



AUTOMATED OCCLUDERS FOR GPU CULLING

HIERARCHICAL Z-BUFFER OCCLUSION CULLING AND HOW TO AUTOMATE OCCLUDER CREATION

THERE ARE LOTS OF OBVIOUS REASONS WHY PERFORMANCE CAN TAKE A HIT IN YOUR GAME, BUT IT'S WHAT YOU CAN'T see that could be hurting you the most. One of these areas is unseen geometry. The art of culling unseen geometry from games has evolved over the years—developers often use a combination of frustum culling, BSP, portals, precomputed visibility, and occlusion queries to prevent the game from rendering objects that are outside the player's view. One of the more interesting solutions to emerge recently is hierarchical z-buffer occlusion culling (hi-z culling). This algorithm provides us with a fast and nearly fixed-time solution to determining visibility for static and dynamic objects in the game world. It should not be confused with the hardware hierarchical z-buffer that is used to reject pixel quads.

In this article I'll introduce you to the hierarchical z-buffer occlusion culling algorithm, and explain how you can automate the creation of proxy geometry for the occluders to minimize impact on the productivity of your artists and level designers.

BACKGROUND

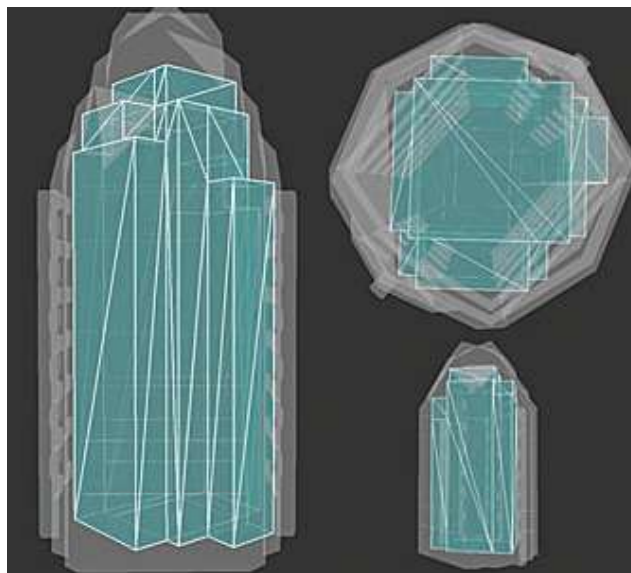
The first example of this hierarchical z-buffer occlusion culling algorithm can be traced to a paper called "Advances in Real-Time Rendering" from SIGGRAPH 2008 (Section 3.3.3). Since then, several high-profile games have extended this method to cull geometry in their renderer. TOM CLANCY'S SPLINTER CELL: CONVICTION and KILLZONE 3 both use this approach, and DICE has integrated it into the company's Frostbite 2 engine.

Before I explain how to automate generating occluders, let me first get into how the Hi-Z culling algorithm works, to illustrate why automating occluder generation is important.

HOW IT WORKS

The goal for any culling algorithm is to determine the smallest visible or potentially visible set as fast as possible. GPUs offer their own occlusion culling system in the form of occlusion queries, which report the number of pixels that pass the z-buffer test, but that means having to render the exact mesh you're trying to avoid rendering. So in the end you don't save that much GPU time unless the object contains very complex materials.

A far better solution is to take a simpler representation of an object, like its bounding box, and test to see if just the bounding box is visible. In order to know if the object's bounding box is visible, we need to know about the large occluders that could block it.



Determining the Building Mesh (White) versus the Occlusion Mesh (Teal).

This is the core concept behind hierarchical z-buffer occlusion culling. First we render simple representations of the large occluding bodies (buildings, walls, terrain, and so forth) to a depth buffer. We render simple proxies because we need the rendering to be performed very quickly, and the original meshes are likely to be a little too complicated or may need too many draw calls to be useful.

After rendering the proxies to a depth buffer we create mipmaps (a hierarchy) of depth buffers. Using the hierarchy we can batch up the entire list of potentially visible object bounding boxes and test them all at once on the GPU at different levels in the mipmap, representing different granularities of depth buffer depending on the size of the object.

