LOD and Occlusion

Christian Miller
CS 354 - Fall 2011
**Problem**

- You want to render an enormous island covered in dense vegetation in realtime
Scene complexity

- Many billions of triangles
- Many gigabytes of textures
- Rendering that in realtime is hopeless
Common sense

- Observation 1: You can’t see something outside your field of view
- Observation 2: Near objects can hide far objects (occlusion)
- Observation 3: Far away objects are very tiny, and therefore require less detail
**Frustum culling**

- Only draw things that intersect with or are totally contained in the view frustum
- Pre-emptively get rid of things that will clip later
Frustum culling

- Use a spatial data structure and only draw nodes that are not outside the frustum
Time spent drawing an object is wasted if it’s just going to get covered up later.

We want to detect and avoid drawing such cases.
**Backface culling**

- Triangles on the back of objects are guaranteed to be occluded, so we simply don’t rasterize them.
- Use winding order in screen space to determine which way a triangle is facing.
- Still requires vertex transformation, but it does save lots of unnecessary pixel shading.
  - Halves the work for closed objects.
Portal culling

- A culling method that helps with indoor scenes
- Have an artist define a set of portals in the scene
  - Portals are just openings through which you can see other rooms
  - Think windows, doorways, and mirrors
Portal culling

- When rendering a room, do a frustum check on all the portals in that room
- If you can see a portal, clip the frustum to it, then render the room on the other side
  - For mirrors, flip the viewpoint first
- Repeat recursively for other rooms
Portal culling

- Frustums get smaller with each room they enter
For more general scenes, you need some way of identifying which objects are hidden behind others.
Preprocessed occlusion

- Discretize your scene into cells and compute all visible objects from each cell as a preprocess.
- At render time, just look up which cell the viewer is in and render the visible objects.
Preprocessed occlusion

- Very efficient at runtime, but has a very expensive preprocess stage
- Requires lots of storage
- Only works for static objects
Depth buffer occlusion culling

- The depth buffer has visibility information in it
- Before rendering an object, try rendering its bounding box (but don’t overwrite color or depth)
- If any pixels of the bounding box make it through the depth test, then render the object
- There’s hardware support for this on most GPUs
Hierarchical occlusion culling

- Can do one better: combine the depth buffer / bounding-box occlusion test with a spatial data structure
- Only render a node if its bounding box is visible given the current depth buffer
**Level of detail (LOD)**

- Far away objects take up very few pixels
- We should scale detail based on how large the object will end up being on the screen
- Tons of ways to do this
Discrete LOD

- Have several versions of each piece of geometry at different resolutions (called LOD levels)
- Can be artist-created or generated by a mesh simplification program
**Edge collapse**

- Merge two vertices into one
- Eliminates one edge and two triangles
- Lots of caveats to getting it to work right
Things to keep in mind
Range-based selection

- At runtime, need to figure out what LOD to use
- The simplest way is to do this is as a function of distance from the viewer
- Doesn’t take orientation into account
Projected area LOD

- Alternative to range-based selection
- Project the object’s bounding box onto the near plane and compute its area
- Look up LOD level as a function of that
Switching LOD levels

• If you transition directly from one LOD level to another, you can get a distracting popping effect
  • Sometimes noticeable even at a distance
• Must be careful to not rapidly switch back and forth between levels
Progressive meshes

- You can animate edge collapses between LODs during the transition
- Replaces popping with morphing
• Terrain is usually represented as a heightfield

• Gives an enormous mesh, can’t use just one LOD
One method: the ROAM algorithm

Represent the terrain as a simple subdivision surface, splitting the triangles in two at each level
**Picking tile LOD**

- At runtime, take a look at the current screen-space error of each triangle tile
- If it’s too high, increase resolution
- If it’s low enough, decrease resolution
Mind the gap

- Adjacent tiles at different levels of detail will develop cracks
- Must alter subdivision to match up with neighbors
Another idea: static objects that are far away don’t change very much as the viewer moves around.

Render it to a texture, then just display it as a quad.

Update the imposter when viewer moves too much.
Deferred shading

- Draw diffuse color, normals, depth, specular coefficient, etc. into separate buffers
  - Each one of these has normal depth test applied, but no shading happens yet
- Take one pass at the end and apply the lighting shader to everything at once
  - Guaranteed to shade each pixel only once
**FIGURES COURTESY...**

- Real-Time Rendering, 3rd ed. [RTR]
  - Tomas Akenine-Moller, Eric Haines, Naty Hoffman
  - Eric Lengyel
  - Edward Angel, Dave Shreiner
- Wikipedia [WP]
- Hugues Hoppe, “Progressive Meshes” SIGGRAPH 1996 [HH]
- David Luebke and Chris Georges, “Portals and Mirrors”, I3D 1995 [DL]