VI

1

Practical, Dynamic Visibility for Games

Stephen Hill and Daniel Collin

1.1 Introduction

With the complexity and interactivity of game worlds on the rise, the need for efficient dynamic visibility is becoming increasingly important.

This article covers two complementary approaches to visibility determination that have shipped in recent AAA titles across Xbox 360, PS3, and PC: *Splinter Cell Conviction* and *Battlefield: Bad Company 1 & 2*.

These solutions should be of broad interest, since they are capable of handling completely dynamic environments consisting of a large number of objects, with low overhead, straightforward implementations, and only a modest impact on asset authoring.

Before we describe our approaches in detail, it is important to understand what motivated their development, through the lens of existing techniques that are more commonly employed in games.

1.2 Surveying the Field

Static potentially visible sets (PVSs) is an approach popularized by the Quake engine [Abrash 96] and is still in common use today, in part because of its low runtime cost. Put simply, the world is discretized in some way (BSP, grid, etc.) and the binary visibility from each sector (leaf node, cell, or cluster, respectively) to all other sectors is precomputed and stored. At runtime, given the current sector containing the camera, determining the set of potentially visible objects becomes a simple matter of retrieving the potentially visible sectors (and by extension, their associated objects) and performing frustum culling.

One major drawback of using PVS by itself is that any destructible or moving objects (e.g., doors) typically have to be treated as nonoccluding from the perspective of visibility determination. This naturally produces over inclusion—in addition to that coming from sector-to-sector visibility—and can therefore constrain level-design choices in order to avoid pathological situations.

Another disadvantage stems from the fact that a PVS database can be extremely time consuming to precompute,[1] which may in turn disrupt or slow production.

Portals are another approach that can complement or replace static PVS. Here, sectors are connected via convex openings or "portals" and the view frustum is progressively clipped against them [Akenine-Möller et al. 08], while objects are simultaneously gathered and tested against the active subfrustum.

Since clipping happens at runtime, the state of portals can be modified to handle a subset of dynamic changes to the world, such as a door closing or opening. But, even though portals can ameliorate some of the limitations of a static PVS solution, they are still best suited to indoor environments, with corridors, windows, and doorways providing natural opportunities to constrain and clip the view frustum.

Antiportals are a related technique for handling localized or dynamic occlusion whereby, instead of constraining visibility, convex shapes are used to occlude (cull away) objects behind them with respect to the player. Though antiportals can be effective in open areas, one can employ only a limited number in any given frame, for performance reasons. Similarly, occluder fusion—culling from the combined effect of several antiportals—is typically not viable, due to the much higher cost of inclusion testing against concave volumes.

In recent years, hardware occlusion queries (OQs) have become another popular tool for visibility determination [Soininen 08]. The canonical approach involves rendering the depth of a subset (or a simplified representation) of the scene—the occluders—and then rasterizing (without depth writes) the bounds of objects, or groups of objects. The associated draw calls are bracketed by a query, which instructs the GPU to count pixels passing the depth test. If a query returns that no pixels passed, then those objects can be skipped in subsequent rendering passes for that camera.

This technique has several advantages over those previously discussed: it is applicable to a wider range of environments, it trivially adapts to changes in the world (occluders can even deform), and it handles occluder fusion effortlessly, by nature of z-buffer-based testing. In contrast, whereas both static PVS and portals can handle dynamic objects, too, via sector relocation, those objects cannot themselves occlude in general.

1.3 Query Quandaries

On paper OQs are an attractive approach, but personal experience has uncovered a number of severe drawbacks, which render them unsuitable for the afore-

[1]On the order of 10 hours, in some cases [Hastings 07].

mentioned titles. We will now outline the problems encountered with occlusion queries.

1.3.1 Batching

First, though OQs can be batched in the sense that more than one can be issued at a time [Soininen 08]—thereby avoiding lock-step CPU-GPU synchronization—one cannot batch several bounds into a single draw call with individual query counters. This is a pity, since CPU overhead alone can limit the number of tests to several hundred per frame on current-generation consoles, which may be fine if OQs are used to supplement another visibility approach [Hastings 07], but is less than ideal otherwise.

1.3.2 Latency

To overcome latency, and as a general means of scaling OQs up to large environments, a hierarchy can be employed [Bittner et al. 09]. By grouping, via a bounding volume hierarchy (BVH) or octree for instance, tests can be performed progressively, based on parent results, with sets of objects typically rejected earlier.

