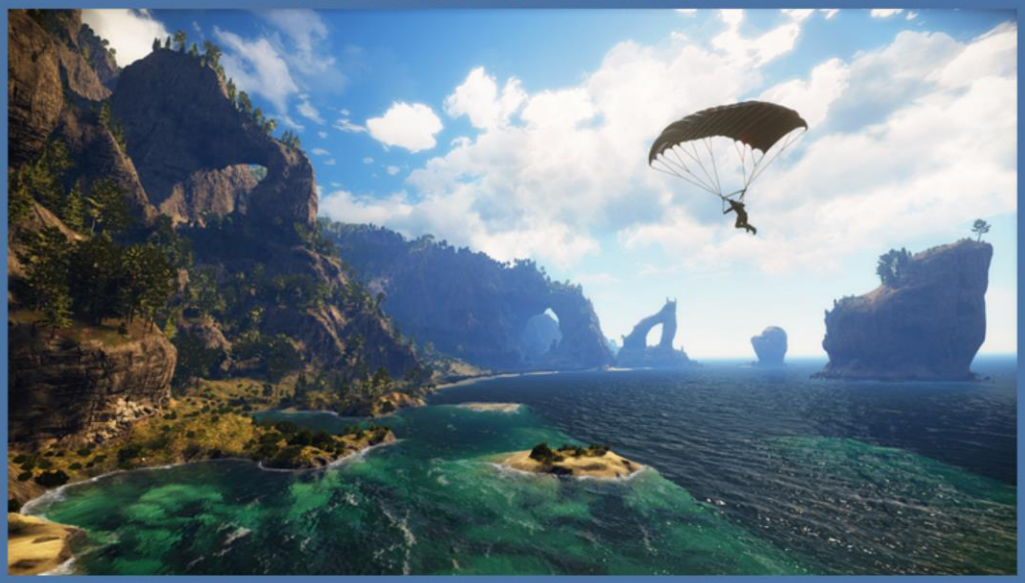

SIGGRAPH2015
Xroads of Discovery


SIGGRAPH2015
Xroads of Discovery
The 42nd International Conference and Exhibition
on Computer Graphics and Interactive Techniques
AVALANCHE STUDIOS
Practical Clustered Shading
Emil Persson
Avalanche Studios

What's new?

- Preaching the Clustered gospel for 2.5 years
  - Nordic Game 2013
  - SIGGRAPH 2013
  - CEDEC 2013
  - SIGGRAPH Asia 2014
  - SIGGRAPH 2015
- Avalanche Studios still using it in production
  - Will ship in **Just Cause 3**
  - Very happy with it
  - All old lighting paths removed

What's new?



What's new?

- Interest in Clustered Shading increasing
  - Shipped in Forza Horizon 2
  - Intel samples for PC and Android [Intel 14]
  - AMD talk at GDC [Thomas 15]
  - Gets name-dropped a lot by game developers
    - Although few have actually implemented it yet

Agenda

- The old stuff
  - History of lighting in the Avalanche Engine
  - Why Clustered Shading?
  - Adaptations for the Avalanche Engine
  - Performance
  - Future work
- The new stuff
  - New research and insights
  - Alternative implementations

I have split the talk into separate sections, representing slightly updated content that has been previously presented, and a new section containing all new material.

The old stuff

Lighting in Avalanche Engine

- Just Cause 1
  - Forward rendering
  - 3 global pointlights
- Just Cause 2, Renegade Ops
  - Forward rendering
  - World-space XZ-tiled light-indexing [Persson 10]
    - 4 lights per 4m x 4m tile
    - 128x128 RGBA8 light index texture
    - Lights in constant registers (PC/Xenon) or 1D texture (PS3)
  - Per-object lighting
  - Customs solutions

The first Just Cause had 3 global pointlights. This meant that if, for instance, three streetlights were enabled and you fired your gun, one of the lights would shut off for the duration of the gun flash. Clearly, this solution was hardly ideal.

For Just Cause 2 we switched to a world-space 2D tiled solution where light indexes were stored in texels. The technique has been described in detail in the article ”Making it Large, Beautiful, Fast, and Consistent: Lessons Learned Developing Just Cause 2” in GPU Pro. This technique was actually in some ways similar to clustered shading, although much more limited and designed around DX9 level hardware. It worked reasonably well on platforms with decent dynamic branching, such as PC and Xenon, whereas the PS3 struggled. Ultimately this caused us to implement numerous workarounds to get PS3 running well, so that in the end this technique mostly ended up being a fallback option if the light count was too high for a specialized shader to work. The amount of specialized shaders also became quite a bit of a maintenance problems, and figuring out the light count a performance issue on the CPU side.

