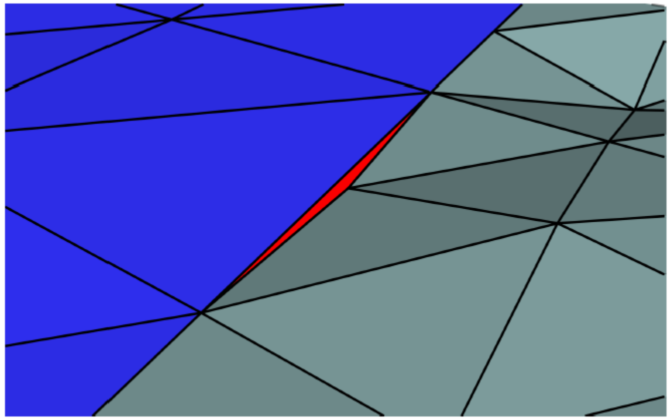
$$\boxed{\rho = \frac{\epsilon x}{2d \tan \frac{\theta}{2}}}$$



LOD Artifacts

Rendering problems introduced by an LOD scheme

- **Cracking**
  - Usually filled by skirts



- **T-junctions** - Tiny cracks caused by floating point rounding
  - Fill with degenerate triangles or skirts

Popping - Abrupt changes between different LODs
  - Blending, or Mantra of LOD: An LOD should only switch when that switch would be imperceptible to the user.

Preprocessing

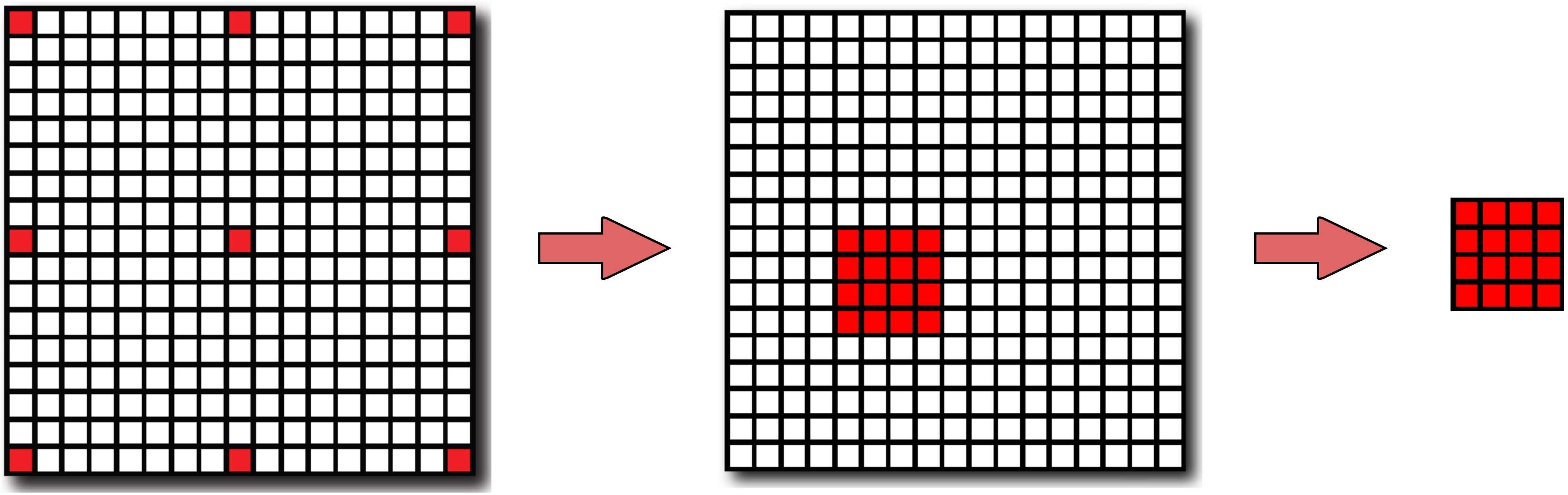
Rendering a planet-sized terrain dataset at interactive frame rates requires that the terrain dataset be preprocessed.

We wish this weren't true, but it is!

Preprocessing arranges the data so that the *subset* we need is available quickly.

Preprocessing Height Maps

At a minimum: mipmap and tile the height map



Some terrain rendering algorithms benefit from more aggressive preprocessing:

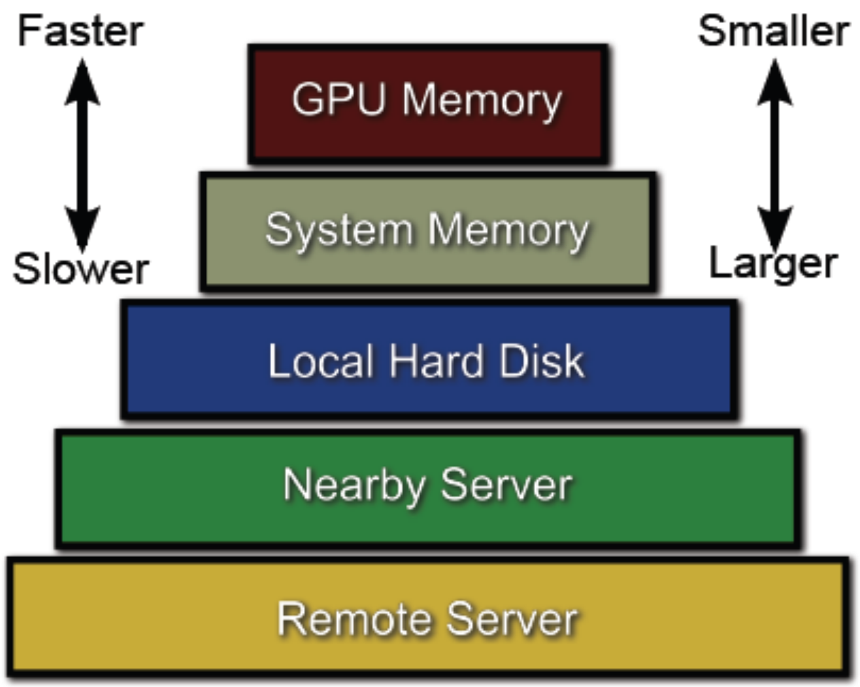
- Simplify "flat-ish" areas of the height map
- Compute geometric error bounds

Out of Core Rendering means...

Load-ordering policy - Bring new terrain data into memory as needed

Replacement policy - Unload old data to make room for new data

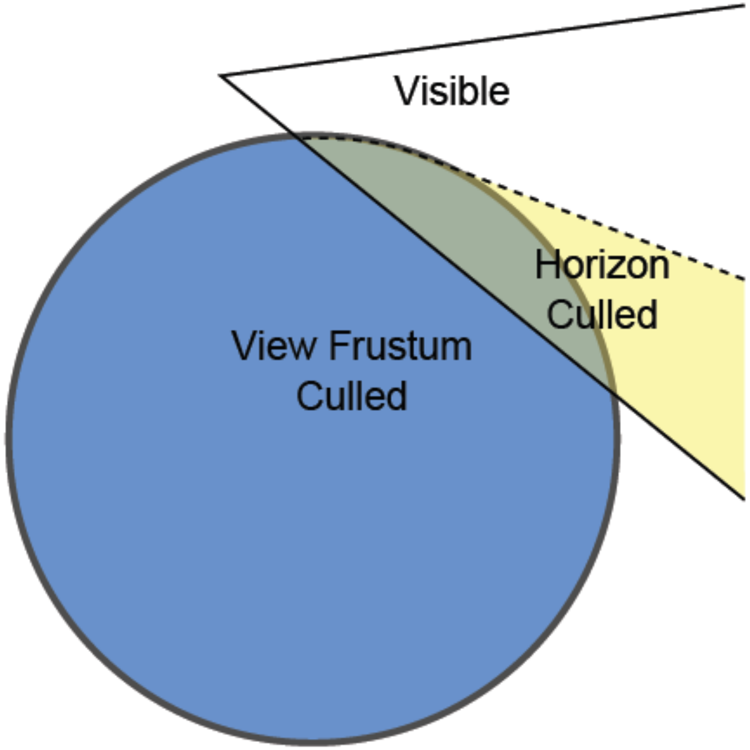
Prefetching - Predict the data that will be needed soon and load it