However, this dependency chain generally implies more CPU-GPU synchronization within a frame since, at the time of this writing, only the CPU can issue queries.[2] Hiding latency perfectly in this instance can be tricky and may require overlapping *query* and *real* rendering work, which implies redundant state changes in addition to a more complicated renderer design.

1.3.3 Popping

By compromising on correctness, one can opt instead to defer checking the results of OQs until the next frame—so called latent queries [Soininen 08]—which practically eliminates synchronization penalties, while avoiding the potential added burden of interleaved rendering. Unfortunately, the major downside of this strategy is that it typically leads to objects "popping" due to incorrect visibility classification [Soininen 08]. Figure 1.1 shows two cases where this can occur. First, the camera tracks back to reveal object A in Frame 1, but A was classified as outside of the frustum in Frame 0. Second, object B moves out from behind an occluder in Frame 1 but was previously occluded in Frame 0.

Such artifacts can be reduced by extruding object-bounding volumes,[3] similarly padding the view frustum, or even eroding occluders. However, these fixes come with their own processing overhead, which can make eliminating all sources of artifacts practically impossible.

[2]Predicated rendering is one indirect and limited alternative on Xbox 360.

[3]A more accurate extrusion should take into account rotational as well as spatial velocity, as with continuous collision detection [Redon et al. 02].

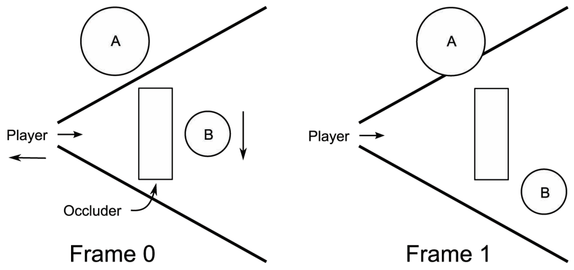

Figure 1.1. Camera or object movement can lead to popping with latent queries.

Sudden changes such as camera cuts are also problematic with latent queries, potentially leading to either a visibility or processing spike [Hastings 07] to avoid rampant popping. As such, it may be preferable to simply skip rendering for a frame and only process visibility updates behind the scenes.

1.3.4 GPU Overhead

The GPU is a precious resource and a common bottleneck in games, as we seek to maximize visual fidelity for a given target frame rate. Therefore, a visibility solution that relies heavily on GPU processing is less than ideal, particularly on modern consoles where multiple CPU cores or SPUs are available. While it is true to say that OQs should be issued only when there is an overall GPU saving [Soininen 08], this cannot be guaranteed in general and we would ideally like to dedicate as much GPU muscle as possible to direct rendering.

1.3.5 Variable Costs

A final disadvantage with OQs is that the cost of testing an object is roughly proportional to its size on screen, which typically does not reflect its true rendering cost. While one can, for instance, choose to always render objects with high screen-space coverage to avoid this penalty, it is a less viable strategy when working with a hierarchy.

Even if one develops a more sophisticated oracle [Bittner et al. 09] to normalize performance, this can come at the cost of reduced culling effectiveness. Furthermore, a hierarchy requires additional CPU overhead when objects move, or parts become visible or occluded. As with queries, the per-frame update cost can be bounded by distributing work over multiple frames, but this can similarly compromise culling.

Ideally we would like to avoid these kinds of unfortunate trade-offs, especially when major changes to the environment occur; although leveraging coherency can be a great way to reduce the average per-frame processing time, it should not exacerbate worst-case performance!

1.4 Wish List

Ideally we would like to take the strengths of OQs but reduce or eliminate the negatives. Here is our wish list:

These first items are already taken into account with OQs:

- no precomputation
- general applicability
- cccluder fusion

Here is a list of improvements we would like to achieve:

- low latency
- reduced CPU-GPU dependency
- no reliance on coherency
- bounded, high performance
- simple, unified solution

In summary, we would like to be able to handle a wide range of dynamic scenes with the minimum of fuss and no less than great performance. Essentially, we *want it all* and in the case of *Splinter Cell Conviction*—as you will now learn—we wanted it *yesterday*!

1.5 *Conviction* Solution

One of the initial technical goals of *Splinter Cell Conviction* was to support dense environments with plenty of clutter and where, in some situations, only localized occlusion could be exploited.

We initially switched from PVS visibility to OQs because of these requirements, but having battled for a long time with the drawbacks outlined earlier, and becoming increasingly frustrated by mounting implementation complexity, hacks, and failed work-arounds, we started to look for alternatives. Unfortunately, by this point we had little time and few resources to dedicate to switching solutions yet again.