Lighting in Avalanche Engine

- Mad Max
  - Classic deferred shading
    - 3 G-Buffers
    - Flexible lighting setup
      - Point lights
      - Spot lights
        - » Optional shadow caster
        - » Optional projected texture
    - HDR
  - Transparency problematic
    - Solved by not really using any transparency
  - FXAA for anti-aliasing

After Just Cause 2 we ended going the deferred shading route, initially using classic deferred. This worked relatively well for last generation console hardware and allowed us to support many more lights, different light types, shadow casting dynamic lights etc. This was great, but naturally we also got all the downsides of deferred shading, such as problems with transparency, problems with custom material or lighting models, as well as large increase in memory consumption. Initially we supported MSAA, but ultimately we dropped it in favor of FXAA for performance and memory reasons.

Unfortunately, the old forward pass also had to stick around for transparency to work to some extent, although it only ever supported pointlights and the lighting didn't quite match the much more sophisticated deferred pass. For Mad Max we ultimately moved away from fully supporting transparency with lighting because of its problems with deferred, plus that the game environment has very little need for transparency anyway beyond particle effects. But for other projects where transparency might be desirable we started looking into alternatives, especially with a new generation consoles on the horizon at the time.

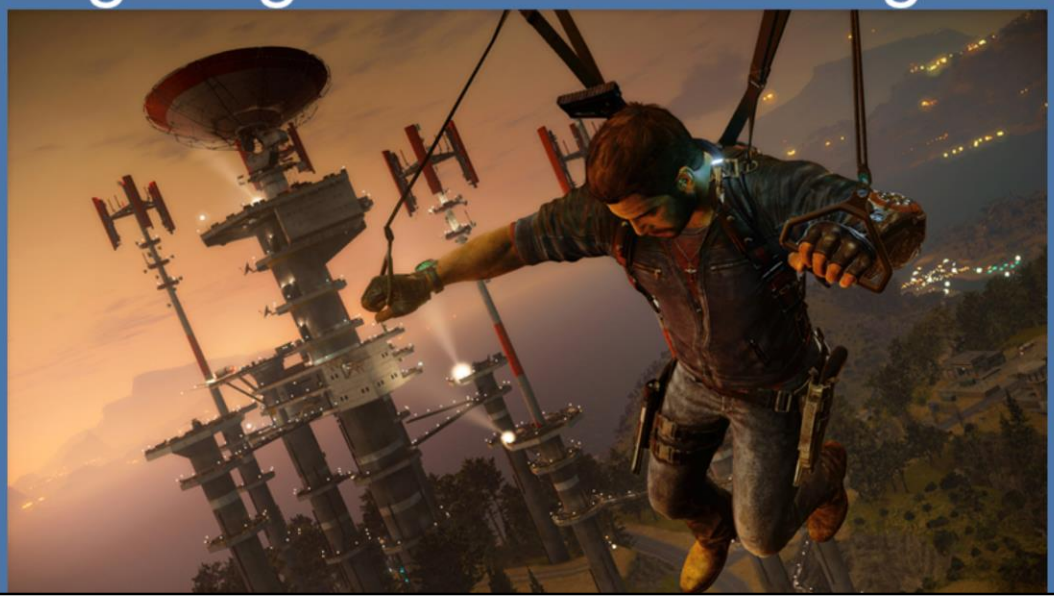
Lighting in Avalanche Engine

- Just Cause 3
  - Clustered deferred shading
    - 4 G-Buffers
    - Flexible lighting setup
    - Physically Based Lighting
  - Transparency with lighting just works
    - Really needed for this kind of game
    - Enables the use of Wire AA [Persson 12]

For Just Cause 3, which was next-gen/PC only from the beginning, we went with Clustered Shading as our main lighting solution. For this kind of game it really wasn't a feasible solution to limit the options for transparency beyond what we had in earlier games. We are still using deferred, but with clustered shading we can use the same lighting data for doing lighting in the forward passes for transparent objects.

A nice bonus of having properly working transparency with lighting is that we could now use the Wire AA technique we invented. It mostly just worked out of the box for Just Cause 3.