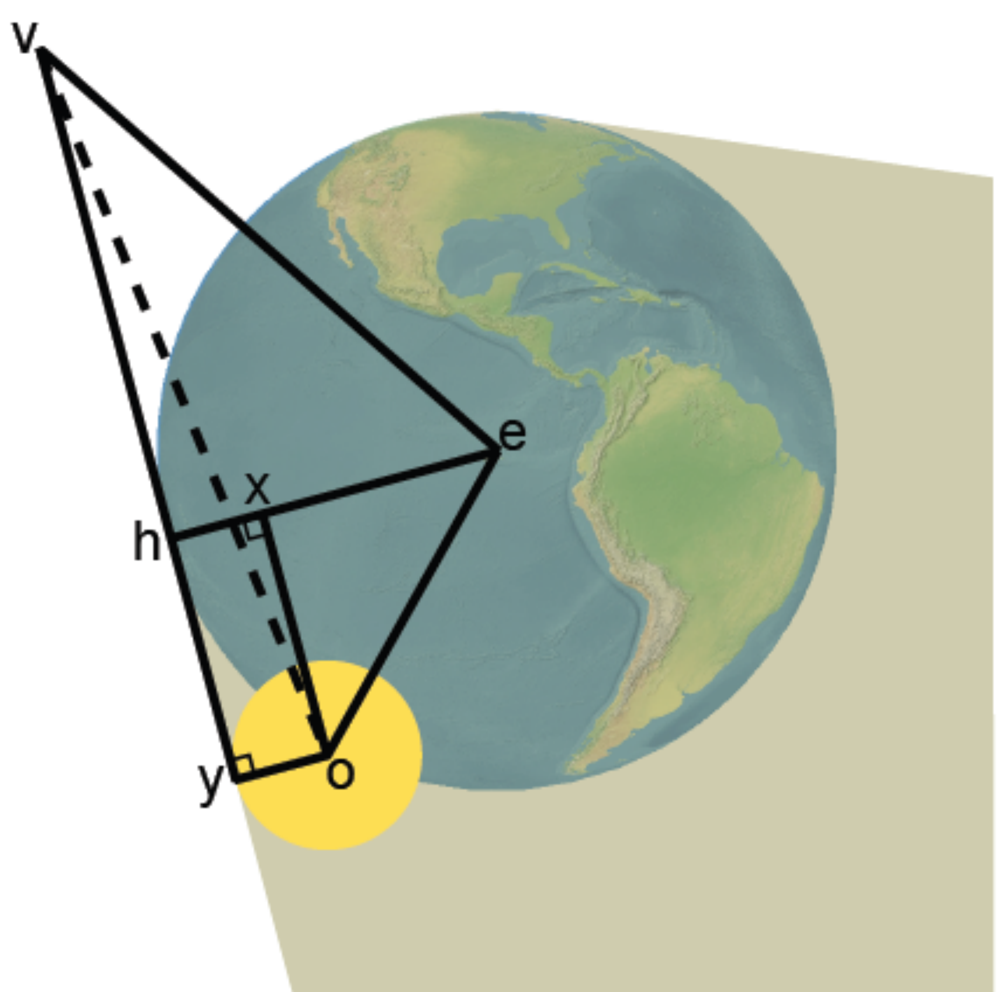


Out of core rendering is almost always a multithreaded process!

Horizon Culling

Don't render objects or terrain that are below the horizon





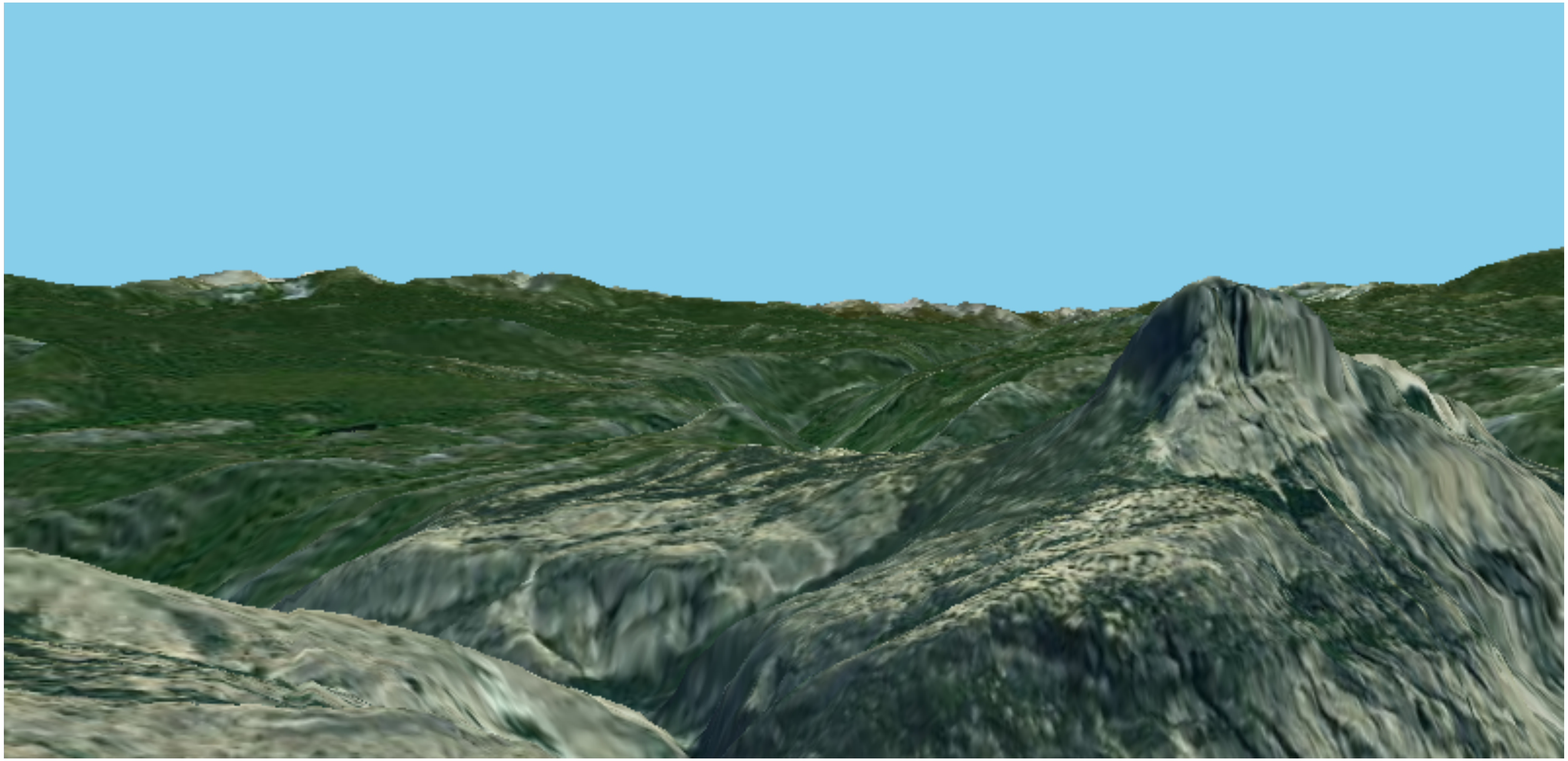
Geometry Clipmapping

Wherein we finally present a real terrain LOD algorithm

See [Losasso04]

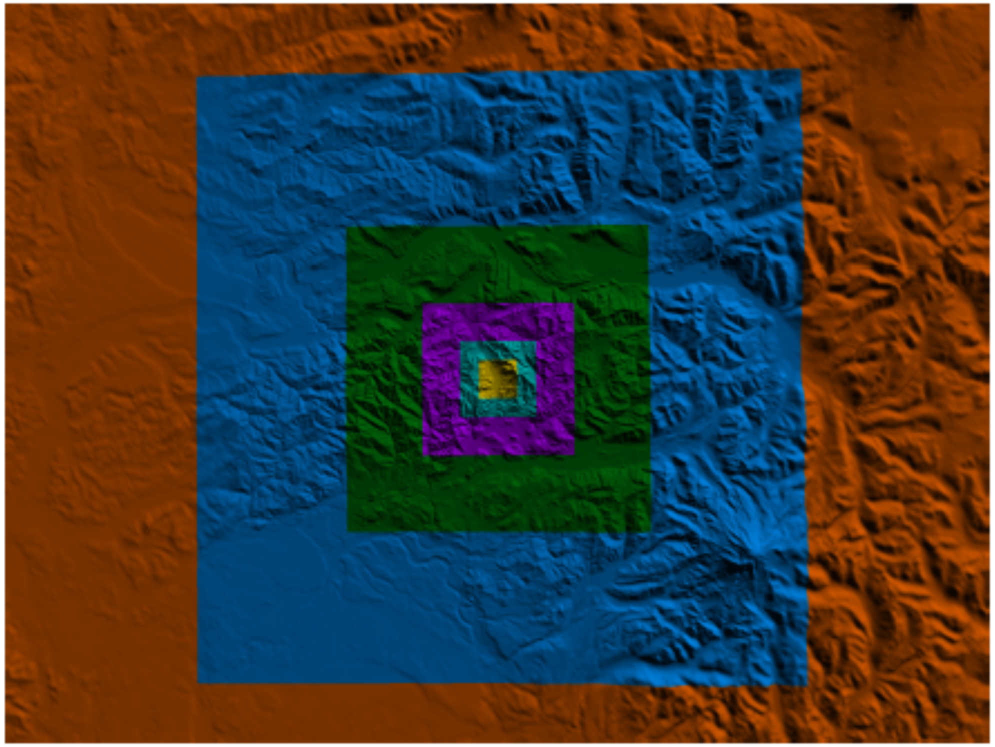
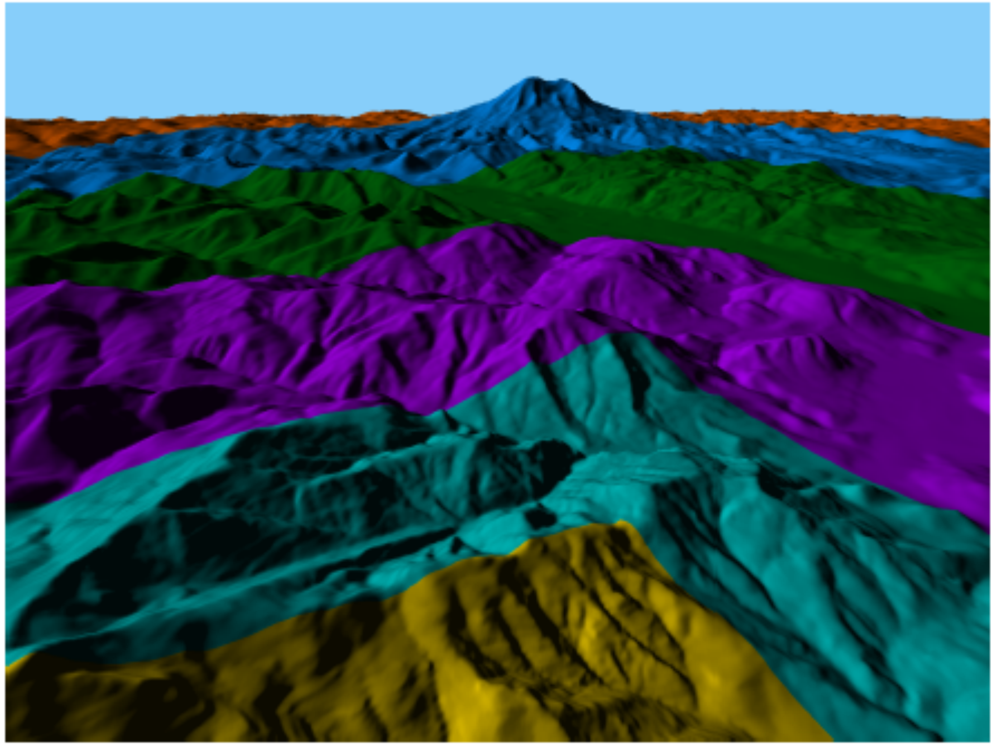
Geometry Clipmapping