Luckily for us, [Shopf et al. 08] provided a guiding light, by demonstrating that the hierarchical Z-buffer (HZB) [Greene et al. 93] could be implemented efficiently on modern GPUs—albeit via DX10—as part of an AMD demo. The demo largely validated that the HZB was a viable option for games, whereas we had previously been skeptical, even with a previous proof of concept by [Décoret 05].

Most importantly, it immediately addressed all of our requirements, particularly with respect to implementation simplicity and bounded performance. In fact, the elegance of this approach cannot be understated, comparing favorably with the illusory simplicity of OQs, but without any of the associated limitations or management complexity in practice.

1.5.1 The Process

The steps of the process are detailed here.

Render occluder depth. As with OQs, we first render the depth of a subset of the scene, this time to a render target texture, which will later be used for visibility testing, but in a slightly different way than before.

For *Conviction*, these occluders were typically artist authored[4] for performance reasons, although any object could be optionally flagged as an occluder by an artist.

Create a depth hierarchy. The resulting depth buffer is then used to create a depth hierarchy or z-pyramid, as in [Greene et al. 93]. This step is analogous to generating a mipmap chain for a texture, but instead of successive, weighted down-sampling from each level to the next, we take the maximum depth of sets of four texels to form each new texel, as in Figure 1.2.

This step also takes place on the GPU, as a series of quad passes, reading from one level and writing to the next. To simplify the process, we restrict the visibility resolution to a power of two, in order to avoid the additional logic of [Shopf et al. 08]. Figure 1.3 shows an example HZB generated in this way.

In practice, we render at 512×256,[5] since this seems to strike a good balance between accuracy and speed. This could theoretically result in false occlusion for objects of 2×2 pixels or less at native resolution, but since we contribution-cull small objects anyway, this has not proven to be a problem for us.

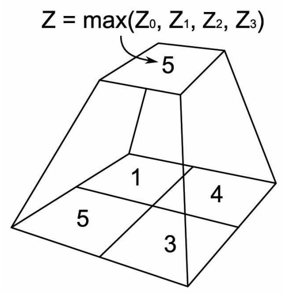

Figure 1.2. Generating successive levels of the HZB.

[4]These are often a simplified version of the union of several adjoining, structural meshes.
[5]This is approximately a quarter of the resolution of our main camera in single-player mode.

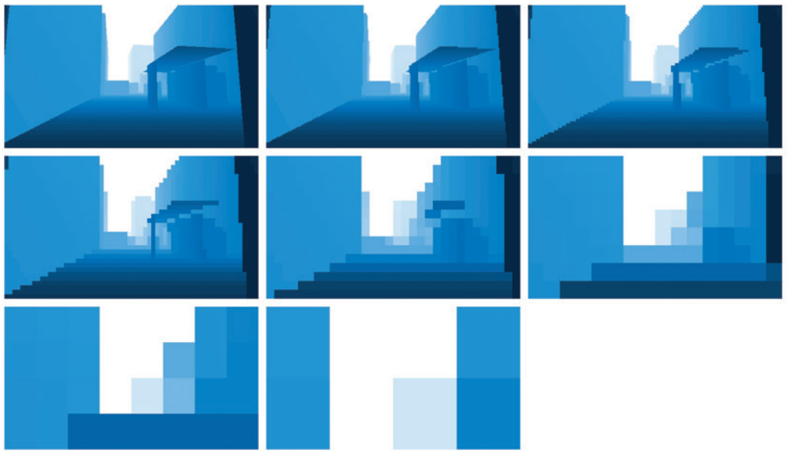

Figure 1.3. The resulting depth hierarchy. Note that the sky in the distance increasingly dominates at coarser levels.

Test object bounds. We pack object bounds (world-space AABBs) into a dynamic point-list vertex buffer and issue the tests as a single draw call. For each point, we determine, in the vertex shader, the screen-space extents of the object by transforming and projecting the bounds (see Figure 1.4). From this, we calculate the finest mip level of the hierarchy that covers these extents with a fixed number of texels or fewer and also the minimum, projected depth of the object (see Listing 1.1).

```
//Contains the dimensions of the viewport.
//In this case x = 512, y = 256
float2 cViewport;

OUTPUT main(INPUT input)
{
    OUTPUT output;

    bool visible = !FrustumCull(input.center, input.extents);

    // Transform/project AABB to screen-space
    float min_z;
    float4 sbox;
    GetScreenBounds(input.center, input.extents, min_z, sbox);

    // Calculate HZB level
    float4 sbox_vp = sbox*cViewport.xyxy;
    float2 size = sbox_vp.zw - sbox_vp.xy;
    float level = ceil(log2(max(size.x, size.y)));
    output.pos = input.pos;
    output.sbox = sbox;
    output.data = float4(level, min_z, visible, 0);

    return output;
}
```

Listing 1.1. HZB query vertex shader.