Large objects use a very coarse granularity depth buffer and are more likely to be visible. Small objects use a very fine granularity depth buffer and are less likely to be visible.

1 // RENDER OCCLUDERS

The first thing we need to do is render simple pieces of geometry representing the occluders in the level. You should start out by frustum culling the occluders on the CPU to reduce the draw call overhead. If you have a bake or cooking step in your level editor, that would be a great time to merge groups of the static occluder meshes into a single mesh to further reduce the draw call overhead.



Figure 1: Hierarchical Z-Buffer Downsampled Mipmaps.

The remaining potentially visible occluders will be rendered to a small (512x256) render target with a full mip chain. You don't have to use a render target with the same dimensions in your implementation, but using a power of two textures simplifies the code. You could also choose a smaller buffer, but you risk a loss in accuracy for smaller objects on the screen, potentially culling objects that would be visible in a full-size depth buffer.

DirectX 11 cannot create a hardware depth buffer with a mip chain. So instead we need to render the scene and write the depth values to the first mip level in our render target.

2 // DOWNSAMPLE THE DEPTH BUFFER

The next step is to fill out the hierarchy of z buffers by downsampling the first mip level that we just rendered to into the other levels of the mip chain. This will result in conservative depth buffers with coarser and coarser approximations of the occluders in the environment that we can test bounding boxes against (see Figure 1).

In order to perform the downsampling operation we can render a full-screen quad with a pixel shader that takes as input the previous mip level and conservatively downsamples it into the current mip level. Conservatively downsampling the depth values differs from standard mipmap downsampling because we have to preserve the highest depth value in a sample group of four pixels that are merging into a single pixel.

We do this because as the depth information is compressed and lost at each mip level we have to assume the worst case—that more things are visible—otherwise we may cull something that is actually visible. Therefore we take the furthest depth from that group so that we don't accidentally cull something that might be visible in one of the compressed pixels.

When performing the downsampling operation, one advantage of DirectX 11 over 9 is that you can constrain the shader resource view so that you can both sample from and render to different levels in the same mip chain. In DirectX 9 you had to copy a separate render target into the mipmap because there was no way to isolate a single mip level to sample from. The following pixel shader excerpt demonstrates how you can efficiently downsample four pixel groups conservatively.

```
Texture2D<float> PrevMip;
SamplerState PrevMipSS;
...
float4 PS( PS_INPUT IN ) : SV_TARGET {
    float4 vTexels =
        PrevMip.Gather(PrevMipSS, IN.texcoord);
    return max(max(vTexels.x, vTexels.y),
               max(vTexels.z, vTexels.w));
}
```

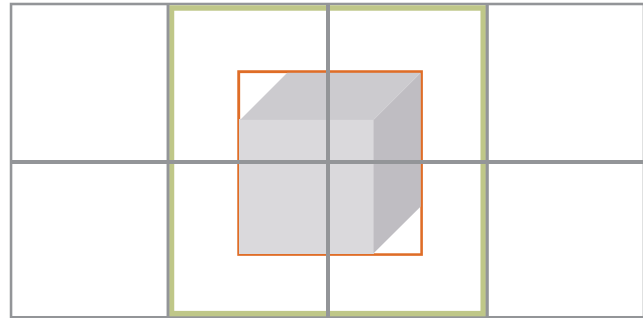


Figure 2: Screen Space Bounding Box (Orange), Sampled Pixels (Green).

3 // TESTING BOUNDS

In order to determine visibility we need to find the level in our depth buffer mip chain where the object takes up at most a four-pixel region. This is so we only have to test four points for visibility instead of testing every point that makes up the bounding box in screen space. This implies that large objects on the screen will sample from low-resolution mip levels, and small objects on the screen will sample from high-resolution mip levels. Even though the lower mip levels are coarser and thus less accurate, large objects are likely more visible on the screen anyway.