- Renders terrain data in the form of a mipmapped, tiled height map - minimal preprocessing required
- Extremely GPU friendly - impressive triangle throughput
- Relatively easy to implement
- Legacy GPUs need not apply - needs vertex texture fetch



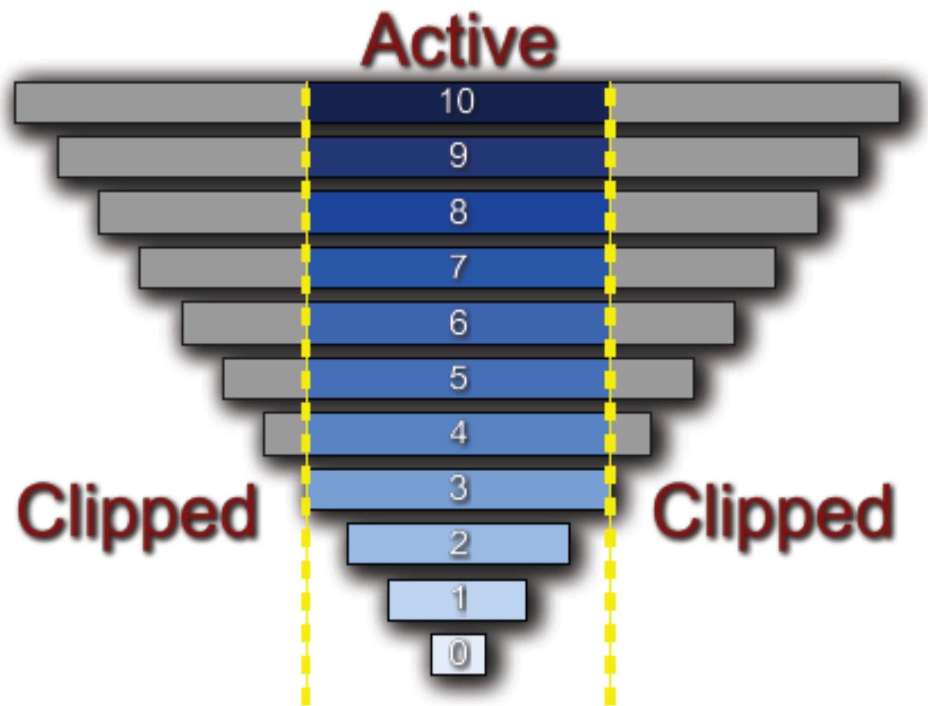
Structure of Geometry Clipmaps

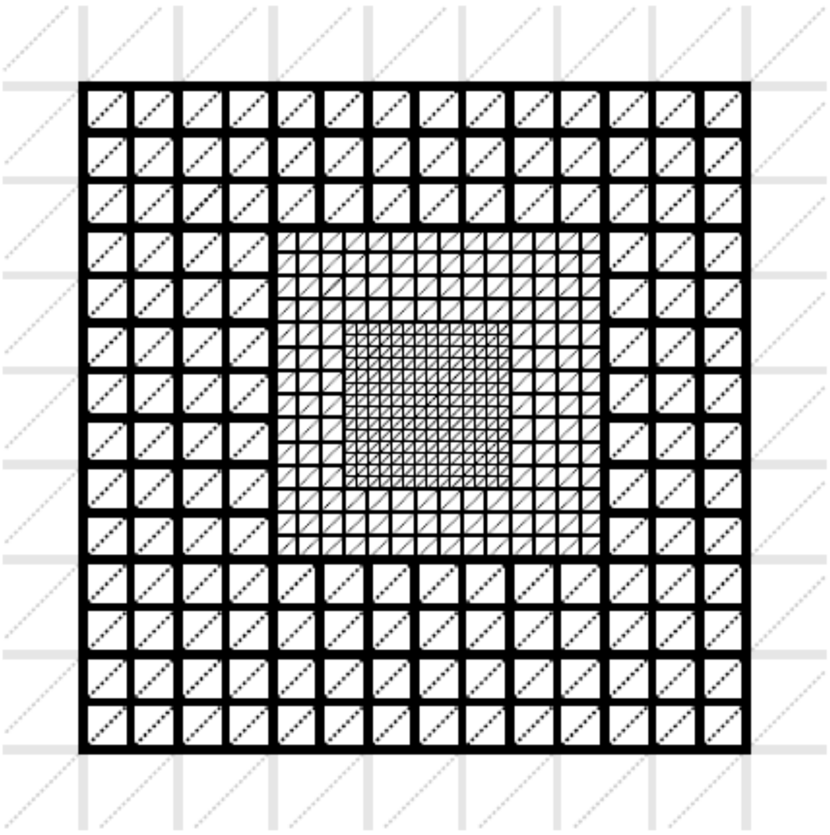
- A series of nested, regular grids (clipmap levels) are cached on the GPU
- Levels are centered around the viewer, incrementally updated with new data as the viewer moves

Structure of Geometry Clipmaps

- Level closest to the viewer has the highest detail
- Each successive level has half the detail, twice the area of the level before it
- Vertices of coarser level are coincident with vertices of finer level



