GPU-based Scene Management for Rendering Large Crowds

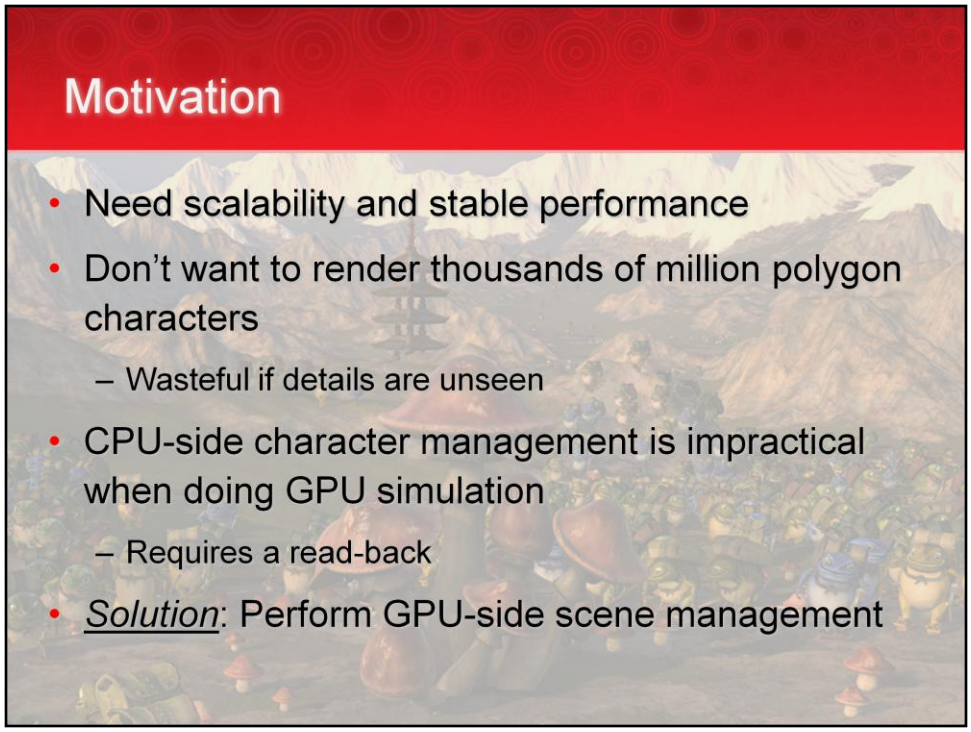
Joshua Barczak, Natalya Tatarchuk,
Christoper Oat

Outline

- Motivation
- GPU Crowds
  - Management
  - Rendering
- Conclusion

Motivation

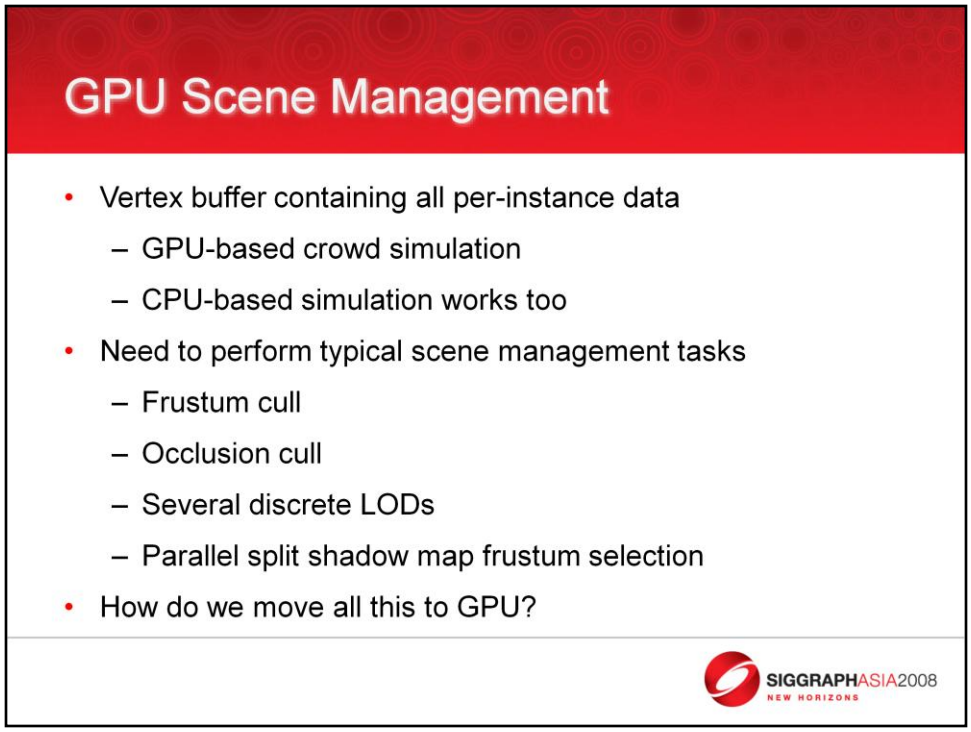
Many rendering scenarios, such as battle scenes or urban environments, require rendering of large numbers of autonomous characters. Crowd rendering in large environments presents a number of challenges, including visibility culling, animation, and level of detail (LOD) management.