The testing algorithm is broken up into five discrete steps to perform in the compute shader:

1. Perform a frustum cull on the bounding box or sphere. If it is outside the frustum, it's not visible.
2. Determine the maximum screen space width or height of the bounding box (represented by the red box in Figure 2), taking the maximum value calculate the mip level to sample from so that the object takes up no more than a two pixel square region [the green box in Figure 2]. This value can be calculated using, $\text{ceil}(\log_2(\text{max_screen_size}))$.
3. Take the four values making up the screen space bounding box of the object's bounding sphere or box and sample at those four locations in the mip level computed in step 2.
4. If the maximum sampled depth value is closer to the camera than the closest point on the bounding sphere or box then we can conclude the object is entirely behind the occluding surface and therefore not visible.
5. Store a float in a writable buffer representing visible (1) or not visible (0).

4 // READING THE RESULTS

After dispatching the compute shader to test all the bounds for visibility, you'll have time to do some processing on the CPU while you wait for the results. The amount of time obviously varies depending on the GPU and the number of bounds, but for a reasonably new GPU and 1,000 bounds the processing time should fall well below 1ms. Keep in mind, though, that the GPU could be performing previous work when your compute shader is dispatched, which would increase the time it takes to complete.

Once the compute shader finishes, it's just a matter of locking the writable buffer from steps 3–5, and copying the float values representing visible or hidden objects into a CPU-accessible buffer so that a final list of visible objects can be prepared for rendering.

5 // EXTENSIONS

The algorithm can be extended to also test whether shadows are visible as well. By creating a hi-z buffer from the large directional shadow casting

light in the scene (generally the sun) and using the hi-z buffer of the observer you can determine whether a shadow volume is visible.

By extruding the bounding box of an object in the light space hi-z buffer until all four sampling points collide with a surface, you can generate a new volume representing the shadow bounding box. This “extrusion” all happens in the shader; you’re not actually generating geometry for this volume. This new bounding box can then be tested against the observer’s hi-z buffer for visibility to know if an object off-camera is casting a shadow visible to the player. If the shadow volume is not visible, you don’t have to render the object into your shadow map.

ART PIPELINE INTEGRATION

With any new change in the rendering pipeline a studio must consider what, if any, ramifications it will have on artist and level designer productivity. Because hi-z culling, in practice, renders proxy geometry to represent the occluders in a level, there is an inherent change in artist or level designer workflow because they must create this proxy geometry, unless they were already doing so for another purpose.

You may already have one source of these proxy meshes, in the form of physics meshes. In cases where your physics mesh is conservative (does not extend beyond the surface of the visible mesh) it can make for good occlusion proxy geometry. This is especially handy if your level has lots of destructive walls or buildings that also make for good occluders. One downside of this approach is that there are now many small occluders that must be updated.

Also, not every game has a physics mesh for objects that can be used as occluders. In these cases you might want to be able to automatically generate the occlusion geometry from the visible mesh, perhaps during export time in Max or Maya.

GENERATING OCCLUSION VOLUMES

Before we generate any occlusion volumes let’s consider the important characteristics of good occlusion geometries:

- **CONSERVATIVE** – Doesn’t extend beyond the surface of the mesh.
- **SIMPLICITY** – The occlusion mesh is made of very few triangles or is fast to render.
- **VOLUME CONSERVATION** – Closely matches the original mesh’s volume.
- **MOVABLE** – Some games have large moving occluders or destructible walls.

Normal methods of simplifying a mesh such as naive triangle simplification can cause both a significant reduction in volume as well as triangles penetrating the surface of the mesh. Neither of these are desirable outcomes.

What if instead we took the mesh and first converted it into a voxel representation. With a voxel representation we could perform our simplification operations on the volume structure of the object instead of on the topology of the object. The technique presented here does exactly that. However, it does have some caveats, and should be seen as a starting point to be extended and improved.

Let me start by summarizing the process:

1. Find all the voxels completely inside a mesh.
2. Find the voxel at the densest point in the volume.
3. Expand a box from this point until all sides collide with the mesh surface or another box.
4. Repeat 2–3 until you’ve consumed X% of the total volume.
5. Use a Constructive Solid Geometry (CSG) algorithm to merge the boxes you create.

