Benjamin Roghani

CSCI-6962 Advanced Computer Graphics, Fall 2005
Final Project

Any algorithms or data structures you implemented:

Used Occlusion Testing:

Occlusion testing is the process of determining the visibility of an object. A hardware occlusion test quickly determines how much of an object will be visible from a given camera viewpoint by utilizing existing depth and stencil testing hardware. The *nv_occlusion_query* and *arb_occlusion_test* OpenGL extensions expose these capabilities by returning the number of fragments of a set of tested geometry which pass both the depth and stencil tests.

Algorithm:

Stage 1 – View Frustum Culling – Optional [Use for Perspective, Don't use for Global]

Position the camera where it will be when the frame is drawn. Next, test the bounding volumes of all objects and all light sources in the scene against the camera's view frustum. All passing objects are placed into an array for further processing, while failing objects are discarded. Objects which lie partially inside the frustum and partially outside the frustum can optionally be broken into sections, with passing sections being placed into the array and failing sections being discarded.

Stage 2 – View Occlusion Culling – Also Optional

In this stage, the number of objects present in the array is reduced by eliminating all objects which are inside the camera's view frustum, but are completely occluded from the camera's view. This is accomplished using occlusion testing, by removing from the array all objects for which the occlusion test returns false.

Stage 3 – Light Source Frustum Culling – Optional

This is a conditional stage which increases the efficiency of 2D light sources and directional light sources with angles of influence less than 180 degrees. Omni-directional light sources and light sources with angles of influence greater than 180 degrees skip this stage entirely. If at least one light source enters this stage, the camera's projection and modelview matrices should be pushed onto the stack before the any processing is done. 2D light sources which emit light from both sides are treated as two separate sources that emit light in opposite directions by this stage.

In this stage, all light sources passing the aforementioned criteria are assigned view frustums (which are usually exaggerated unless the light source is rectangular). These view frustums have near planes just in front of the end of their light source's own geometry, and far planes which are the same distance from their near planes as the camera's far plane is from its near plane. The center point of each light source is collinear

with the centers of its frustum's near and far planes, and with the light source view vector. The light source view vector is the normal of an imaginary plane, facing in the direction that the light source emits light, located at the light source's center point. The up vector of the light source is the cross product of the light source position and the light source normal. The up vector can also be determined by rotating the light source view vector 90 degrees about the light source center point, along the y-axis; this rotation can be performed using the rotation matrix discussed in the preliminaries section. The field of view of each frustum is equivalent to its angle of influence, unless its angle of influence is 180 degrees, in which case, the field of view must be reduced by some constant due to limitations of the projection matrix (this does induce a small accuracy issue because light should be emitted between 180 – C degrees and 180 degrees (C = small constant), but it isn't).

For each light source being processed by this stage, a viewing frustum should be created and then the projection and modelview matrices should be pushed onto the stack (the final light source to be processed need not push its matrices onto the stack, as it will be using them immediately). Next, the bounding volumes of all objects within the object array should be tested against the frustum of each light source, in the reverse order that the frustums were created, and any objects which fail all of the frustum tests should be discarded from the object array. After all objects have been tested against a frustum, the next projection and modelview matrices should be popped off the stack so that the objects can be processed against the next frustum, but the final light source should not pop the camera's matrices off the stack.

Stage 4 – Software Back-face Culling – Optional

This stage removes the back-faces of all objects currently within the object array. This is done by testing the camera's position against each face's plane equation; if the camera position is in front of the plane, the face is kept in the array; otherwise, it is discarded. This stage reduces the future number of required occlusion tests by 50%.

Stage 5, 6 and 7 – Per Vertex Back-face and Frustum Culling & Occlusion Testing

All vertices remaining in the object array will be processed by this set of stages. This stage creates a view frustum (or more) for each vertex remaining in the scene (read object array). In this stage, one of two approaches can be taken; I'll describe them in separate paragraphs due to the great differences between them. The tradeoffs are as follows: the high quality approach executes each test separately for each visible light for each vertex; as such, it requires more fill rate, has more frustum culling overhead and requires more occlusion tests, but doesn't sacrifice any accuracy or precision; the low quality approach, on the other hand, batches all per-vertex work into a single occlusion test, sacrificing a good amount of precision and accuracy for speed.