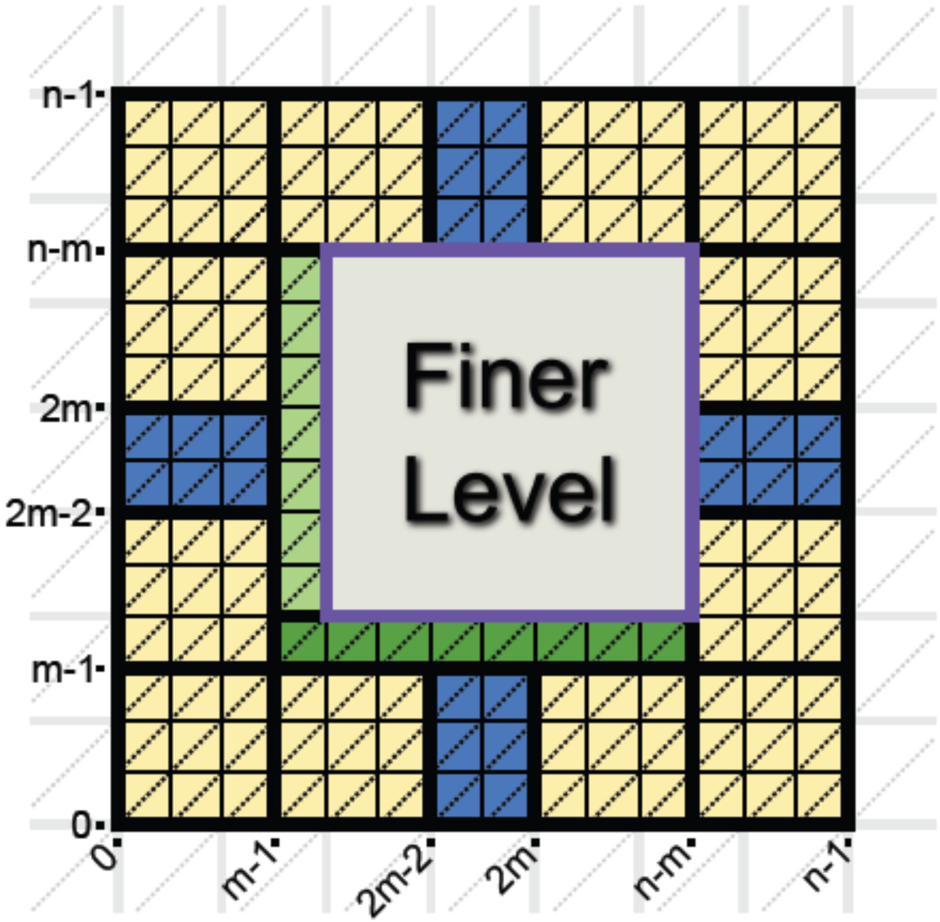


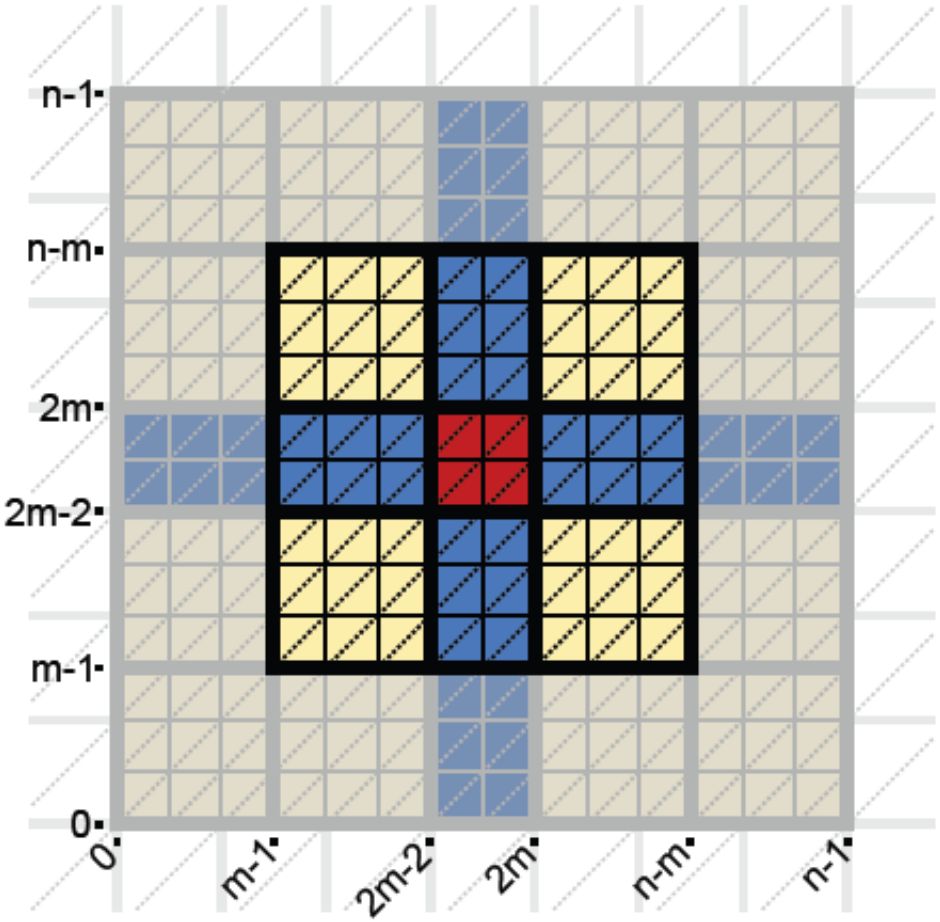
Demo

- Geometry Clipmapping

Rendering Geometry Clipmaps

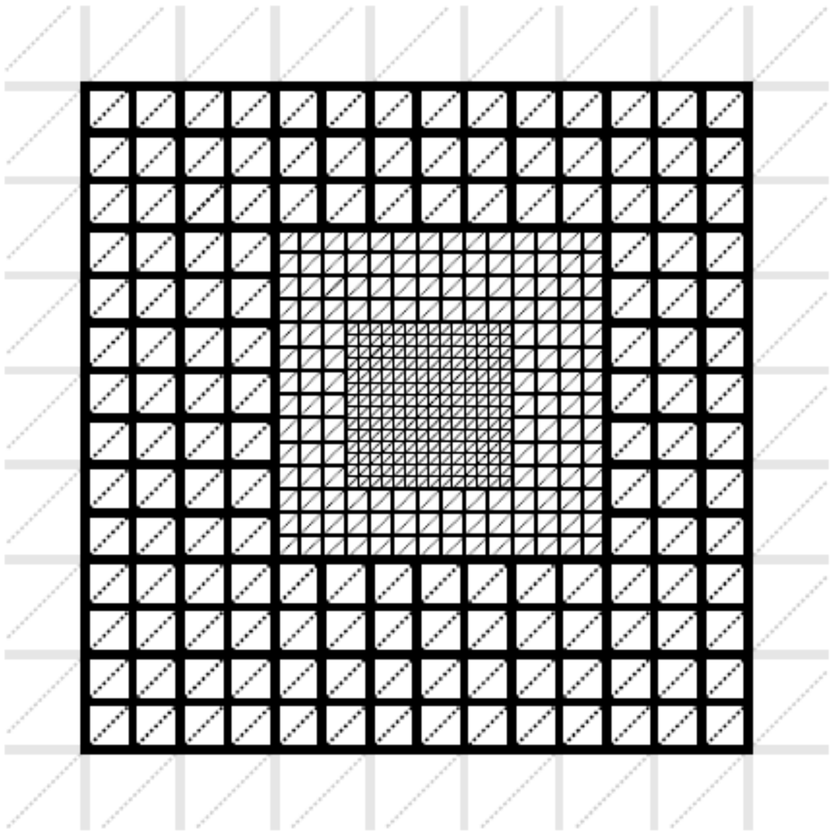
- All levels are rendered with one set of static vertex and index buffers!





Blending Between Levels

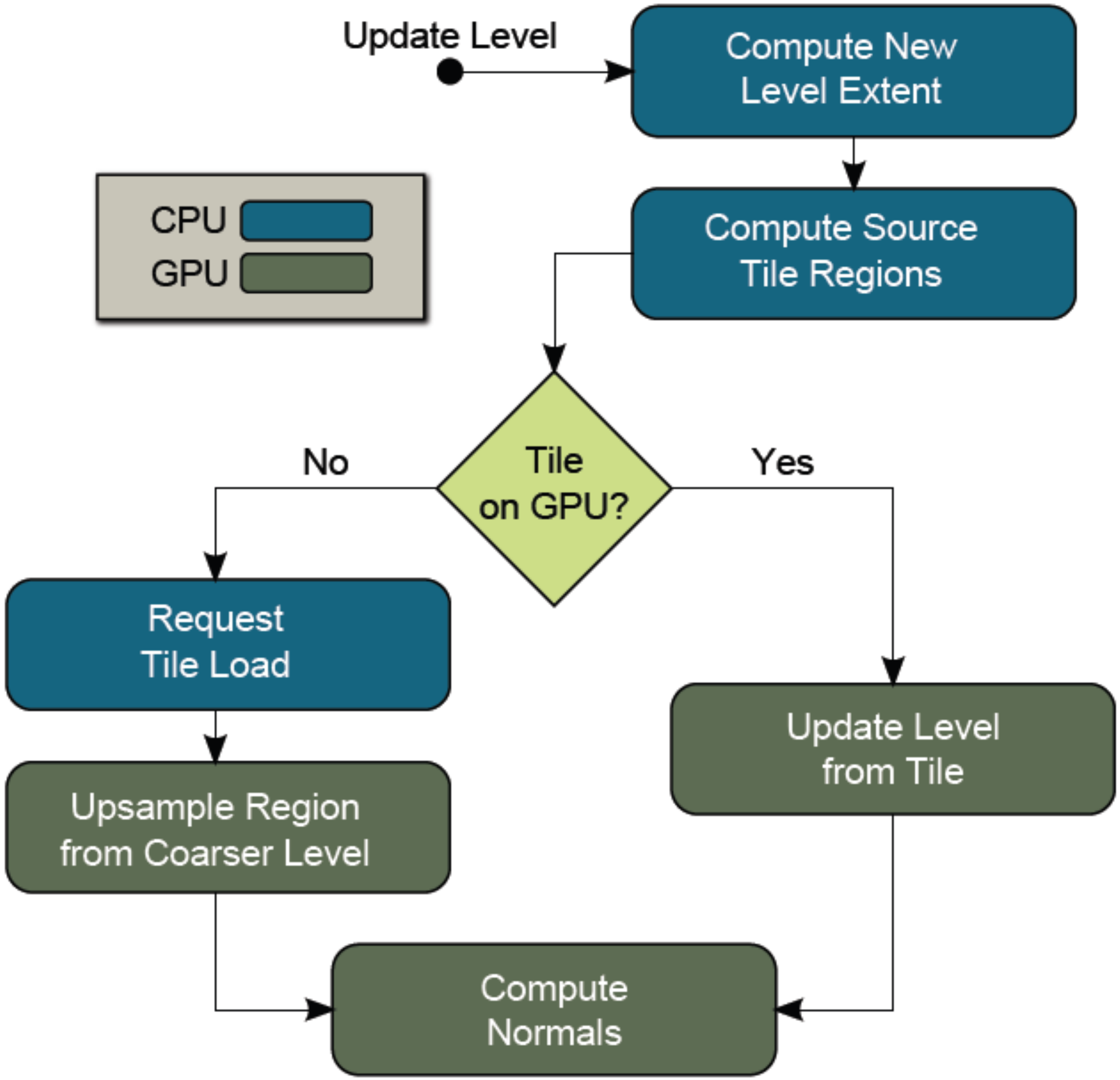
- Vertices coincident in x-y must have the same height in adjacent clipmap levels
  - watertight mesh
- But they're displaced by different mipmap levels!
- So, blend vertices near perimeter with next coarser level