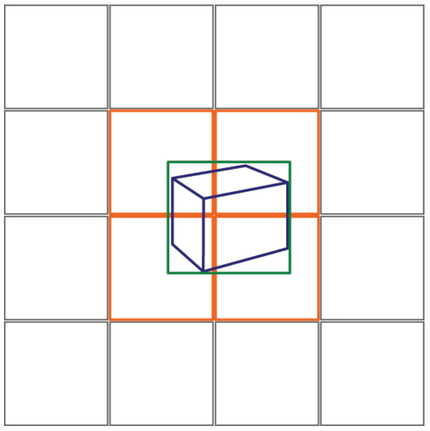

Figure 1.4. The object's world-space AABB (blue), screen extents (green) and overlapping HZB texels (orange).

This depth, plus the UVs (sbox is the screen-space AABB) and mip level for HZB lookup are then passed to the pixel shader. Here we test for visibility by comparing the depth against the overlapping HZB texels and write out 1 or 0 as appropriate (see Listing 1.2).

```
sampler2D sHZB : register(s0);

float4 main(INPUT input) : COLOR0
{
    float4 sbox = input.sbox;
    float level = input.data.x;
    float min_z = input.data.y;
    bool visible = input.data.z;

    float4 samples;
    samples.x = tex2Dlod(sHZB, float4(sbox.xy, 0, level)).x;
    samples.y = tex2Dlod(sHZB, float4(sbox.zy, 0, level)).x;
    samples.z = tex2Dlod(sHZB, float4(sbox.xw, 0, level)).x;
    samples.w = tex2Dlod(sHZB, float4(sbox.zw, 0, level)).x;

    float max_z = max4(samples);

    // Modulate culling with depth test result
    visible *= min_z <= max_z;

    return visible;
}
```

Listing 1.2. HZB query pixel shader.

In practice, we chose to use 4×4 HZB depth comparisons in contrast to the simpler example code above, since this balanced ALU instructions and texture lookups on the Xbox 360.

Also note that we perform world-space frustum testing and screen-bound generation separately. While the two can be combined as in [Blinn 96], we found that we got better code generation by performing them separately and could add additional planes to the frustum test when processing visibility for mirrors.

It is also possible to improve on the mip level selection for situations when an object covers fewer texels.

Process the results. Finally, the results are read back to the CPU via MemExport on Xbox 360. On PC, under DX9, we instead emulate DX10 stream-out by rendering with a point size of one to an off-screen render-target, followed by a copy to system memory via `GetRenderTargetData`.

1.5.2 Tradeoffs

By using a fixed number of lookups instead of rasterization, the performance of the visibility tests is highly predictable for a given number of objects. That said, this bounded performance comes at the cost of reduced accuracy for objects that are large on screen.

On the other hand, this approach can be viewed as probabilistic: large objects are, on average, more likely to be visible anyway, so performing more work (in the form of rasterization with OQs) is counter-productive. Instead, with HZB testing, accuracy is distributed proportionally. This proved to be a particularly good fit for us, given that we wanted a lot of relatively small clutter objects, for which instancing was not appropriate for various reasons.

We also benefited from the high granularity afforded by a query per object, whereas wholly OQ-based methods require some degree of aggregation in order to be efficient, leading to reduced accuracy and more variable performance. This became clear in our own analysis when we switched to HZB visibility from OQs. We started off with a 2×2 depth-test configuration, and even that out-performed hand-placed occlusion query volumes, both in terms of performance and amount of culling. Essentially, what we lost in terms of occlusion accuracy, we gained back in being able to test objects individually.

Point rendering with a vertex buffer was chosen primarily for ease of development because vertex buffers offered the convenience of heterogeneous data structures. However, a more efficient option could be to render a single quad and fetch object information from one or more textures instead. Not only would this ensure better pixel-quad utilization on some hardware, but it would also play to the strength of GPUs with a nonunified shader architecture such as the PS3's RSX, where the bulk of the shader hardware is dedicated to pixel processing.

1.5.3 Performance

Table 1.1 represents typical numbers seen in PIX on Xbox 360, for a single camera with around 22000 objects, all of which are processed in each each frame.