High Quality Approach

First, disable writing to the color buffer, then, for each vertex of each face, process each light source separately. First, test the light source's center position against the face's plane equation; if the light source is behind the plane, discard it; if it is not behind the plane, test each of the face's vertex positions against the light source's plane equation; if a vertex position is not behind the plane, process the light source; otherwise, skip that

vertex and test the light source against the next vertex. Each time a vertex passes a light source plane test, create a view frustum from the perspective of the vertex, fit around the light source. The far plane of this frustum should be behind the light source (at the back of the bounding volume), and the near plane should be just in front of the vertex. The view (center) vector of the view frustum should extend directly from the vertex position to the light source center point. The up vector can be computed using the same method as used in Stage 3. The field of view can be derived by the following equation: $\theta = \tan(Fw / 2) * 2 * D$. Where θ (theta) is the field of view, tan is the tangent function, Fw is the width of the far plane, and D is the distance from the vertex position to the far plane. The width of the far plane, Fw, is equal to the width of the bounding volume of the light source. Once this view frustum has been created, frustum culling should be performed against the objects in the object array, with each passing object being drawn to the depth and stencil buffers. After frustum culling is complete, disable writing to the depth and stencil buffers, and begin an occlusion test. Finally, draw the light source geometry vertex buffer, end the occlusion test and re-enable writing to the depth and stencil buffers. Repeat that process for each vertex of each face in the object array. After all faces and vertices in the object array have been processed, re-enable writing to the color buffer.

Low Quality Approach

First, disable writing to the color buffer, then, for each vertex of each face, process all light sources together. First, test each light source's center position against the face's plane equation; if the light source is behind the plane, discard it; otherwise, place a pointer to the light source into a temporary array for this face. After all light sources have been tested against the face's plane equation, begin to test the face's vertices. For each vertex, test the vertex position against the light source's plane equation; if a vertex is not behind the plane, place the light source geometry into a server-side vertex buffer; otherwise, discard it and test the next light source. Once all light sources have been tested against the vertex, create a view frustum from the perspective of the vertex, with a field of view of 180 – C, where C is a small constant, just as in Stage 3. The far plane of this frustum should be behind the farthest light source (at the back of its bounding volume), and the near plane should be just in front of the vertex. Use the vertex normal as the view (center) vector of the view frustum. The up vector can be computed using the same method as used in Stage 3. Once this view frustum has been created, frustum culling should be performed against the objects in the object array, with each passing object being drawn to the depth and stencil buffers. After frustum culling is complete, disable writing to the depth and stencil buffers, and begin an occlusion test. Finally, draw the light source geometry vertex buffer (which contains all light sources visible to this vertex), end the occlusion test and re-enable writing to the depth and stencil buffers. Repeat that process for each vertex of each face in the object array. After all faces and vertices in the object array have been processed, re-enable writing to the color buffer.

Note about the Low Quality Approach

Note that the aforementioned explanation assumed that all of the light sources fall within the described ~180 degree frustum. In reality, this might not be the case. This process can either be repeated multiple times (up to 5, in the worst case scenario, if one samples in a cubemap-esque fasion), until all of the lights have been sampled, or one can revert to the High Quality approach. The largest downside to sampling using the low

quality approach with up to five cameras, is the selection of which camera to sample a light source from, when more than one camera can see that light source. This becomes an optimization problem aiming to minimize sampling error, and is not something I've focused on yet (Currently, I've only implemented the high quality approach). This topic should be covered extensively in Computer Vision and Mobile Sensing research/literature.