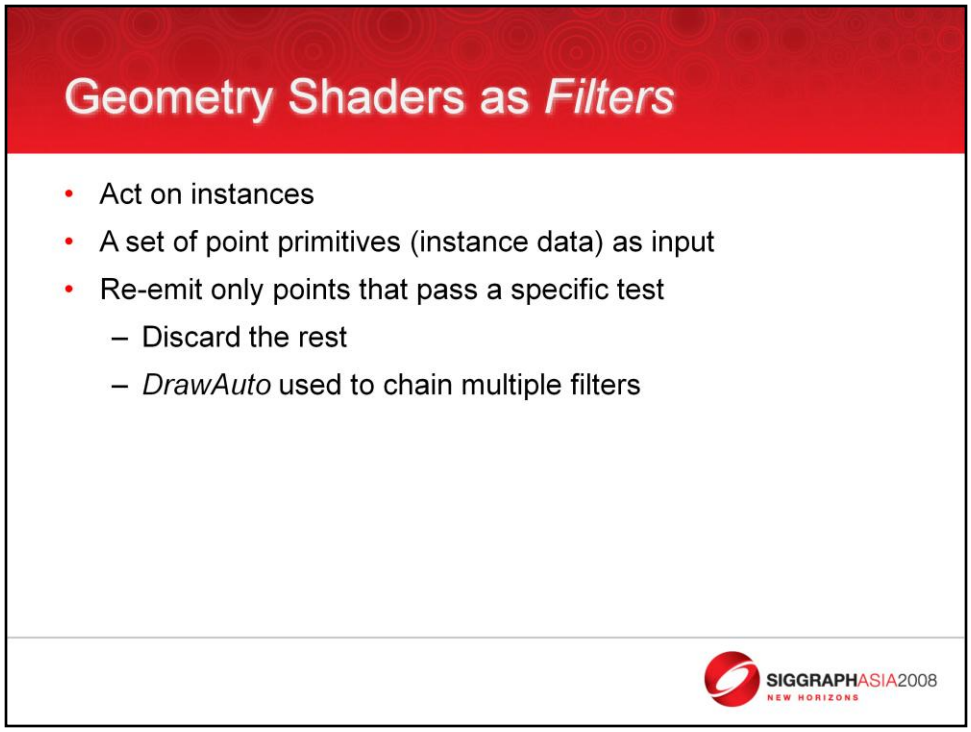
These have traditionally been CPU-based tasks, trading some extra CPU work for a larger reduction in the GPU load, but the per-character cost can be a serious bottleneck. Furthermore, CPU-side scene management is difficult if objects are simulated and animated on the GPU. We present a practical solution that allows rendering of large crowds of characters, from a variety of viewpoints, with stable performance and excellent visual quality. Our system uses Direct3D 10 functionality to perform view-frustum culling, occlusion culling, and LOD selection entirely on the GPU, allowing thousands

of GPU-simulated characters to be rendered with full shadows in arbitrary environments. To our knowledge this is the first system presented that supports this functionality.