Pass	Time (ms)
Occlusion	0.06
Resolve	0.04
HZB Generation	0.10
HZB Queries	0.32
Total	**0.52**

Table 1.1. Performance timings.

1.5.4 Extensions

Once you have a system like this in place, it becomes easy to piggy-back related work that could otherwise take up significant CPU time compared with the GPU, which barely breaks a sweat. Contribution fading/culling, texture streaming and LOD selection, for instance, can all be determined based on each object's screen extents,[6] with results returned in additional bits.

On Xbox 360, we can also bin objects into multiple tiles ourselves, thereby avoiding the added complexity and restrictions that come with using the predicated tiling API, not to mention the extra latency and memory overhead when double-buffering the command buffer.

Finally, there is no reason to limit visibility processing to meshes. We also test and cull lights, particle systems, ambient occlusion volumes [Hill 10], and dynamic decals.

[6]We choose to use the object's bounding sphere for rotational invariance.

1.5.5 Shadow Caster Culling

We also extend our system to accelerate shadow-map rendering, with a two-pass technique initially inspired by [Lloyd et al. 04], but with a more straightforward approach. For instance, we do not slice up the view frustum and test subregions as they do. This is primarily because we are not using shadow volumes for rendering and therefore are not aiming to minimize fill-rate[7]—only the number of casters—for CPU and vertex transform savings. Development time and ease of GPU implementation are also factors.

In the first pass, we test caster visibility from the light's point of view, in exactly the same way that we do for a regular camera: via another HZB. If a given caster is visible, we write out the active shaft bounds, which are formed from the 2D light-space extents, the caster's minimum depth, and the maximum depth from the HZB (see Listing 1.3), otherwise it is culled as before:

```
float3 shaft_min = float3(input.sbox.xy, min_z)
float3 shaft_max = float3(input.sbox.zw, max_z)
```

Figure 1.5 shows this in action for a parallel light source. Here, caster C is fully behind an occluder,[8] so it can be culled away since it will not contribute to the shadow map.

In the second pass, we transform these shafts into camera space and test their visibility from the player's point of view via the existing player camera HZB—again just like regular objects. Here, since the shafts of A and B have been clamped to the occluder underneath, they are not visible either.

```
// Use the lower level if we only touch <= 2 texels
// in both dimensions

float level_new = max(level - 1, 0);
float2 scale = pow(2, -level_new);

float2 a = floor(sbox_vp.xy*scale);
float2 b = ceil(sbox_vp.zw*scale);

float2 dims = b - a;

if (dims.x <= 2 && dims.y <= 2)
    level = level_new;
```

Listing 1.3. HZB level refinement.

[7]But we could adapt this type of testing to cull more. See Section 1.7.
[8]Occluders used for shadow culling always cast shadows.

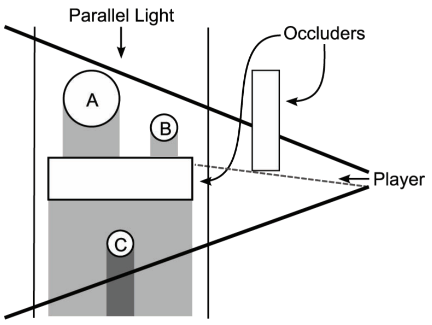

Figure 1.5. Two-pass shadow visibility.

Conceptually, we are exploiting the redundancy of shadow-volume overlap across two viewpoints in order to reduce our set of potential casters.

1.5.6 Summary

To reiterate, this entire process takes place as a series of GPU passes; the CPU is involved only in dispatching the draw calls and processing the results at the end.

In retrospect, a CPU solution could have also worked well as an alternative, but we found the small amount of extra GPU processing to be well within our budget. Additionally, we were able to leverage fixed-function rasterization hardware, stream processing, and a mature HLSL compiler, all with literally man-years of optimization effort behind them. In contrast to the simple shaders listed earlier, a hand-optimized VMX software rasterizer would have taken significantly longer to develop and would have been harder to extend.

If you already have a PVS or portal visibility system, there can still be significant benefits to performing HZB processing as an additional step. In the first place, either system can act as an initial high-level cull, thus reducing the number of HZB queries. In the case of portals, the "narrow-phase" subfrusta testing could also be shifted to the GPU. Indeed, from our own experience, moving basic frustum testing to the GPU alone was a significant performance improvement over VMX tests on the CPU. Finally, in the case of BSP-based PVS, the faces could be preconverted to a number of large-scale occluders for direct rendering.