$$\delta = \frac{n-1}{2} - w - 1$$

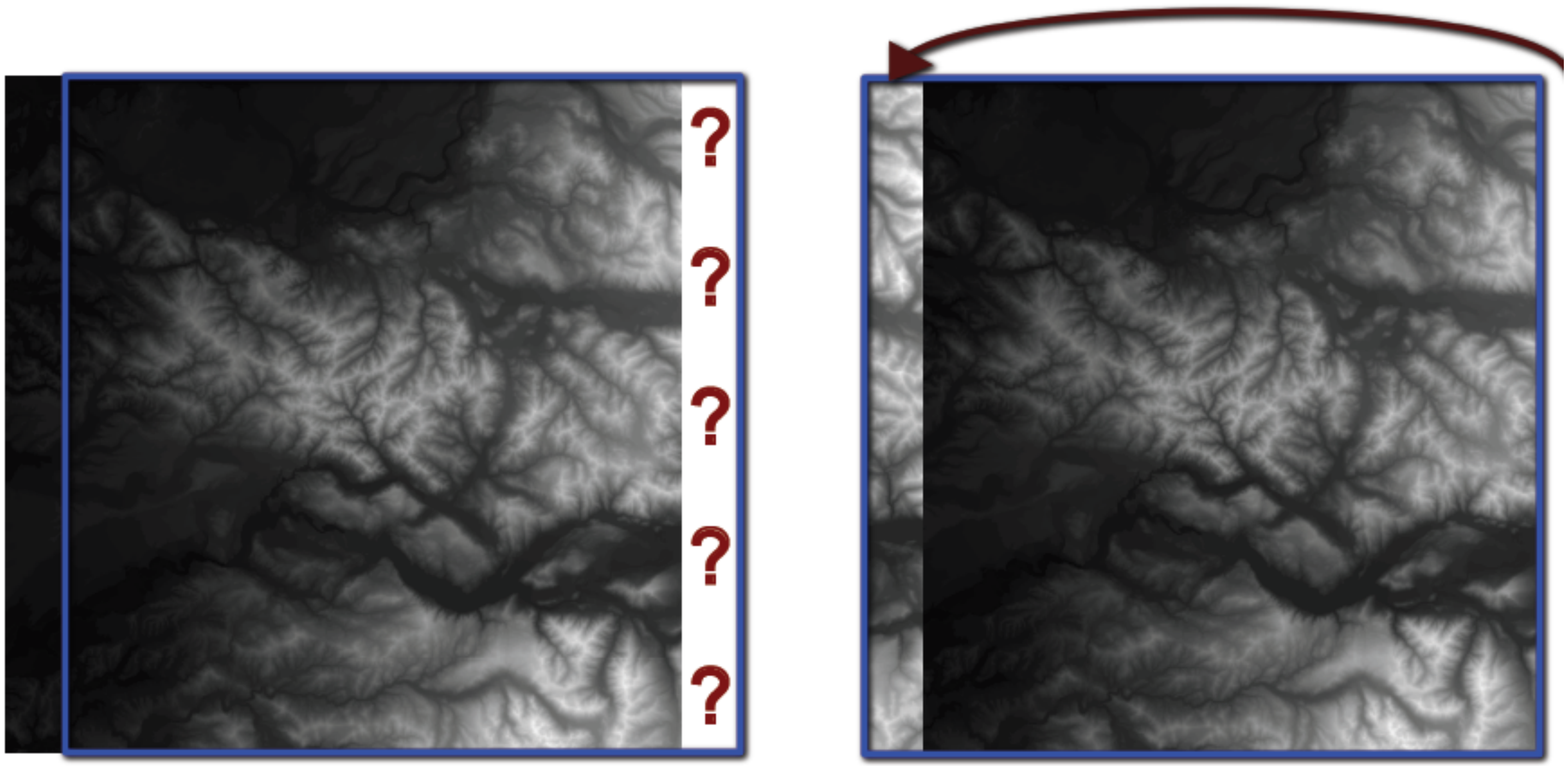
$$\alpha_x = \mathrm{clamp}\left(\frac{|x - v_x| - \delta}{w}, 0, 1\right)$$

Geometry Clipmap Update



Geometry Clipmap Update

- Clipmap levels always centered on the viewer
- Viewer moves east → entire height map shifts left
- How can we avoid rewriting the entire height texture?
  - Toroidal texture addressing



Geometry Clipmap Upsampling

The viewer moves and a new tile is needed to update a clipmap level height texture, but the tile is not in memory

What do we do?

Wait until the tile is available?
Draw the region with zero height?
Don't draw the region at all?

Upsample from coarser data!

Once the tile is loaded, the terrain improves

Wait a Minute...

How does any of this work for a globe rather than a plane?

Good question...

Geometry Clipmapping Rocks Because

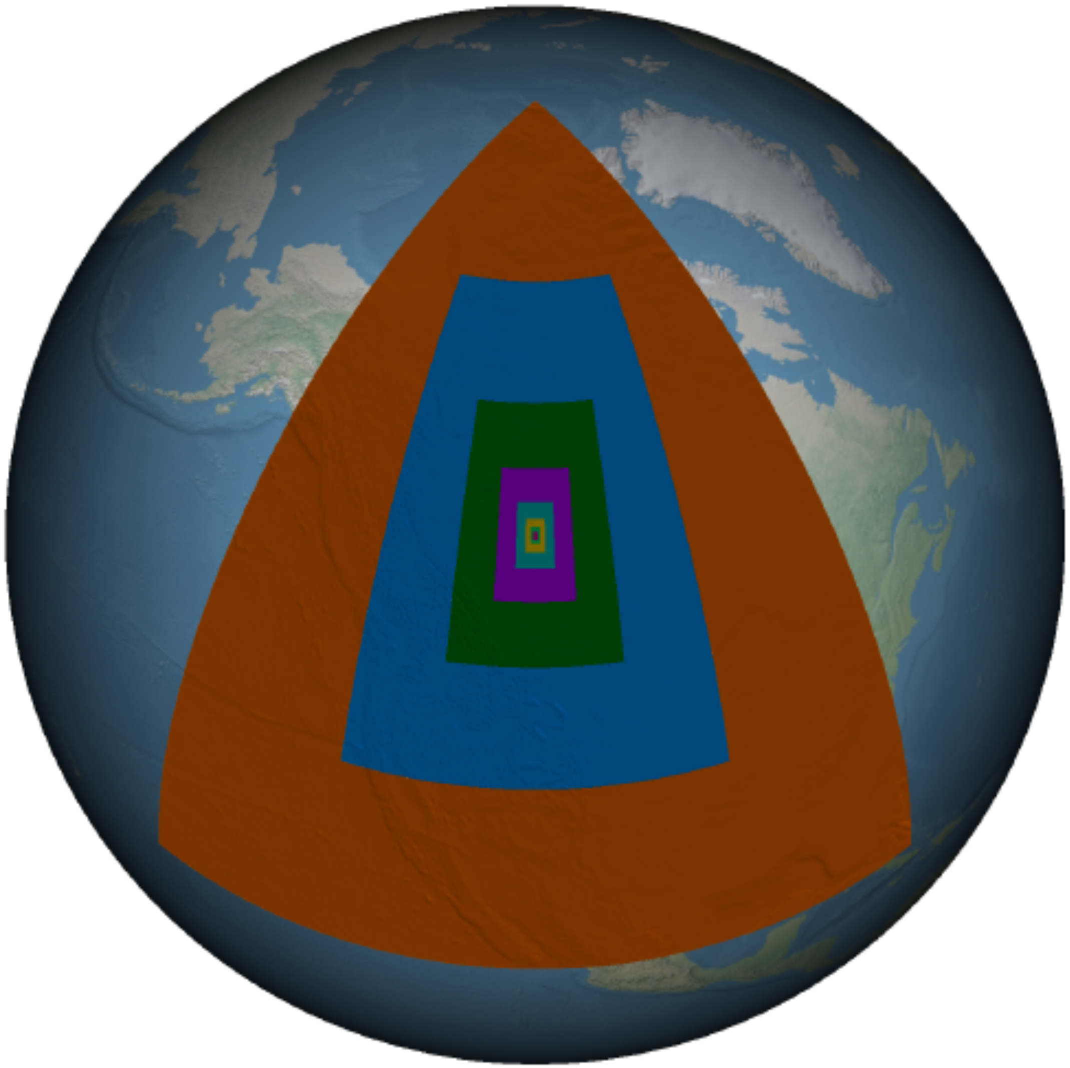
- Very little horizontal coordinate data is needed
- The geometry is independent of the viewer
  - No jittering problems
- Terrain vertices are precisely aligned with height-map texels
  - No aliasing artifacts

But accurate virtual globe rendering requires that we extrude from an ellipsoid rather than from a plane.

