Culling

CS535 Fall 2010

Daniel G. Aliaga
Department of Computer Science
Purdue University
Types of culling

• View frustum culling
• Visibility culling
  – e.g., backface culling, portal culling, textured-depth mesh culling
• Occlusion culling
  – e.g., in hardware, hierarchical occlusion maps, hierarchical z-buffer
Portal Culling

• Increase interactive rendering performance of architectural walkthroughs
  – Cull away non-visible “areas”
  – Replace geometry behind portals (doorways and windows) with images warped to the current eye position
  – Reduce the rendering complexity with a small number of image samples
Cells and Portals
Portal Images
Portal Image Warping

McMillan & Bishop Warping Equation:

\[ x_2 = \delta(x_1) P_{2}^{-1} (c_1 - c_2) + P_{2}^{-1} P_{1} x_1 \]

 Move pixels based on distance to eye

 ~Texture mapping

• Per-pixel distance values are used to warp pixels to their correct location for the current eye position
Portal Image Warping

- Projecting the eye onto the reference image partitions it into 1, 2 or 4 sheets.
- A raster scan of each sheet produces a back-to-front ordering of warped pixels.
Creating Portal Images

Ideal portal image would be one sampled from the current eye position.
Creating Portal Images

Reference COPs

A large number of static reference images (~120) needed to eliminate *popping*
Creating Portal Images

With warping we use a much smaller number of reference images
Creating Portal Images

Choose closest reference image and warp to current eye position

Reference COPs

eye

portal
Problem: Exposure Events

We observe *tears* in regions with unknown visibility data.
Reducing Exposure Events

To address this we:

1. Render multiple reference images
2. Allow warped pixels to persist and fill-in minor tears
Parallelization: Sheets

Eye view direction similar to reference COP view direction

Since the projection plane is perpendicular to the view direction, we usually have 4 sheets

We warp each sheet on a different processor
Sample Path: 1 Image/Portal

![Graph showing rendering time over frames for Portal Culling only and Portal Image with frame numbers from 0 to 500 and rendering times from 0 to 0.5 seconds.]

- Red line: Portal Culling only
- Blue line: Portal Image

Frame

Rendering Time (sec.)
Sample Path: 2 Images/Portal

- Portal Culling only
- Portal Image

Rendering Time (sec.)

Frame
Textured-Depth Mesh Culling

- Model Space Partitioning
  - Preference to important areas
- Far Geometry
  - Replace with textured depth meshes
- Near Geometry
  - Could use LOD, occlusion and view-frustum culling
Model Space Partitioning

• View Preference Function (VPF)
  – Similar to progressive rendering, allocate resources (e.g. time, storage, CPU, bandwidth) to most important areas first
Model Space Partitioning

- Subdivide important areas into *cells*
Model Space Partitioning

- Associate a *cull box* with each cell
Model Space Partitioning

- Cull box divides near and far geometry
- Replace far geometry with Textured Depth Meshes (TDMs)
Far Geometry

• Replace far geometry with Textured Depth Meshes (TDMs)
Far Geometry

• Replace far geometry with Textured Depth Meshes (TDMs)
Far Geometry

- Replace far geometry with Textured Depth Meshes (TDMs)
Textured Depth Mesh

- Image
- Depth Image
- Simplify
- Texture
- Mesh
- Rendering
- TDM

Preprocessing
Run-time
Example TDM

- Foreground
  - Geometry
- Background
  - TDM with mesh outlined in white
TDM Skins

- Example of a simple TDM
TDM Skins

- Example of a simple TDM
TDM Skins

• Disocclusions appear as *skins*
Near Geometry

- Cull to view frustum
- Select LOD based on distance
Near Geometry

- If sufficiently complex, do occlusion culling
  - e.g., Hierarchical Occlusion Maps
What size cull box?

• Given a fixed primitive budget and a fixed set of cells:
  – Large cull boxes require coarse LODs
  – Small cull boxes exaggerate skins in the TDMs
  – (visibility culling does not alter the visual content)
Balancing Near-Far Geometry

• Error metrics
  – LOD: deviation in pixels from original surface
  – TDM: maximum skin size, in screen pixels
• Reduce and balance the combined error
  – Adjust cull box size and LODs to meet budget and balance errors
Power Plant Frame Rates

![Graph showing Power Plant Frame Rates]

- VFC Only
- Cold Cache
- Warm Cache
Questions

Order of the acceleration techniques affects the results, so which is best?