1.6 *Battlefield* Solution

When developing the game (*Battlefield: Bad Company 1)* using our new in-house Frostbite engine for the first time, we knew that we needed a solution for removing objects occluded by others. We discussed many methods, but it all came down to a list of things that we wanted the system to have:

- must be fully dynamic, since the environment—both objects and terrain—can deform
- low GPU overhead
- results accessible from the CPU, so we can skip updating certain aspects of occluded objects, such as animation

After reading about Warhawk's approach [Woodard 07] based on software rasterization on SPUs, we decided to try a similar approach since we had spare processing power available on the CPU side. The resulting implementation was subsequently rolled out across all of our target platforms (PlayStation 3, Xbox 360, and PC), but we will focus in particular on the details of the PS3 version.

At a high level, the steps involved are very similar to those used for *Conviction*: a software occlusion rasterizer renders low polygon meshes to a z-buffer, against which occluders are tested to determine if they are visible or not.

In reality, the work is broken down into a number of stages, which are job-scheduled in turn across several SPUs. We will now describe these in detail.

1.6.1 The Process

Occluder triangle setup. This stage goes through all occluders in the world space (a flat array) in preparation for rasterization:

1. Each job grabs a mesh from the array using `InterlockedIncrement`.
2. The job checks if the mesh is inside the frustum. If it is not, it continues to the next one (Step 1).
3. If the mesh is fully inside the frustum, its triangles are immediately appended to an output array (also interlocked and shared between the jobs).
4. If the mesh was not fully inside, its triangles are clipped before being added.

Terrain triangle setup. This is effectively the same as the previous stage, except that it generates and adds conservative triangles for the terrain[9] to the array.

Occluder render. This is the stage that actually rasterizes the triangles. Each SPU job has its own z-buffer (256×114) and grabs 16 triangles at a time from the triangle array generated previously.

When the jobs are finished getting triangles from the triangle array, they will each try to lock a shared mutex. The first one will simply DMA its z-buffer to main memory, unlock the mutex, and exit so that the next job can start running.

As the mutex gets unlocked, the next job will now merge its own buffer with the one in main memory and send back the result, and so on. (*Note*: There are

[9] As the terrain can deform, these must be regenerated.

several ways to improve on this and make it faster. We could, for example, DMA directly from each SPU.)

Frustum cull. This stage performs frustum versus sphere/bounding box (BB) checks on all meshes in the world—typically between 10,000 and 15,000—and builds an array for the next stage. The implementation traverses a tree of spheres (prebuilt by our pipeline) and at each leaf we do bounding-box testing if the sphere is not fully inside.

Occlusion cull. Finally, this is where visibility testing against the z-buffer happens. We first project the bounding box of the mesh to screen-space and calculate its 2D area. If this is smaller than a certain value—determined on a per-mesh basis—it will be immediately discarded (i.e, contribution culled).

Then, for the actual test against the z-buffer, we take the minimum distance from the camera to the bounding box and compare it against the z-buffer over the whole screen-space rectangle. This falls somewhere between the approach of [Woodard 07]—which actually rasterizes occluders—and that of *Conviction* in terms of accuracy.

Performance The timings reflect best-case parallelism over five SPUs and were measured in a typical scene (see Table 1.2). In practice, workloads between SPU jobs will vary slightly and may be intermixed with other jobs, so the overall time for visibility processing will be higher in practice.

In this case we rasterized around 6000 occluder triangles (we normally observe 3000 to 5000), and performed around 3000 occlusion tests after frustum and extent culling.

Stage	**Time/SPU (ms)**
Triangle Setup	0.4
Rasterization	1.0
Frustum Cull	0.6
Occlusion Cull	0.3
Total	**2.3**

Table 1.2. Performance timings.

1.7 Future Development

1.7.1 Tools

Although artist-authored occluders are generally a good idea for performance reasons (particularly so with a software rasterizer), we encountered a couple of notable problems with this strategy on *Conviction*. First, with a large team and therefore a number of people making changes to a particular map, there were a few cases where modifications to the layout of visual meshes would not be applied to the associated occluders. Even with the blueprint of a map largely locked down, cosmetic changes sometimes introduced significant errors and these tended to occur right at the end of testing when production was most stretched!

Second, some artists had a tendency to think of modeling occluders in the same way as collision meshes—when, in fact, occluders should always be flush with, or inside of, the visual meshes they represent—or they did not feel that a small inaccuracy would be that important. This simply was not the case: time and again, testers would uncover these problems, particularly in "scope mode" where the reduced field of view can magnify these subtle differences up to half of the screen, causing large chunks of the world to disappear.