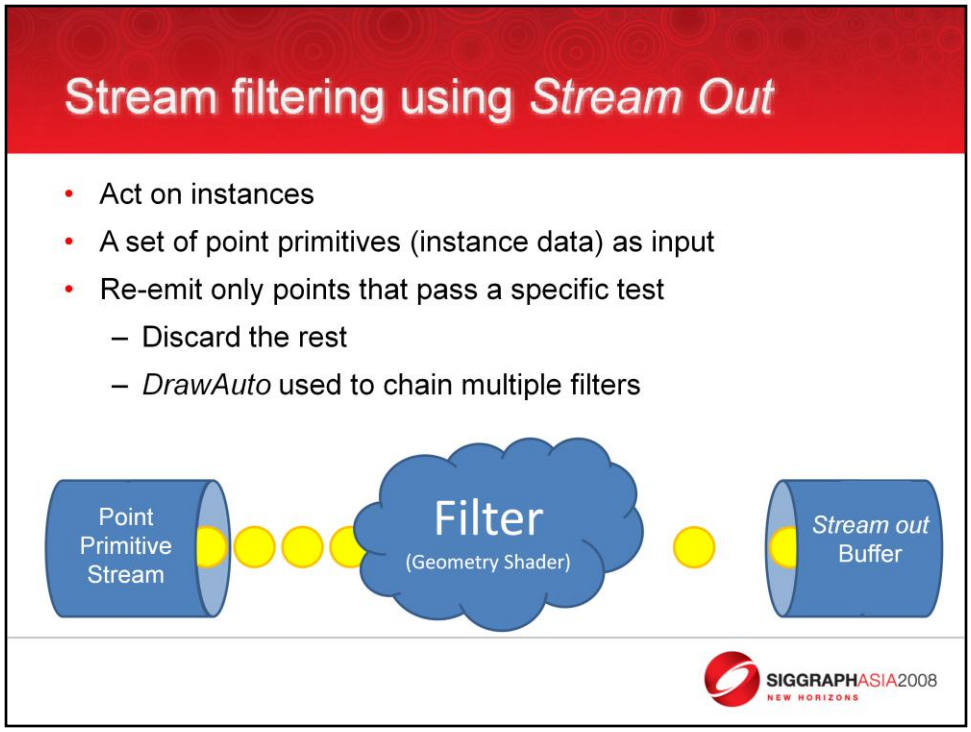
Scene Management



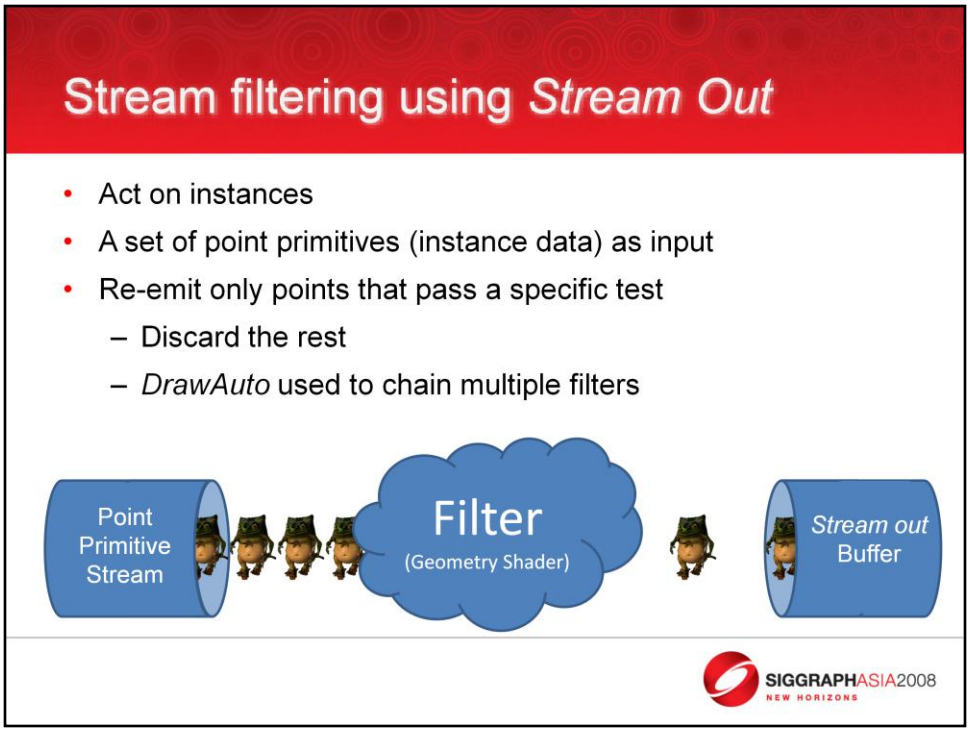
We start with a vertex buffer containing all of the per-instance data needed to render each character, such as character positions and orientations. In our case, this information is obtained from a GPU-based crowd simulation, but a CPU-based simulation or user input could also be used. In order to avoid a read-back, we wanted to perform all the typical scene management tasks on the GPU. In particular we wanted to do frustum and occlusion culling of the characters, sort the visible characters into discrete LOD groups as well as shadow frustum groups.



The key idea behind our scene management approach is the use of geometry shaders that act as *filters* for a set of character instances.



A filtering shader works by taking a set of point primitives as input, where each point contains the per-instance data needed to render a given character (position, orientation, and animation state).



The filtering shader re-emits only those points which pass a particular test, while discarding the rest. The emitted points are streamed into a buffer which can then be re-bound as instance data and used to render the characters. Multiple filtering passes can be chained together by using successive *DrawAuto* calls, and different tests can be set up simply by using different shaders.

In practice, we use a shared geometry shader to perform the actual filtering, and perform the different filtering tests in vertex shaders. Aside from providing more modular code, this approach can also provide performance benefits.

Filters Manage Crowd Complexity

- Different filters for:
  - View frustum culling
  - Occlusion culling
  - LOD Selection
  - Shadow frustum selection

Next, I will show how to construct filters for performing view frustum culling, occlusion culling, LOD selection, and shadow frustum selection.

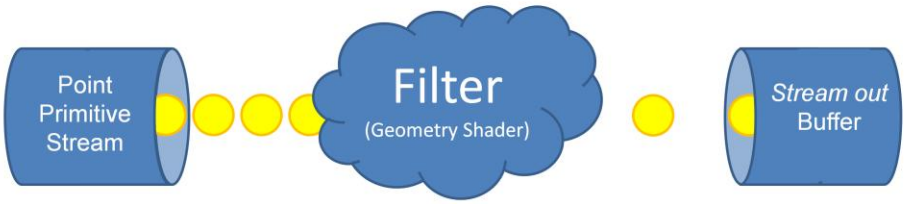
View Frustum Culling

- Filter removes characters outside view frustum
  - Checks for intersection between character's bounding volume and the view frustum

For view-frustum culling, the vertex shader simply performs an intersection check between the character bounding volume and the view frustum.