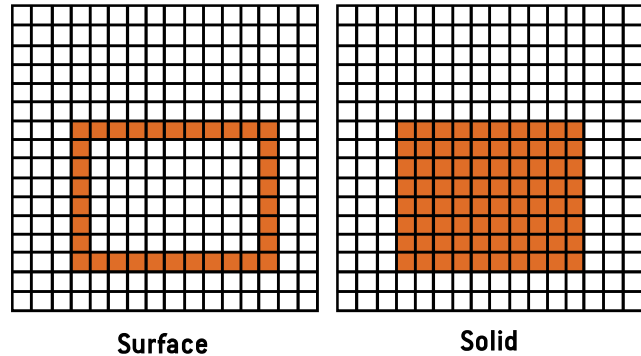


Figure 3: Surface vs. Solid Voxelization.

1 // VOXELIZATION

First you have to find all the voxels completely inside the mesh. That way we can have complete confidence that anything we generate contained inside these voxels will be conservative. It also gives us a very easy way of quantifying the total volume and the volume remaining in the object.

The algorithm I used to perform my voxelization is laid out in a paper by Michael Schwarz titled “Fast Parallel Surface and Solid Voxelization on GPUs,” sections 3.1 and 4.1.

The paper talks about two voxelization types that we care about, surface and solid voxelization (see Figure 3). Surface voxelization gives us all the voxels that intersect with any triangles in the mesh providing us a shell of the mesh. Solid voxelization gives us the voxels making up the inner volume of a mesh. By determining both of these sets we can calculate the set of voxels completely inside a mesh by removing any voxel in the surface set that is also in the solid set.

SURFACE VOXELIZATION

The algorithm for finding all the voxels that intersect with triangles on the surface of the mesh is fairly simple. You are performing a collision check between a triangle and a box. After taking each voxel and determining its size and position in model space you can perform a check against every triangle for collision.

SOLID VOXELIZATION

In order to determine whether a voxel is part of the solid voxel volume we need to know if the voxel is inside the mesh. To do this we have to shoot a ray down the center of a column of voxels along one of the major axes. When the ray intersects with triangles along the column we can use the xor operator to set the correct inside/outside status of a voxel.

Imagine a column of voxels all starting at 0, and the ray “->” starting on the far left, with triangles “|” dividing the region into inside and outside areas. We would start with this:

```
-> 000 | 00 | 000
```

As the ray crosses the boundary, all the voxels beyond the first triangle are xor-ed with 1 and are now:

```
000 | -> 11 | 111
```

Continuing on, after the ray intersects with the next boundary we again xor the values with 1 beyond the new boundary, giving us:

```
000 | 11 | 000 ->
```



0 represents outside and 1 represents inside; which is to say inside regions will be xor-ed an odd number of times, and outside regions will be xor-ed an even number of times.

One unfortunate limitation of this approach is that the mesh has to be watertight, so if there are any holes, this solid voxelization algorithm is likely to fail. This is one area that will require improvement, as much of what artists tend to create is not watertight.

2 // FIND THE HIGHEST-DENSITY VOXEL

In this step you will need to find the voxel that is the furthest away from any empty voxel, excluding any voxel already enclosed in an occlusion volume in step 3. Because the number of surface voxels is likely smaller than both the number of empty voxels and the number of total voxels you should create a list of surface voxels to test the distance against.

3 // BOX EXPANSION

Once you've found the densest voxel you should create a 1^3 -voxel size box at that location. You'll then proceed to iteratively expand each side of the box in voxel space until you can't expand any side of the box without entering an empty voxel, or another box that has already been placed.

As you verify each expansion of the box you'll mark the enclosed voxels so that the next time you choose the densest voxel you can exclude any already enclosed in a box.

The only approach I've tried is using a uniform expansion strategy. However a better approach might be to include a small amount of prediction allowing the growth of the box to maximize the number of expansions by not colliding early along one side if there is a lot of room to expand along another axis first.

While this approach will work well for axis-aligned objects, like buildings and other man-made objects, it won't work very well for more organic structures. So perhaps instead of expanding boxes, shrinking OBBs around the voxel structure of an object creating an OBB tree would be a better approach overall. However, this approach is completely untested and is only mentioned as an area of research for possible improvement.