The core features of your assignment and how you tested them:

- Preliminary culling of the scene in the Perspective case to increase efficiency.
    - Tested this by visualizing bounding volumes and moving the camera while watching that the objects were culled when you would expect them to be.
- Per vertex (on models in the scene) visibility sampling of arbitrary geometric Area/Volume Light Sources (In this case, represented as a triangle mesh).
    - Tested that the samples were sampling the right thing by visualizing the sampling process (rendering to the color buffer instead of just the depth buffer, using blue for light sources and green for occluders). It was visually obvious that this part of the algorithm was functioning correctly.
    - Tested this by dumping the per-vertex results of the sampling tests to a file and inspecting it to ensure that vertices had real data (as opposed to all 0's as they were initialized with, or all identical numbers like 3billion-something [indicating that the light source took up the same number of fragments on every vertex, despite differences in distance from the light source, and occlusion.].).
- Transfer of per-frame (perspective case) lighting data to the graphics card via fast hardware Vertex Buffer Objects.
    - Tested this first by testing to see if the buffers were initialized correctly, and if the scene could draw [without crashing] with them enabled (both of which failed, eventually I fixed the first problem, and a long while later, the second).
- Rendering of the scene with shaders which interpolate per vertex lighting data per pixel and then utilize that lighting data to determine how brightly to shade each pixel in the scene.
    - Tested this by coloring the scene green where lighting samples were > 0 and red where samples where = 0. This indicated (eventually, after a lot of head banging [not from rock music]), that the lighting data was reaching the shader correctly.

The challenges that you overcame (or failed to overcome):

- I seem to have introduced a bug in my engine with regard to model loading, which I was unable to fix (and I'm working off of a non-version controlled fork, so I couldn't tell where I introduced it), so I developed the stand-alone demo implementation of the algorithm, which required re-writing tons of code – and writing lots of new code, in addition to re-writing large parts of the

algorithm itself, (that part for good measure when I couldn't find a different bug).

- As mentioned above, I overcame several problems with Vertex Buffer objects:
  - Creating and drawing multiple vertex buffer objects, where static data for a single model resided in one VBO, while the lighting sample data for the same model resided in a different, dynamic VBO.
  - Mapping the buffers for dynamic streaming of per-frame data using VBOs.
  - Many small problems related to the above two bullets.
  - Finding a data-type which I could disguise the lighting data as (I initially tried texture coordinates, then normals, and finally ended up using color data.)
  - Getting the vertex shader to accept the dynamic data correctly.
- Implementing the Cg shaders in the demo (I've used Cg in my engine, but I had to change a lot to get it working in the demo).
- I didn't overcome problems with trying to do a simple static hack of tone mapping without creating any textures, so I gave up on that for now, and am going to work on an implementation of a real tone mapping algorithm later.
- I didn't have time to implement the global illumination (radiosity) version [But, I did think a lot about how to do it.]
- I didn't have time to implement the minimal recomputation – caching version (for when the scene changes) – but that would be easy, the way I've structured the code.

Any known bugs or limitations in your implementation:

- Bug (Demo): after drawing the first area-light shaded frame, one of the matrices gets fubar'd, and all of the geometry starts projecting oddly, in some cases, to infinity.
- Bug (Engine): as mentioned above, something is now wrong with my model loading code, and the geometry is just plain fubar'd in general.
- Limitation (Demo): I'm currently only using one hard-coded light source (hard coded in that it takes the first object in the model and uses that as the light source). This would be about a 5 minute fix, but I didn't have the ability to make new models, so I didn't really focus on writing code for extracting light sources from the models.
- Limitation (Demo): For now, the scene has to be composed of separate meshes inside a single .3DS model (file).
- Limitation (Demo): I hacked (commented) the pixel shader to simply color the scene in grayscale using the lighting value as the intensity (for R,G, and B). This would only take a few minutes and a good model to fix (to have Phong working).
- Limitation (Demo): Textures aren't loaded.

Note that these limitations don't exist on the Engine; however, the model loading bug is getting in the way of using that version for now.

How long it took you to complete the assignment:

About 85 hours, most of which was unfortunately wrestling with the VBO problems, and another good chunk of which was re-implementing the standalone demo (because of the geometry bug in the engine). It took about 20 hours to implement the algorithm and its surrounding code initially (before trying to tackle all the bugs to get it working). Another chunk of time was devoted to optimization and getting that to work (the culling). Writing the shaders only took about 30 minutes, but debugging them took a lot longer, mainly due to the relationship with the VBO problems. I also spent about 2 hours on tone-mapping, most of which [implementation-wise] was a waste, but at least I gained knowledge from the reading I did on them [tone-mapping algorithms].