# Some Ideas on Shadow Algorithms

Hard Shadows

Let's consider a games scenario for a few moving ``characters'' over a 3D static background of the scene. Can we achieve fast shadow computations while retaining high quality? This idea revolves around the feasibility of using multiple shadow depth maps per shadow casting light.

In the shadow depth map generation step:

- The user can identify the floor(s) and not put it as part of any shadow depth map. This still allows shadows to be cast on the floor but there is no self-shadowing on the floor. Not a bad assumption?
- For each moving character, create a single shadow depth map that tightly fights just that character.
- For static items, clump it together in some spatial manner---occupying neighboring regions, either in worldspace or lightspace---and create a shadow depth map for each clump. These clumps can be summed up to have them represented by one shadow depth map.
- The above steps will create multiple shadow depth maps per shadow casting light.

Then for the shading step, we can create a spatial data structure to keep track of the bounding box of the clumps of static items, and bounding box of the characters. When rendering the shading step, we can easily identify which shadow depth maps we need to access based on the bounding boxes' information. We compute shadowing based on those shadow depth maps.

The advantages of this approach include:

- Each shadow depth map focuses the resolution just on the characters (or static items), with little useless free space in each shadow depth map. As a result, the quality should be quite good.
- Shadow depth map generation can easily be parallelized.
- During the game, only the characters' shadow depth maps need to be regenerated (assuming everything else is static). Thus regeneration of shadow depth map should be quite fast.
- There should be a lot of coherency in accessing the shadow depth maps.

But:

- What is the best approach to clumping regions for the occluders---in worldspace or lightspace?
- Is the context switches between multiple shadow depth maps faster or slower than a single (higher resolution) shadow depth map?

Soft Shadows

In [643], a technique to accelerate distributed ray tracing was proposed for triangles with respect to a linear light. Basically a rayID is stored per triangle, and a global rayID is created and incremented only after the total set of distributed rays are shot for the current point to be shaded. The triangle rayID, when not equal to the global rayID, does a triangle and linear-light-triangle intersection test, then the triangle flags 0 if there is an intersection and 1 if not. When the triangle rayID is equal to the global rayID and the triangle flag is 1, then we know that this triangle cannot possibly occlude for this point to be shaded. In this way, there are many instances where a cheap triangle-triangle intersection can cull away many potential occluders from being tested for ray-triangle intersection tests.

Results from [643] show that there were negligible improvements. Why? One possibility could be that although this approach can save a lot of flops, the memory accesses of a large data set remains the same, which is likely the main performance costs of the ray tracer. Can we employ something similar but improve memory cache hits? How about...

Instead of storing a triangle rayID, we store a rayID per voxel (assuming a voxel based intersection culling structure). A global rayID is incremented every time a point to be shaded is done (regardless of how many shadow rays are shot). The voxel rayID is set to this global rayID when a shadow ray hits this voxel during traversal.

If the voxel rayID is not equal to the global rayID, the ray-triangle intersections for this voxel does some extra work. We do the triangle and linear-light-triangle intersection test. If there is no intersection, then any extra shadow rays shot from this shaded point cannot possibly hit this triangle. And the number of such intersection hits should be small per voxel. Any triangles that intersects the linear light triangle will be added to the voxel's dynamic array of triangle candidates. Any extra shadow ray which hits this voxel again (for the same shaded point) should have the voxel rayID equal to the global rayID, and should only go through this (much reduced) dynamic array to do the actual ray-triangle intersection tests.

Using this approach, the number of ray-triangle intersection tests should be dramatically reduced, and not much more than just for a point light. And because the dynamic arrays tend to be small, and there is likely coherency in accessing similar set of the dynamic arrays, then L1 cache hits would tend to be high to make them quite fast to access.