4 // REPEAT 2-3

Continue to repeat the process of finding the high-density voxel and expanding a box until you reach a desired stopping condition. Examples of a stopping condition might include:

- An absolute percentage in volume is consumed.
- The volumes being constructed fall below a point of diminishing returns or are too small either in terms of total percentage or size in voxel space.
- A maximum number of boxes have been created.

5 // MERGE BOXES

We could stop here and create a single mesh out of these separate boxes, but rendering many overlapping individual boxes could potentially cause lots of overdraw. Instead, it would be better to merge the boxes together using mesh Boolean operations, commonly referred to as Constructive Solid Geometry (CSG).

The topic of CSG is too large to cover in this article, but a great explanation and implementation can be found in the book *Game Development Tools* in the article "Real-Time Constructive Solid Geometry," written by Sander van Rossen and Matthew Baranowski.

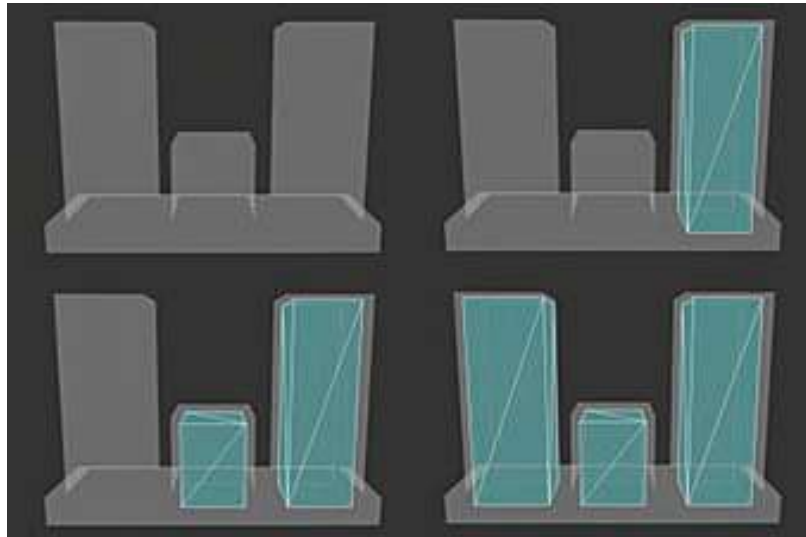


Figure 4: Box Expansion Time-lapse.

THE RESULTS

When you're finished, the process can generate results like those in Figure 4. Starting with the voxelized mesh, boxes are expanded until enough of the volume has been consumed resulting in low-poly occlusion mesh that covers most of the original volume, regardless of the polygon count of the original mesh.

CAVEATS

While this technique is a nice first step, it still has a long way to go. It's important to be able to handle non-water-tight meshes, and not having to worry about that makes things easier on the artist.

Speed can also still be an issue. Even after moving to the GPU to generate the voxel structure of objects it can still take several seconds to complete. Even though this process occurs at export time, it may still feel burdensome.

As always, the artists should be able to override anything automatically generated with a custom mesh of their own. This is important for structures like planes, where a voxel volume can't be generated for any internal structure, because a plane has none.

JUST SCRATCHING THE SURFACE

Hopefully this article has been both informative and thought provoking. While there are still many areas that need improvement, there is some benefit even at this early stage.

Additionally, as GPUs become more advanced they may become capable of issuing draw calls of their own. This is already marginally possible with instanced meshes. If it becomes possible for an arbitrary draw call, the synchronization step could be removed from the hi-z culling algorithm, allowing the GPU to both determine what should be drawn and issue the command to draw it. 🎮

The author wishes to acknowledge Mike Acton at Insomniac Games for encouragement, and starting AltDevBlogADay.com. The author would also like to thank Stephen Hill, Michael Noland, and Shaun Kime for their help and input.

NICK DARNELL is a senior software engineer currently working at Activate3D, advancing the state-of-the-art motion gaming experience for Kinect. Previously he worked on tools and engine development for Gamebryo at Emergent Game Technologies. He is also an active member in the game development AltDevBlogADay community. Nick can be found online on Twitter @NickDarnell and his blog www.nickdarnell.com.