View Frustum Culling

- Filter removes characters outside view frustum
  - Checks for intersection between character's bounding volume and the view frustum
  - If test **passes**, character is **in view**: emit it
  - If test **fails**, character is **out of view**: discard it

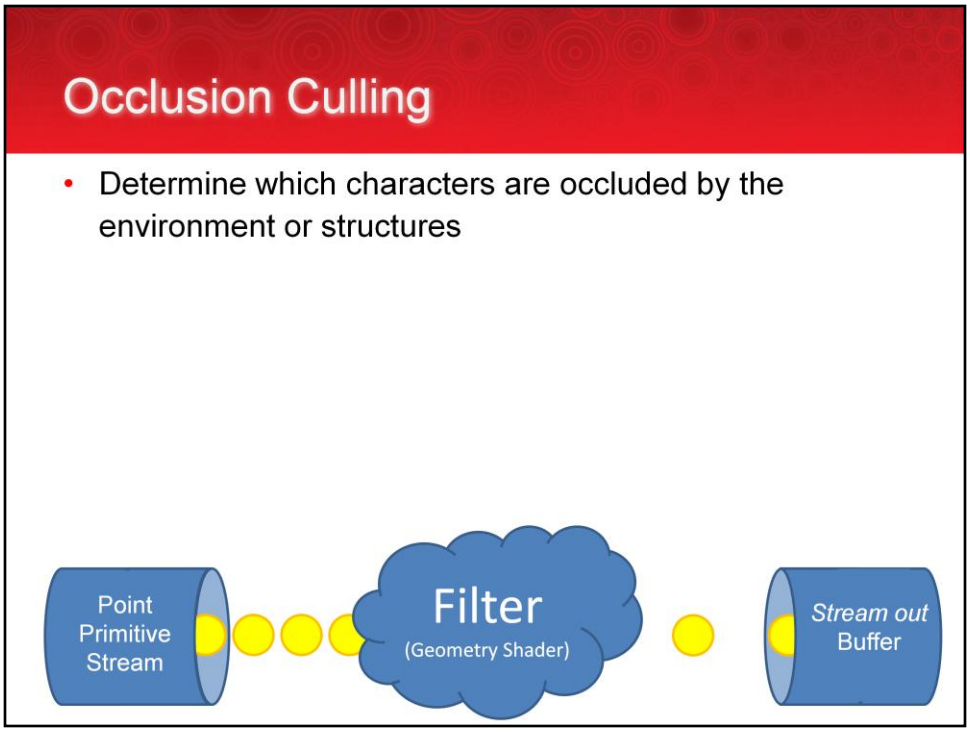


If the test passes, then the corresponding character is visible, and its instance data is emitted from the geometry shader and streamed out. Otherwise, it is discarded and the character's instance data is removed from the stream.

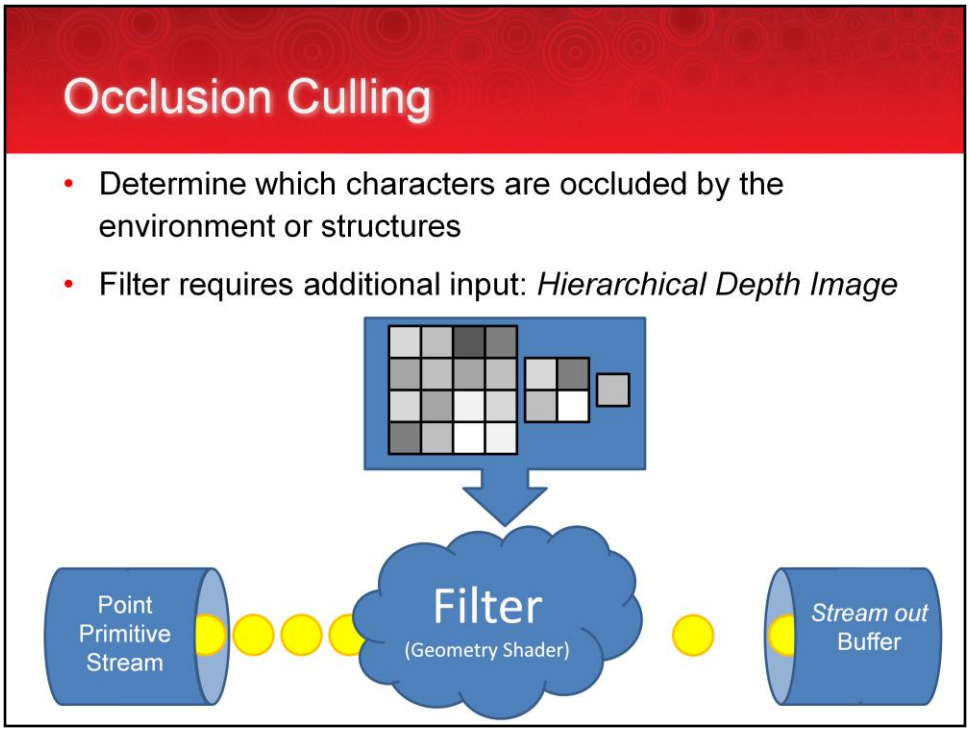
View Frustum Culling

- Filter removes characters outside view frustum
  - Checks for intersection between character's bounding volume and the view frustum
  - If test **passes**, character is **in view**: emit it
  - If test **fails**, character is **out of view**: discard it
- Output is buffer of potentially visible characters
- Output becomes input to subsequent filters

The output of this process is a buffer containing instance data for all characters that fall inside the view frustum. This instance data can then be recirculated and used as input to subsequent filters.