Is it possible to preserve these advantages?

Geometry Clipmapping on a Globe

Direct approach: transform Geodetic coordinates to Cartesian in the vertex shader

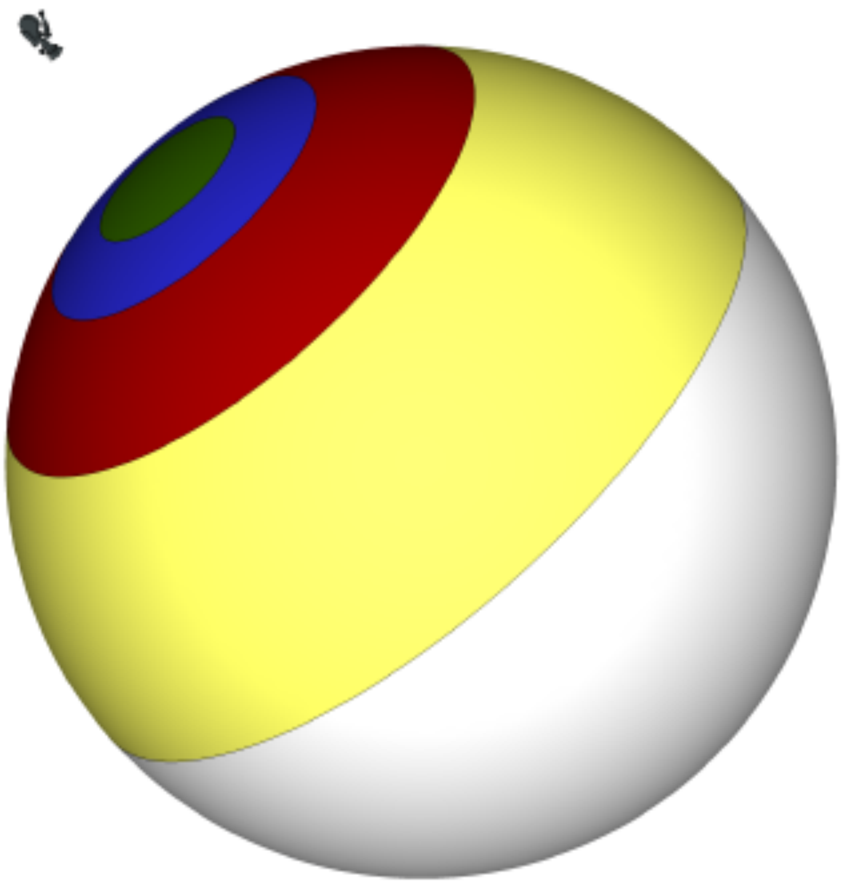


Problems with this approach:

- Precision on 32-bit GPUs
- Artifacts at the poles

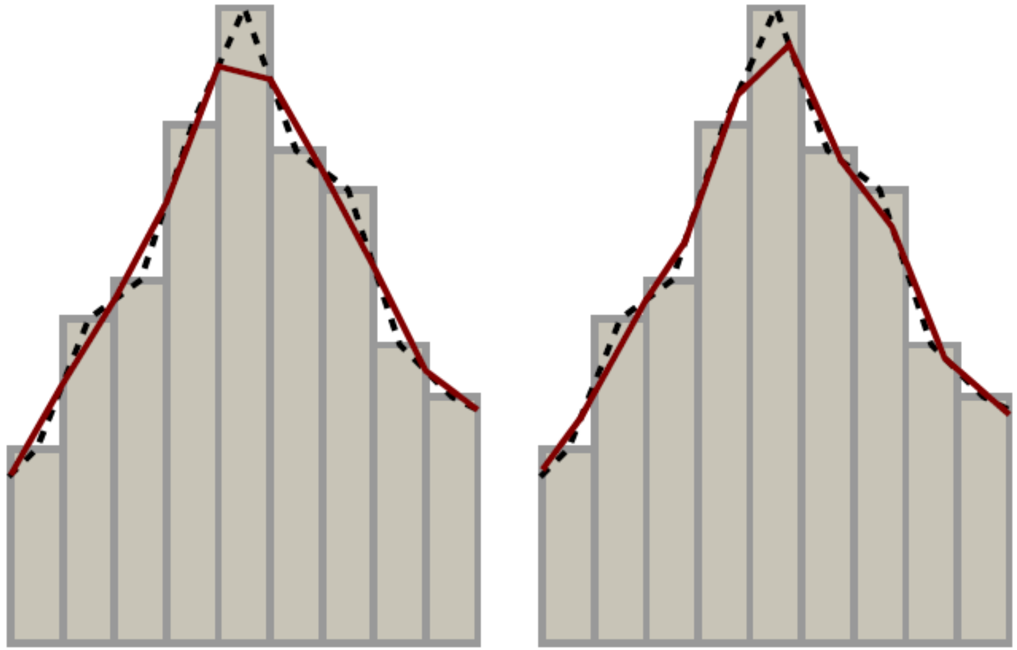
Geometry Clipmapping on a Globe

Spherical Clipmapping



Problem with this approach:

- Aliasing



Geometry Clipmapping on a Globe

Flexible approach: Instead of storing just heights in the texture, store all three position components

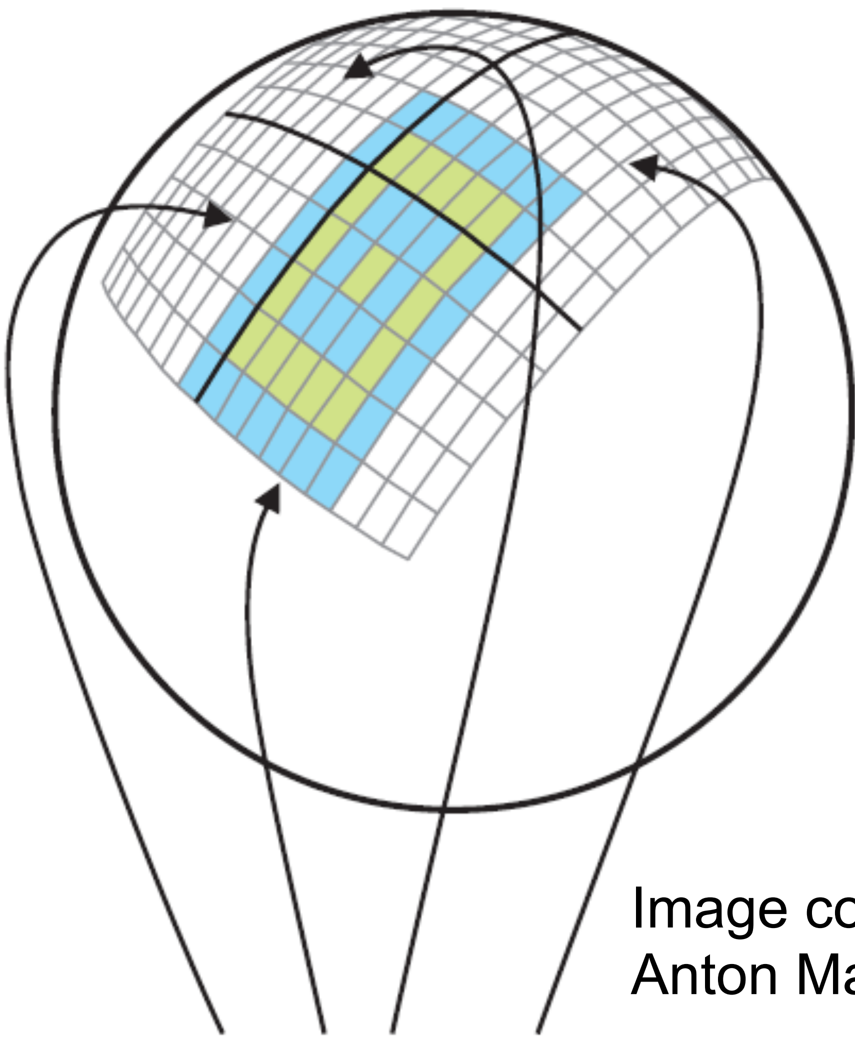


Image courtesy of Anton Malischew

Problems with this approach:

