Rendering Large Environments

(in real time)

Siddhartha Chaudhuri

CS475/675, Fall 2016
15 polygons
~850 polygons

Half Life, 1998
\(~12,000\) polygons
~500,000 polygons
~3,000,000 polygons
The fastest modern GPUs manage a few million shaded polygons* at 30fps

* estimates highly test-dependent
How do we bridge the gap?
Largescale Rendering Cheat Sheet

- Don't render what you can't see
Largescale Rendering Cheat Sheet

- Don't render what you can't see
- **Don't render what the display can't resolve**
Largescale Rendering Cheat Sheet

- Don't render what you can't see
- Don't render what the display can't resolve
- People won't notice small errors, especially in background objects
Largescale Rendering Cheat Sheet

- Don't render what you can't see

- Don't render what the display can't resolve

- People won't notice small errors, especially in background objects

- If all else fails, fog is your best friend :)}
Largescale Rendering Cheat Sheet

• Don't render what you can't see

• Don't render what the display can't resolve

• People won't notice small errors, especially in background objects

• If all else fails, fog is your best friend :)
Don't render what you can't see

- Rasterizing invisible objects is wasteful
- Detect such objects early and ignore (cull) them
Difficulty

Backface < Frustum <<< Occlusion
Backface Culling

- Drop faces on the far side of object meshes
  - Assume face normals consistently point **inside-out**
  - Back faces have **normals pointing away** from the camera
- OpenGL:
  
  ```
  glEnable(GL_CULL_FACE);
  glCullFace(GL_BACK);
  ```
  
- Why would anyone want `glCullFace(GL_FRONT)`?
  - Uses **vertex winding order** to determine front and back faces, not the vertex normals you pass in!
- When does this scheme for back face culling fail?
Frustum Culling

- Test each object against the view frustum
  - Much faster: **test the bounding box** instead
    - If object is visible, no frustum plane can have all 8 corners on invisible side

- Optimization:
  - **Group objects hierarchically**
    - *Octree* (or *quadtree* for 2.5D scenes)
    - *Binary Space Partitioning (BSP) tree* (or a restricted version called a *kd-tree*)
    - *Bounding box/sphere hierarchy*
  - Traverse tree top-down and ignore subtrees whose roots fail the bounding box test
Binary Space Partitioning

- Choose a plane to split the objects
- Recurse
- Methods differ in how splitting plane is chosen
  - Standard BSP-tree: plane of some scene polygon
  - KD-tree: axis-aligned
  - Tries to balance number of objects on each side
Occlusion Culling

- Whole subject by itself

- **General idea:** Ignore background objects covered *(occluded)* by foreground objects
  - *(Hardware)* Occlusion queries
  - From-region visibility
  - Portal-based rendering
Hardware Occlusion Queries

- Part of OpenGL/D3D API
- At any time, pretend to draw a dummy shape (say the bounding box of a complex object) and check if any pixels are affected
- Accelerated by hierarchical z-buffer
- Works for dynamic scenes

Unoccluded pixels of bounding box, so object is potentially visible
From-Region Visibility

- **Preprocessing:**
  - Break scene up into regions
  - For each region, compute a *potentially visible set (PVS)* of objects

- **Runtime:**
  - Detect the region containing the observer
  - Render the objects in the corresponding PVS

- PVS is usually quite conservative, so further culling is needed
Case Study: Quake

- **Preprocessing:**
  - Level map preprocessed into BSP-tree
  - Each leaf node stores potentially visible polygons from that region

- **Runtime:**
  - Leaf node containing player detected by searching the tree (very fast)
  - PVS of polygons for this node are rendered
  - (BSP-tree is NOT used for back-to-front rendering!)
Portal-Based Rendering

• Suitable for indoor environments
• Divide environment into cells, connected by simple polygonal portals (doors/windows/...)
• Render:
  • Neighboring cells with visible portals (check if projected polygon is within screen limits)
  • Neighbors-of-neighbors with portals visible through the first set of portals
  • ... and so on
• Further culling possible with frusta through portals
Don't render what the display can't resolve/people won't notice

- There're only ~1 million pixels on the average screen
- Why spend precious milliseconds rendering the Taj Mahal in exquisite detail if it's only going to take up 10 pixels on the screen?

- **Moral:** Simplify distant objects
  ... assuming perspective projection
- **Moral #2:** Since people are usually not too interested in the background, you can simplify over-aggressively
Guiding Principle

For every object, choose the **simplest possible representation** that will look nearly the same as the original *when rendered at the current distance*
Levels of Detail (LOD)

- **Coarser representations for distant objects**
  - Hierarchy of representations of the same object at different resolutions
- The same idea can also be used for textures (**mipmapping**)

![Rabbit models at different LODs]

<table>
<thead>
<tr>
<th>Resolution</th>
<th>Polys</th>
</tr>
</thead>
<tbody>
<tr>
<td>69,451</td>
<td>2,502</td>
</tr>
<tr>
<td>2,502</td>
<td>251</td>
</tr>
<tr>
<td>251</td>
<td>76</td>
</tr>
</tbody>
</table>
Levels of Detail (LOD)
Terrain LODs
Environment Maps

- Very distant stuff looks the same from anywhere within reasonable limits
- Pre-render distant objects (including the sky) out to a 360° image
- Texture-map it onto a bounding cube at runtime
Image-Based Rendering

- Render complex objects to images and texture-map them to simple proxy shapes (*impostors*)
  - Environment mapping is a specific example
- *Billboards/sprites*: Textured quads always facing the viewer
  - Single image is valid if viewer doesn't move much
Image-Based Rendering

Tree decomposed into a cloud of texture-mapped planar slices

Décoret, Sillion, Durand and Dorsey 2002
Adding Depth to Images

- Store the **depth map** as well as the color
- Impostor is **heightfield** defined by the depth map
- Fixes parallax errors (impostor is still valid when viewing position changes significantly)
- What are the drawbacks?
Images + Geometry

= 

Foreground + Background
Images + Geometry

= 

Foreground + Background Impostor
Images + Geometry (Rendered View)

= 

Foreground + Background Impostor
Another Example

Geometry-to-Image Transition
LOD spectrum
(not exhaustive or exact!)

Distance from viewer

Original geometry
Coarser geometrical LODs
Impostors
Environment maps
Speed Gains (back of the envelope)

• Backface culling: \( \sim 2x \)
• Frustum culling: \( \sim 5x \) (varies inversely with FOV)
• Occlusion culling: can be huge, but no guarantees at all except in special cases, e.g. portal-based indoor environments
• Levels-of-detail: a uniformly dense scene of radius \( r \) takes \( O(\log r) \) time to render (prove!)

Lesson of the day: invest in good LODs!!!