Using this framework, we can also perform occlusion culling to avoid rendering characters which are completely occluded by mountains or structures. Because we are performing our character management on the GPU, we are able to perform occlusion culling in a novel way, by taking advantage of the depth information that exists in the hardware Z buffer.

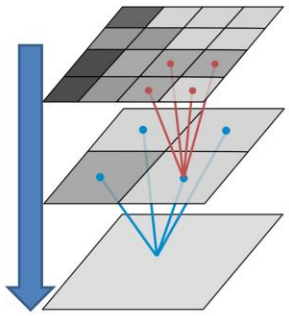


This approach requires far less CPU overhead than an approach based on predicated rendering or occlusion queries, while still allowing culling against arbitrary, dynamic occluders. Our approach is similar in spirit to the hierarchical Z testing that is implemented in modern GPUs.

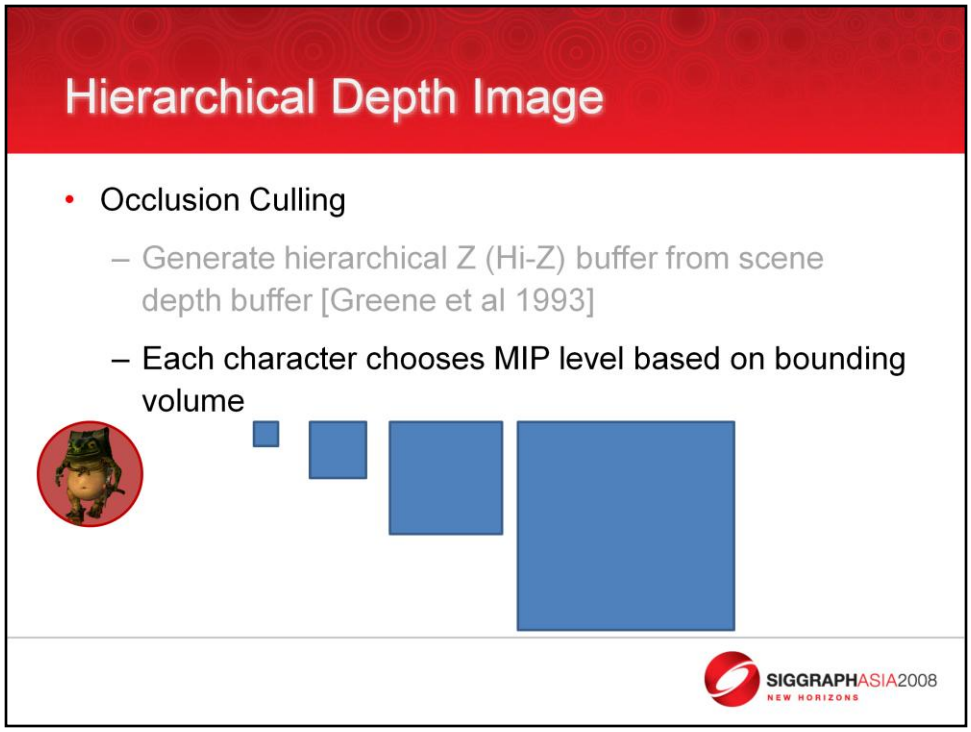
Here our filter uses the hierarchical depth image to test if characters are fully occluded.

Hierarchical Depth Image

- Occlusion Culling
  - Generate hierarchical Z (Hi-Z) buffer from scene depth buffer [Greene et al 1993]



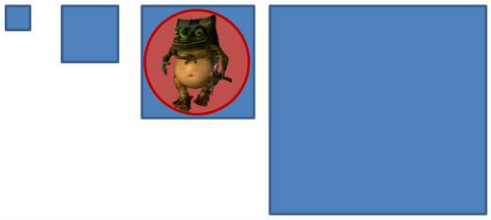
After rendering all of the occluders in the scene, we construct a hierarchical depth image from the Z buffer, which we will refer to as a *Hi-Z map*. The Hi-Z map is a mip-mapped, screen-resolution image, where each texel in a given mip level contains the maximum depth of all corresponding texels in the previous mip level. In the most detailed mip level, each texel simply contains the corresponding depth value from the Z buffer. This depth information can be collected during the main rendering pass for the occluding objects; a separate depth pass is not required to build the Hi-Z map.



Each character's bounding sphere is projected into a appropriate level of the hierarchical depth image.

Hierarchical Depth Image

- Occlusion Culling
  - Generate hierarchical Z (Hi-Z) buffer from scene depth buffer [Greene et al 1993]
  - Each character chooses MIP level based on bounding volume



SIGGRAPHASIA2008 NEW HORIZONS

Each character's bounding sphere is projected into a appropriate level of the hierarchical depth image.