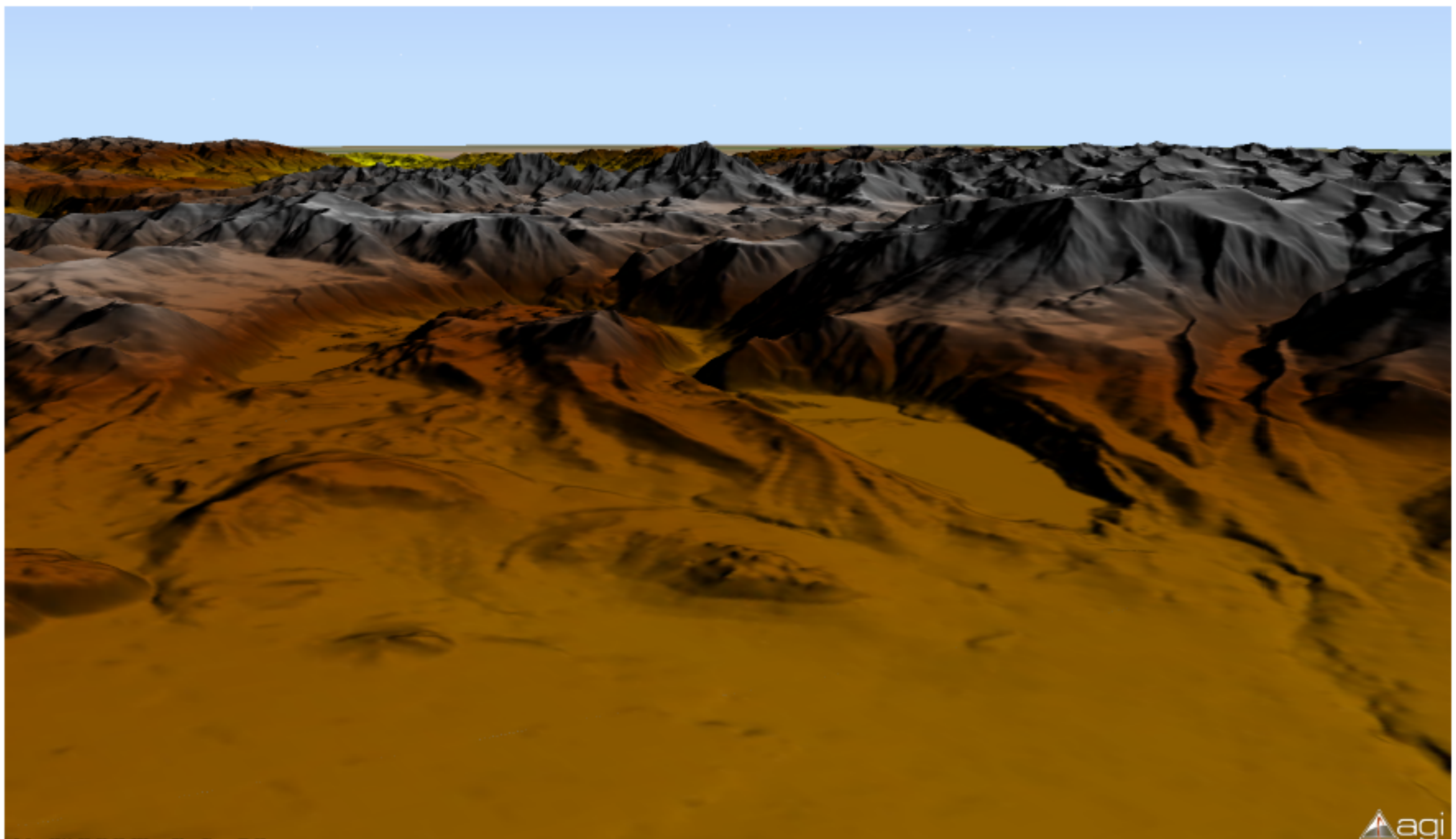
- Triple the data
- Steps necessary to avoid precision problems

Chunked LOD

See [Ulrich02]

Chunked LOD

- Terrain organized as a quadtree of chunks of terrain
- A direct application of hierarchical HLOD to terrain rendering
- Great control over the pixel accuracy of the terrain



Chunked LOD Structure

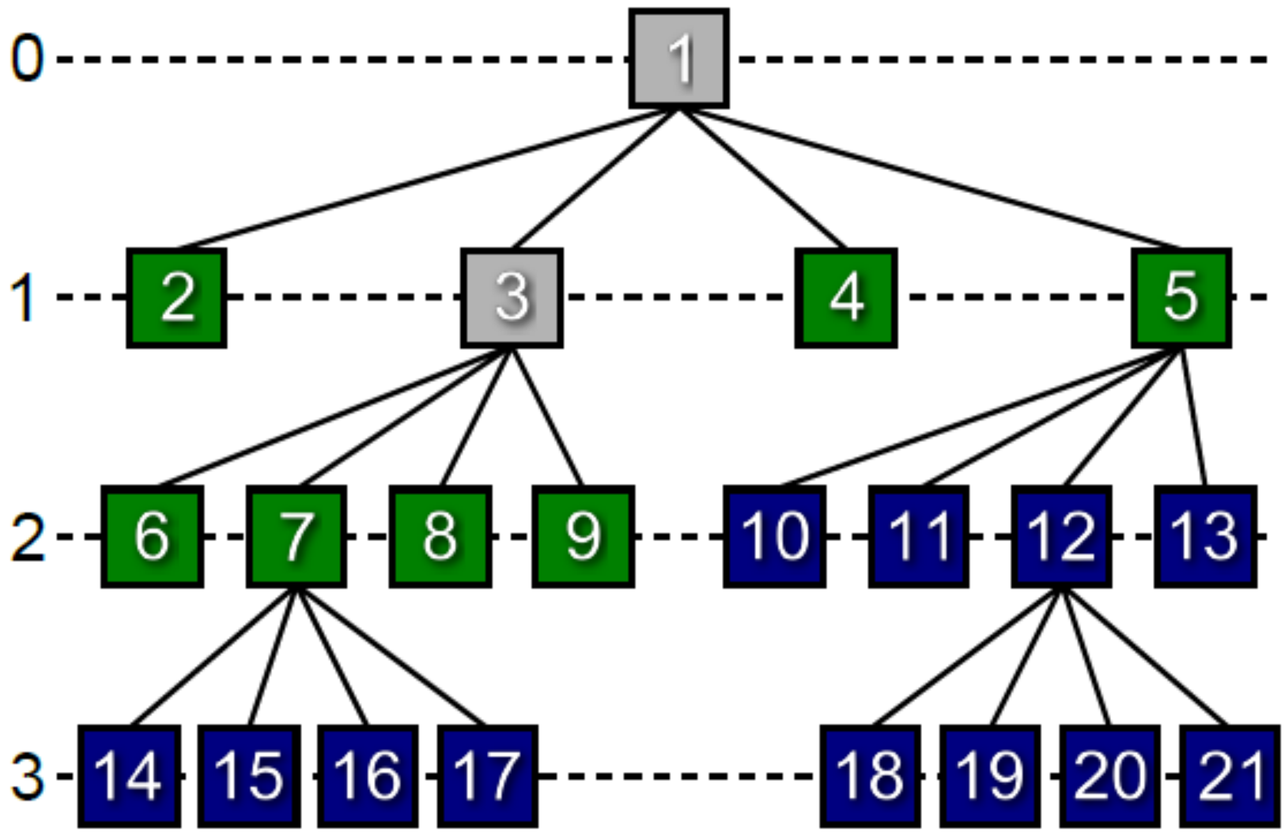
- Terrain is organized into a quadtree of chunks
- Chunks are rectangular triangle meshes
- Child chunks have half the area of their parent, and less geometric error
- Each chunk knows its geometric error relative to the original terrain data
- Each chunk knows its bounding volume

Chunk Generation

- Start with an input mesh (or height map)
- Simplify the mesh based based on the max geometric error at the most detailed level (possibly 0)
- Split the mesh into chunks based on the desired quadtree depth
- Simplify the mesh again for the next coarser level, allowing more geometric error, and divide it into one quarter as many chunks
- Continue to the root of the quadtree which has just one very simple chunk

Chunk Selection

- Each frame, traverse the quadtree depth-first
- For each chunk, compute the screen-space error
  - Under the limit ⇨ render the chunk
  - Over the limit ⇨ traverse children