These errors would also show up as "shadow acne" due to the requirement that shadow occluders—those used for culling casters during shadow map visibility—had to cast shadows themselves. Sometimes, it would have made more sense to have just used these visual meshes directly as occluders, instead of creating separate occluder meshes.

Though checks can be added in the editor to uncover a lot of these issues, another option could be to automatically weld together, simplify, and chunk up existing visual meshes flagged by artists.

At the root of it all, the primary concern is correctness; there is no such thing as "pretty looking" visibility, so one could argue that it is not the best use of an artist's time to be modeling occluders if we can generate them automatically for the most part, particularly if a human element can introduce errors. This is definitely something we would like to put to the test, going forward.

1.7.2 Optimizations

One trivial optimization for the GPU solution would be to add a pre-pass, testing a coarse subdivision of the scene (e.g., regular grid) to perform an earlier, high-level cull—just like in *Battlefield*, but using the occlusion system too. We chose not to do this since performance was already within our budget, but it would certainly allow the approach to scale up to larger environments (e.g., "open world").

Additionally, a less accurate *object-level* pre-pass (for instance, four HZB samples using the bounding sphere, as with [Shopf et al. 08]) could lead to a speed up wherever there is a reasonable amount of occlusion (which by necessity is a common case). Equally, a finer-grained final pass (e.g., 8×8 HZB samples) could improve culling of larger occluders.

In a similar vein, another easy win for the SPU version would be using a hierarchical z-buffer either for early rejection or as a replacement for a complete loop over the screen bounds. As earlier numbers showed, however, the main hotspot performance-wise is occluder rasterization. In that instance we might

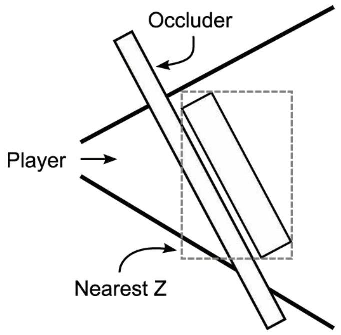

Figure 1.6. A single screen-space z-value for occluders can lead to conservative acceptance in some cases.

gain again, this time from *hierarchical* rasterization as in [Abrash 09], although at the cost of increased implementation complexity. Frustum culling could also be sped up by switching to a different data structure (e.g., grid) to improve load balancing on SPUs as well as memory access patterns.

Although the accuracy-performance trade-off from the HZB was almost always beneficial for *Conviction*, we did encounter a couple of instances where we could have profited from better culling of large, structural geometry. We believe that the biggest factor here was the lack of varying z over the occluder (see Figure 1.6) when testing against the HZB, not the number of tests (beyond 4×4) or the base resolution.

On Xbox 360, we investigated hardware-rasterizing occluder bounds as a proof-of-concept for overcoming this, but we ran out of time and there were some performance pitfalls with MemExport. We hope to pick up where we left off in the future.

Conviction's shadow-caster culling proved to be a significant optimization for cascaded shadow maps. One potential avenue of future development would be to try to adapt the idea of frustum subdivision coupled with caster-receiver intersection testing, as presented in [Diamand 10], with similarities to [Lloyd et al. 04]. [Eisemann and Décoret 06] and [Décoret 05] also build on the latter.

We would also like to extend culling to local shadow lights. As we already cache casters per shadow light (the cache is updated on object or light movement), we could directly evaluate shadow visibility for this subset of the scene. This would avoid the higher fixed overhead of processing all objects in the map as we do for the main view or shadow cascades, which is important since we can have up to eight active shadow lights per camera. These updates could happen either every frame or whenever the list changes.

1.7.3 Future Hardware

The jury is out on exactly what sort of future we face when it comes to the convergence of increasingly multi-core CPUs and more programmable GPUs and when, or indeed if, it will happen. Larrabee is an interesting example, showing that even fixed-function rasterization hardware is potentially on the way out [Abrash 09] and, while a CPU solution could be considered a safe long-term bet, the most efficient method going forward may be closer to the way hardware works than a traditional scan-line approach.

[Andersson 10] describes two possible future scenarios for visibility processing: either a progression of the GPU approach we already described, but with lower latency, or having the ability for the GPU to feed itself commands. A killer application for the latter could be shadow-map rendering, where visibility (as earlier) and subsequent draw calls would happen entirely on the GPU, thereby avoiding any CPU synchronization, processing, and dispatch. This is almost possible today and potentially so on current consoles, but existing APIs are a roadblock.