Hierarchical Depth Image

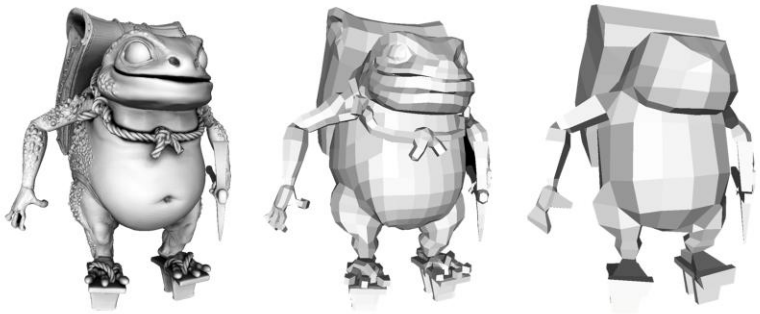
- Occlusion Culling
  - Generate hierarchical Z (Hi-Z) buffer from scene depth buffer [Greene et al 1993]
  - Each character chooses MIP level based on bounding volume
  - Projected depth of character's bounding sphere tested against four texels in chosen MIP level

SIGGRAPHASIA2008 NEW HORIZONS

And a conservative depth test is performed against the texels in the chosen MIP level. If the character's bounding sphere passes the test, the instance data is streamed out. Otherwise, the filter removes the character's instance data from the stream.

LOD Selection

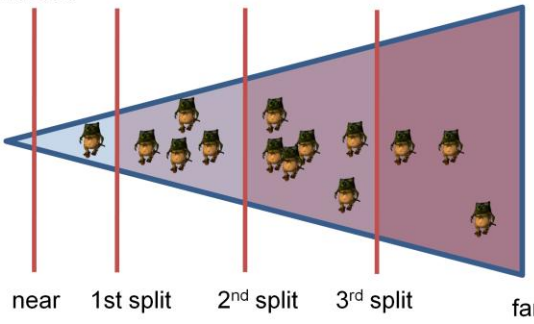
- Agents filtered using distance from camera to centroid
- Uses results of culling filters buffer
- We use three levels of detail
  - Three filter passes into three buffers



LOD selection is also performed using a filtering scheme. Only those instances that have passed the previous frustum and occlusion filters are sent as input to this filter. The LOD selection filter tests visible characters' distance from the camera. This filter is executed once for each LOD group. During the first pass, characters that fall into the first LOD group are selected. Three passes are used to create 3 stream out buffers that contain instanced characters in each LOD group. These groups can then be used to render instances of the 3 character LODs shown here.

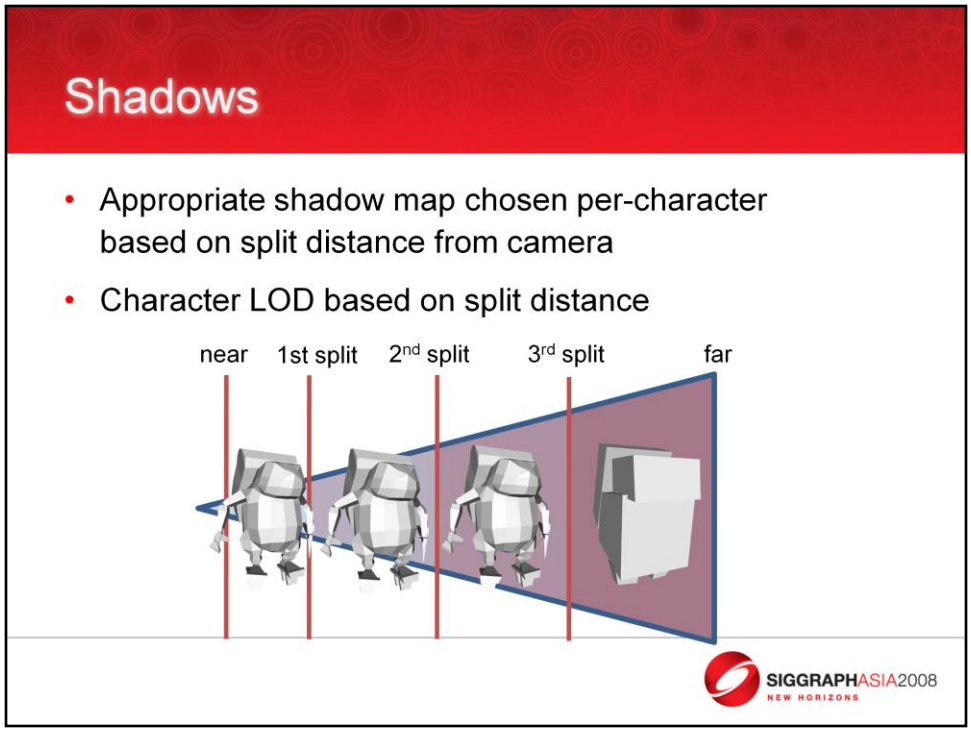
Shadows

- *Parallel Split Shadow Maps* [Zhang et al. 2006]
  - Several shadow maps, selected by distance from camera