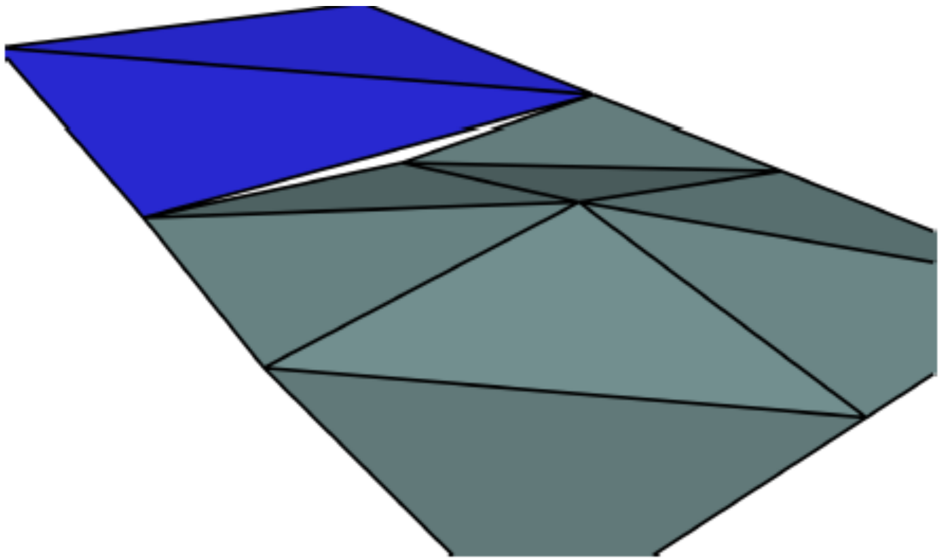


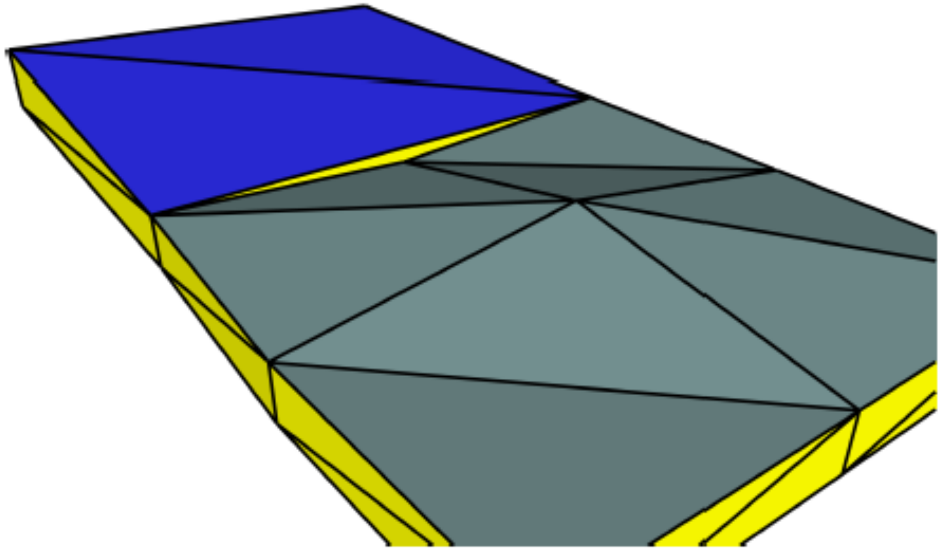
Cracks Between Chunks

Adjacent chunks can be different LODs

This leads to cracking due to the additional vertices

We fill the chunks by dropping skirts below each chunk

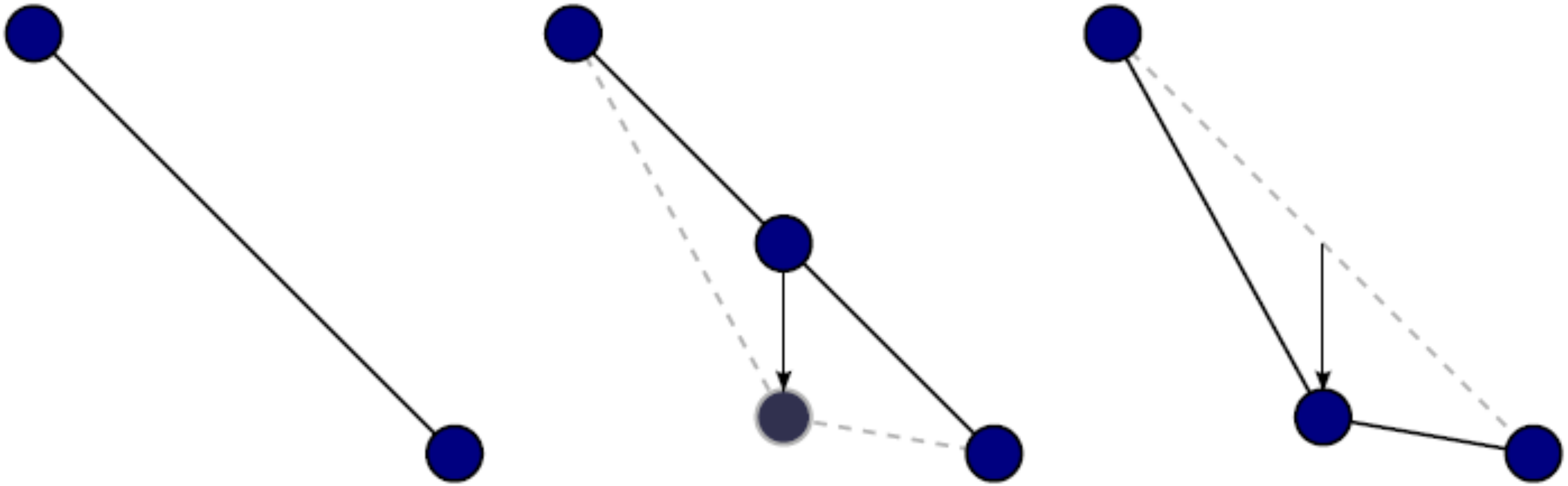




Chunk Switching

- As the viewer moves, the level of detail of the terrain will change - chunks are refined and merged.
  - Sudden "pop" from one LOD to another.
- Make transitions seamless by blending each vertex to its morph target

$$\text{morph} = \text{clamp}\left(\frac{2\rho}{\tau} - 1, 0, 1\right)$$



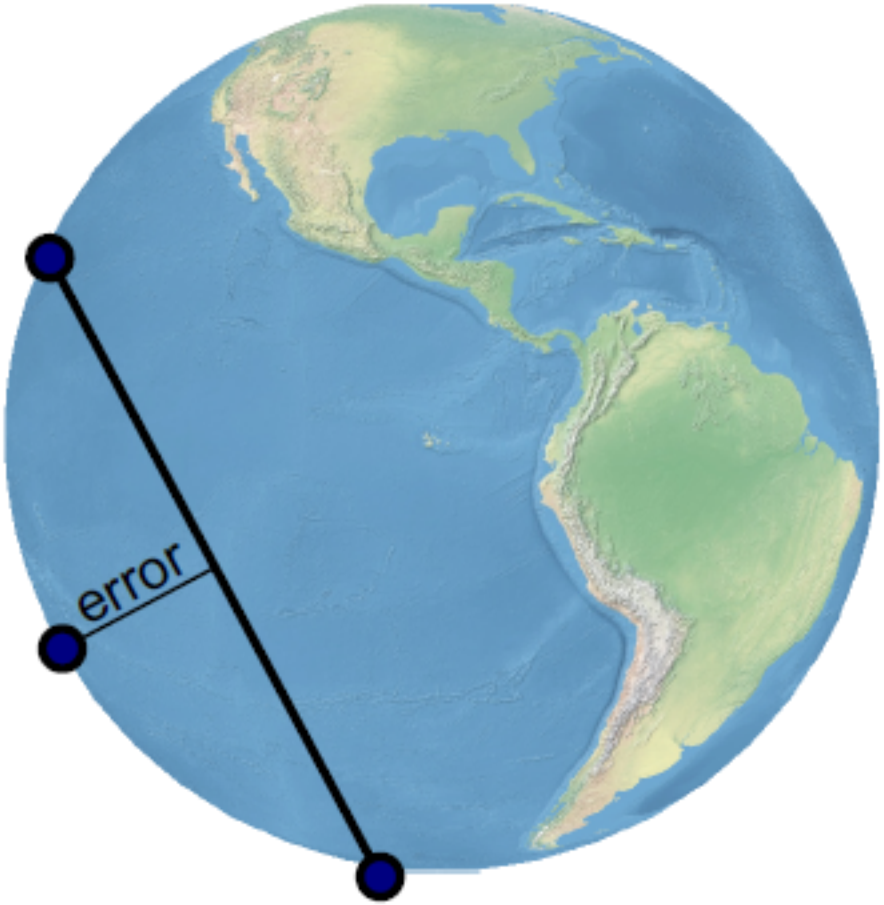
Chunked LOD on a Globe

Fewer tradeoffs than with Geometry Clipmapping.

Generation must take curved surface into account.

Vertex blending must manipulate all three components of position.

Use RTC or GPU RTE to avoid jittering problems.



Which Algorithm Should I Use?

Most virtual globes incorporate aspects of multiple algorithms to tune their terrain engines to their specific needs.

	Geometry Clipmapping	**Chunked LOD**
Preprocessing Needs	✓	
Mesh Flexibility		✓
Triangle Count		✓
Ellipsoid Mapping		✓
Error Control		✓
Frame-Rate Consistency	✓	
Mesh Continuity	✓	
Terrain Data Size	✓	
Legacy Hardware Support		✓

Selected References

[Kemen09] Brano Kemen. *Logarithmic Depth Buffer*. 2009 http://outerra.blogspot.com/2009/08/logarithmic-z-buffer.html

[Losasso04] Frank Losasso and Hugues Hoppe. *Geometry Clipmaps: Terrain Rendering Using Nested Regular Grids*. 2004 http://research.microsoft.com/en-us/um/people/hoppe/proj/geomclipmap/

[Ohlarik08] Deron Ohlarik. *Precisions, Precisions*. 2008 http://blogs.agi.com/insight3d/index.php/2008/09/03/precisions-precisions/

[Ulrich02] Thatcher Ulrich. *Rendering Massive Terrains Using Chunked Level of Detail Control*. 2002 http://tulrich.com/geekstuff/sig-notes.pdf

Images and Imagery Sources

A K Peters / CRC Press

3D Engine Design for Virtual Globes
http://www.virtualglobebook.com/

NASA Visible Earth

http://visibleearth.nasa.gov/

Natural Earth

http://www.naturalearthdata.com/

Thank You

Questions?

Send course feedback to

authors@virtualglobebook.com