<table>
<thead>
<tr>
<th>(a)</th>
<th>(b)</th>
<th>(c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cull Box</td>
<td>Cull Box</td>
<td>VF Culling</td>
</tr>
<tr>
<td>VF Culling</td>
<td>LOD</td>
<td>Occl. Culling</td>
</tr>
<tr>
<td>LOD</td>
<td>VF Culling</td>
<td>Cull Box</td>
</tr>
<tr>
<td>Occl. Culling</td>
<td>Occl. Culling</td>
<td>LOD</td>
</tr>
</tbody>
</table>
Occlusion Culling

• As opposed to visibility culling, in occlusion culling the algorithm determines what is occluded (instead of what is visible)

• This leads to the confounding, but true, equation:

  “not visible” ≠ “occluded”
Occlusion Culling in Hardware

- Occlusion Query (e.g., nv_occlusion_query)
  - Performed at the polygon level
  - Tells the caller whether any pixels (or maybe how many) of a rendered object/polygon actually became visible
  - Thus,
    - can render a “box” instead of a complex object
    - if the entire box is occluded, then complex object is occluded
  - Result is potentially faster rendering performance
Occlusion Culling in Hardware

- Early Z rejection
  - Performed at the pixel level
  - If a fragment is not visible (because the depth test failed), rejection occurs immediately
  - Can save unnecessary texture fetching and execution of GPU level code
  - Result is a saving of memory bandwidth
Hierarchical Occlusion Maps

Occluded geometry
Visible geometry
Occluder geometry

(all images from Hansong Zhang’s dissertation)
Hierarchical Occlusion Maps

- Motivation
  - Occlusion is cumulative
  - Occluders can be “fused”
Occlusion Culling Algorithm

- **Progressive Culling**

  Initialize occlusion representation ($\text{OR}$) to empty;
  
  For each object
  
  Perform occlusion test(s) against $\text{OR}$;
  
  If occluded
  
  Discard object (cull);
  
  else
  
  Render object;
  
  Update $\text{OR}$;

- **Con?**
  - Must traverse in front-to-back order to get best use of culling
Occlusion Culling Algorithm

- Progressive Culling

  - **OR**: the occlusion representation
  - **PO**: the set of potential occluders to be accumulated into OR

  Initialize **OR** to empty;
  Initialize **PO** to empty;

  For each object
    Perform occlusion test(s) against **OR**;
    If occluded (culled)
      Discard object;
    else
      Render object;
      Add object to **PO**;
      If **PO** is large enough
        Update **OR** with objects in **PO**;
        Reset **PO** to empty;

- Con?
  - Multiple passes (could be many...)
Occlusion Culling Algorithm

• Multi-pass occlusion culling

\textbf{OR}: the occlusion representation
\textbf{PO}: the set of potential occluders to be accumulated into \textbf{OR}

Initialize \textbf{OR} to empty;
Initialize \textbf{PO} to empty;

\textit{For each object}
\begin{itemize}
  \item Perform occlusion test(s) against \textbf{OR};
  \item If occluded (culled)
    \begin{itemize}
      \item Discard object;
    \end{itemize}
  \item else
    \begin{itemize}
      \item Render object;
      \item Add object to \textbf{PO};
      \item If \textbf{PO} is large enough
        \begin{itemize}
          \item Update \textbf{OR} with objects in \textbf{PO};
          \item Reset \textbf{PO} to empty;
        \end{itemize}
    \end{itemize}
\end{itemize}
A Definition of Occlusion

\[ \rho_R = \frac{1}{A_R} \int \int_R \rho(x, y) \, dx \, dy \]

\[ \rho(x, y) = \begin{cases} 
1 & \text{if } (x, y) \in R_o \\
0 & \text{if } (x, y) \in R - R_o 
\end{cases} \]

where \( R \) and \( R_o \) are opaque and not opaque regions.

Rendered Image  Occlusion Map
Hierarchical Occlusion Map (HOM)

• The occlusion images can be readily generated by hardware; how?
  – Just render the scene with white=visible and black=background

• Note that this process is at some selected resolution
  – So let's change the resolution!
Hierarchical Occlusion Map (HOM)

• And we get this:
Hierarchical Occlusion Map (HOM)

- And we get this:
Hierarchical Occlusion Map (HOM)

- Creation algorithm for HOMs:

1. **Framebuffer (Originally the rendered image of the occluder scene)**
2. **Copy framebuffer to the texture memory**
3. **Render square half of the current size with proper texture coordinates**
4. **Read framebuffer for occlusion map**
Hierarchical Occlusion Map (HOM)

- How do you use an HOM?
  - First cull using the coarsest HOM
  - Then the next coarsest...
  - Until the finest level of the HOM
  - If something is visible in all levels, then it is visible
  - So what is the gain?
Hierarchical Occlusion Map (HOM)

• Observation
  – Even at the finest level, it is not known if the object is “truly” visible
  – But at the resolution you care about, it is…
Hierarchical Occlusion Map (HOM)

• Conclusion
  – The hierarchy is conservative
  – Generate from the coarest HOM to the finest level you care about
  – Only use those and select at run-time depending on performance
Example Results
Some questions?

• For a 2x2 grid of pixels in the occlusion map, what occlusion value do you choose?
  – Min?
  – Max?
  – Average?

• Welcome to hierarchical z-buffers!