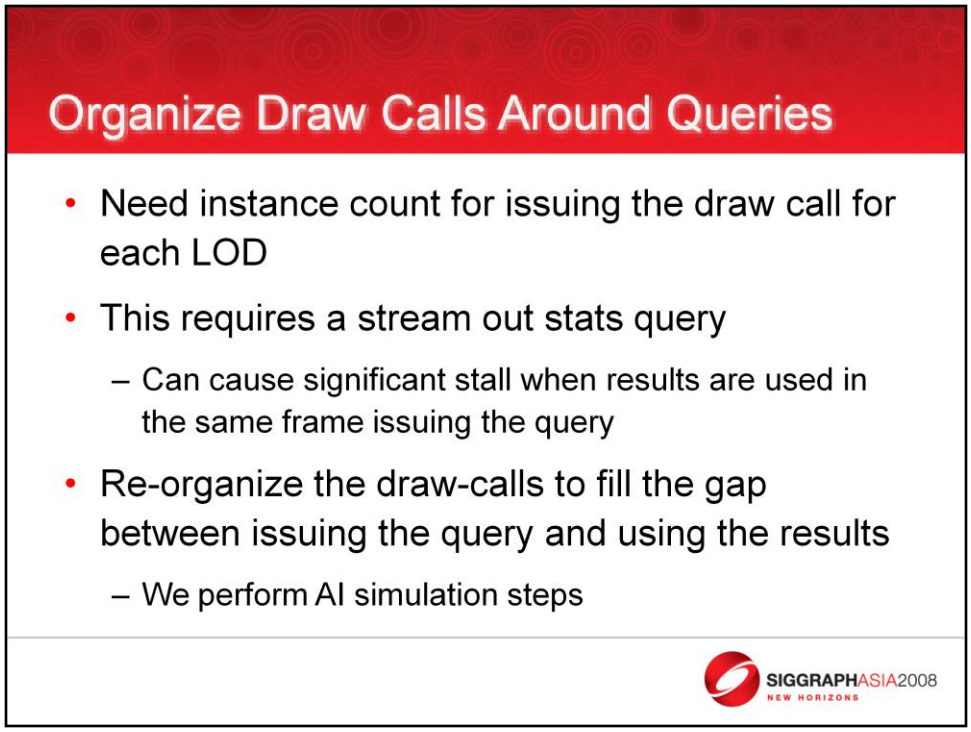
SIGGRAPHASIA2008 NEW HORIZONS

We use a very similar system for generating our shadow maps. We use a Parallel Split shadow mapping scheme where several shadow maps are used for different segments of the view frustum. Parallel split shadow maps require that characters be, once again, sorted by their distance from the camera. This time we must construct groups of instance data for characters that fall into the various shadow map splits. This filter is a combination of the frustum culling filter and the LOD selection filter.



Once we have character instances grouped by shadow map splits, we can use the grouped instances to draw our characters into shadow maps. In our application we used the low LOD mesh to draw characters into the first three levels of the shadow map and used a even coarser mesh to draw characters into the final, furthest shadow map. As you can see in the image, this coarse mesh is comprised of just a few boxes. This is ok because this shadow map will only be used for characters that are very far away from the viewer.

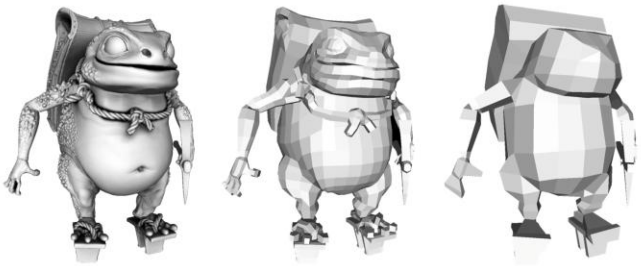
Character Rendering



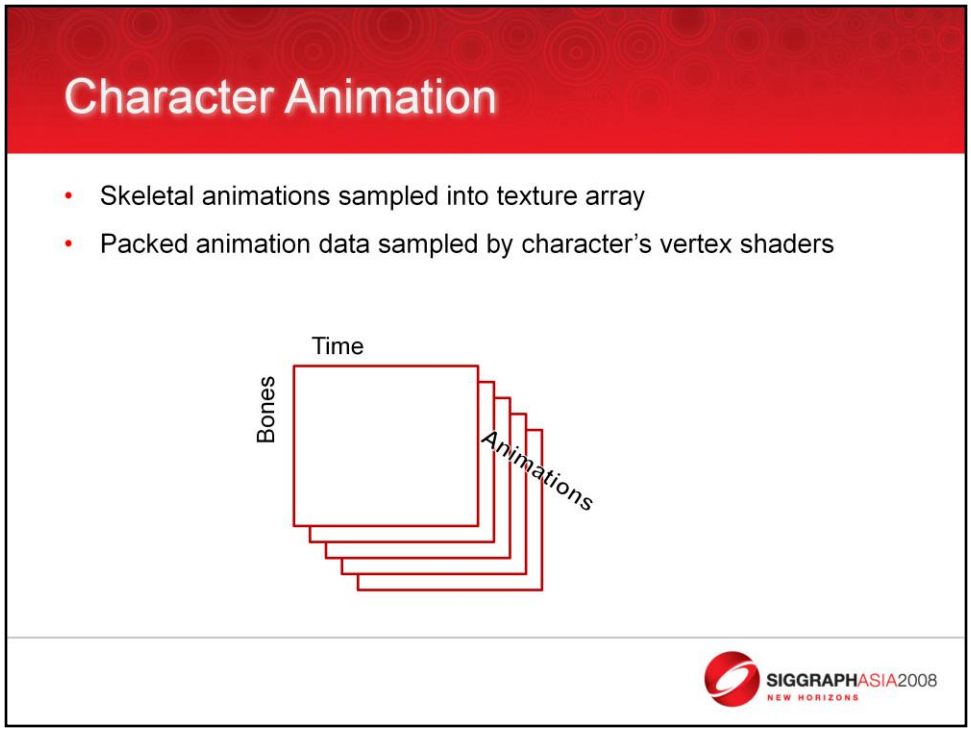
Once we've determined the visible characters in each LOD, we would like to render all of the character instances in each given LOD. In order to issue the draw call for a given LOD, we need to know the instance count. Obtaining this instance count unfortunately requires the use of a stream out statistics query. Like occlusion queries, stream out statistics queries can cause significant stalls, and, thus, performance degradation, when the results are used in the same frame that the query is issued, because the GPU may go idle while the application is processing the query results. However, an easy solution for this is to re-order draw-calls to fill the gap between previous computations and the result of the query. In our system, we are able to offset the GPU stall by interleaving scene management with the next frame's crowd movement simulation. This ensures that the GPU is kept busy while the CPU is reading the query result and issuing the character rendering commands.