1.7.4 General Observations

In *Conviction*, although arbitrary occlusion tests could be issued by the main thread (to accelerate other systems, in much the same way as in *Battlefield*), we had to restrict their use in the end due to the need for deterministic behavior during co-operative play. This was primarily an issue for PC as we could not ensure matching results between GPUs from different IHVs, or indeed across generations from the same vendor. For the next title, we hope to find other applications for exploiting our system so this will not be a problem.

Were we to generate a min/max depth hierarchy, we could also return more information about the state of occlusion, which may open up more applications. By testing the z-range of objects, we can determine one or more states: Completely visible or occluded (all tests pass conclusively), partially occluded (tests pass conclusively as fully visible or occluded), potentially occluded (some tests are inclusive, i.e., z-range overlap with the HZB).

1.8 Conclusion

Whatever the future, experimenting with solutions like these is a good investment; in our experience, we gained significantly from employing these fast yet straightforward visibility systems, both in development and production terms.

The GPU implementation in particular is trivial to add (demonstrated by the fact that our initial version was developed and integrated in a matter of days) and comes with a very reasonable overhead.

1.9 Acknowledgments

We would like to thank Don Williamson, Steven Tovey, Nick Darnell, Christian Desautels, and Brian Karis, for their insightful feedback and correspondence, as well as the authors of all cited papers and presentations, for considerable inspiration.

Bibliography

[Abrash 96] Michael Abrash. "Inside Quake: Visible-Surface Determination." *Dr. Dobb's Sourcebook* Jan/Feb (1996), 41–45.

[Abrash 09] Michael Abrash. "Rasterization on Larrabee." In *Game Deveoper's Conference*, 2009.

[Akenine-Möller et al. 08] Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. *Real-Time Rendering*, Third edition. Natick, MA: A K Peters, 2008.

[Andersson 10] Johan Andersson. "Parallel Futures of a Game Engine v2.0." In *STHLM Game Developer Forum*, 2010.

[Bittner et al. 09] Jiří Bittner, Oliver Mattausch, and Michael Wimmer. "Game Engine Friendly Occlusion Culling." In *ShaderX*^{7}, pp. 637–653. Hingham, MA: Charles River Media, 2009.

[Blinn 96] Jim Blinn. "Calculating Screen Coverage." *IEEE CG&A* 16:3 (1996), 84–88.

[Décoret 05] Xavier Décoret. "N-Buffers for Efficient Depth Map Query." *Computer Graphics Forum (Eurographics)* 24:3 (2005), 8 pp.

[Diamand 10] Ben Diamand. "Shadows in *God of War III*." In *Game Developer's Conference*, 2010.

[Eisemann and Décoret 06] Elmar Eisemann and Xavier Décoret. "Fast Scene Voxelization and Applications." In *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games, I3D '06*, pp. 71–78. New York: ACM, 2006.

[Greene et al. 93] Ned Greene, Michael Kass, and Gavin Miller. "Hierarchical Z-buffer Visibility." In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '93*, pp. 231–238. New York: ACM, 1993.

[Hastings 07] Al Hastings. "Occlusion Systems." http://www.insomniacgames.com/research_dev/articles/2007/1500779, 2007.

[Hill 10] Stephen Hill. "Rendering with Conviction." In *Game Developer's Conference*, 2010.

[Lloyd et al. 04] Brandon Lloyd, Jeremy Wendt, Naga Govindaraju, and Dinesh Manocha. "CC Shadow Volumes." In *ACM SIGGRAPH 2004 Sketches, SIGGRAPH '04*, p. 146. New York: ACM, 2004.

[Redon et al. 02] Stephane Redon, Abderrahmane Kheddar, and Sabine Coquillart. "Fast Continuous Collision Detection between Rigid Bodies." *Computer Graphics Forum* 21:3 (2002), 279–288.

[Shopf et al. 08] Jeremy Shopf, Joshua Barczak, Christopher Oat, and Natalya Tatarchuk. In *ACM SIGGRAPH 2008 Classes*, *SIGGRAPH '08*, pp. 52–101. New York: ACM, 2008.

[Soininen 08] Teppo Soininen. "Visibility Optimization for Games." Gamefest, 2008. Microsoft Download Center, Available at http://www.microsoft.com/downloads/en/details.aspx?FamilyId=B9B33C7D-5CFE-4893-A877-5F0880322AA0&displaylang=en, 2008.

[Woodard 07] Bruce Woodard. "SPU Occlusion Culling." In *SCEA PS3 Graphics Seminar*, 2007.