Character Rendering

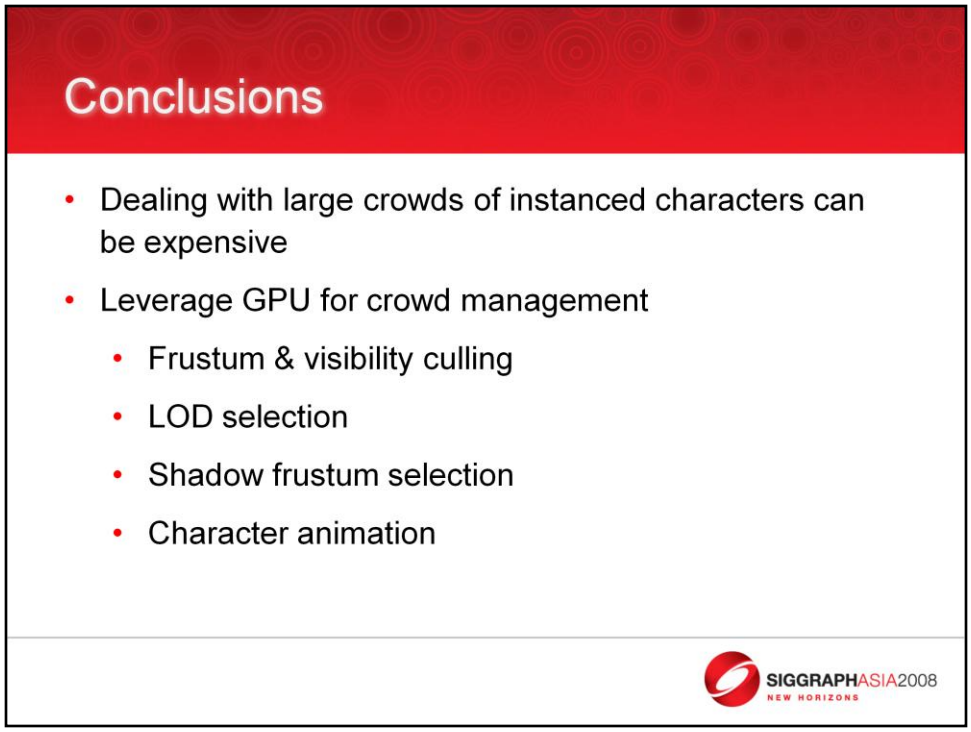
- *DrawInstanced()* call for each LOD
- Hardware tessellation and displacement mapping for closest LOD
- Conventional rendering for middle LOD
- Simplified geometry for farthest LOD



Using the filtering techniques described earlier, we construct LOD groups of visible, frustum culled, instance data. The instance data is used to draw instaced character meshes. An appropriate mesh is used for each LOD group. As you would expect, the nearest LOD uses a very finely detailed mesh which uses hardware tessellation and displacement mapping to represent fine surface detail. Tessellation and displacement mapping are disabled for the medium LOD and a coarse approximating mesh is used for the furthest or lowest LOD.



Because we are doing all of our crowd management on the GPU, we must also perform mesh animation on the GPU. We do this by storing bone animations in textures. A texture array is used to store all of the animations. Each page of the texture array contains point sampled curves for all the bones for that particular animation. Using linear interpolation, these textures are sampled to give piece-wise linear bone animation reconstruction.



I have presented a method for managing large crowds completely on the GPU. The GPU is used to perform filtering operations on a stream of instance data and these filters implement frustum and visibility culling, LOD selection, and shadow frustum selection. Additionally, I had shown how an animation system can be moved to the GPU to support.

Questions?

chris.oat@amd.com


Thank you!