Lighting in Avalanche Engine

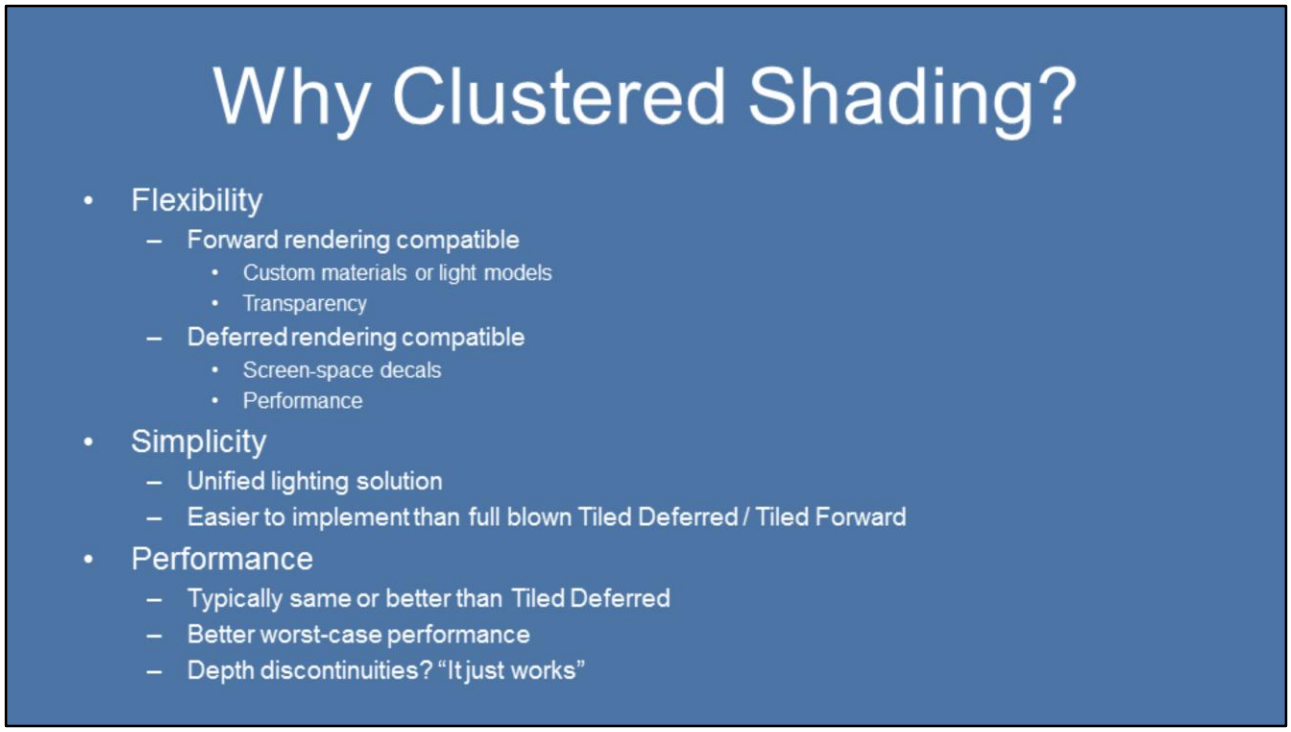


Here's an illustration of what sort of lighting we have in the game. This screenshot is actually just one of the official screenshots, but it does a decent job at showing the different ranges and scales of lighting we support in the game. We have everything from the little light in the foreground on the shoulder of the main character, to the large number of light sources on the military structure in the background (all those white dots are actual light sources), to the distant lights in the upper-left corner of the image, which are also actual light sources in the game data, but not loaded at this distance and only visualized with point sprites to breath life into the game world at large distances.

Tiled Deferred Shading and Forward+ (Tiled Forward Shading) are production proven and have shipped in real games, but they come with a bunch of drawbacks. Tiled deferred offers better performance than classic deferred, but doesn't really solve any of our problems since all drawbacks of classic deferred stays around. In addition, it also imposes a new restriction in that all lights, and consequently shadow buffers, are now required up-front. However, this is a property it shares with all other techniques, including Forward+ and Clustered Shading, and even our old forward solution from JC2.

Tiled Forward (a.k.a. Forward+) has the advantage of working well with MSAA without hassles; however, while it can be made to work with transparency, it requires an extra pass, including another round of pre-z. The requirement of a full pre-z pass for this technique to work made this a non-starter for us. We didn't bother implementing it for evaluation purposes as a full pre-z pass is not an option for us. We did at one point have a fairly complete pre-z pass in Just Cause 2, but over the development the pre-z pass was continuously trimmed until very little remained. The additional overhead just didn't pay off, and the large increase in draw-call count was problematic. After we got a decent occlusion culling system in place there were very few cases pre-z did not, in fact, result in a performance drop. Pre-z is now only enabled on a handful of things specifically marked for pre-z by content creators, and a few code-driven systems that need it for other reasons.

Clustered Shading has the advantage of not requiring a pre-z pass, even in its forward incarnation, while working well with MSAA and transparency out of the box with no particular tricks or hacks. While Avalanche Studios pioneered this technique for the games industry, it actually ended up first shipping in Forza Horizon 2. When we first started out exploring this technique it wasn't production proven, but at this point it is. At Avalanche Studios we have used it in production since January 2013 and by the end of 2015 it will ship in Just Cause 3.



Clustered Shading is really decoupled from the choice between deferred or forward rendering. It works with both, so you're not locked into one or the other. This way you can make an informed choice between the two approaches based on other factors, such as whether you need custom materials and lighting models, or need deferred effects such as screen-space decals, or simply based on performance.

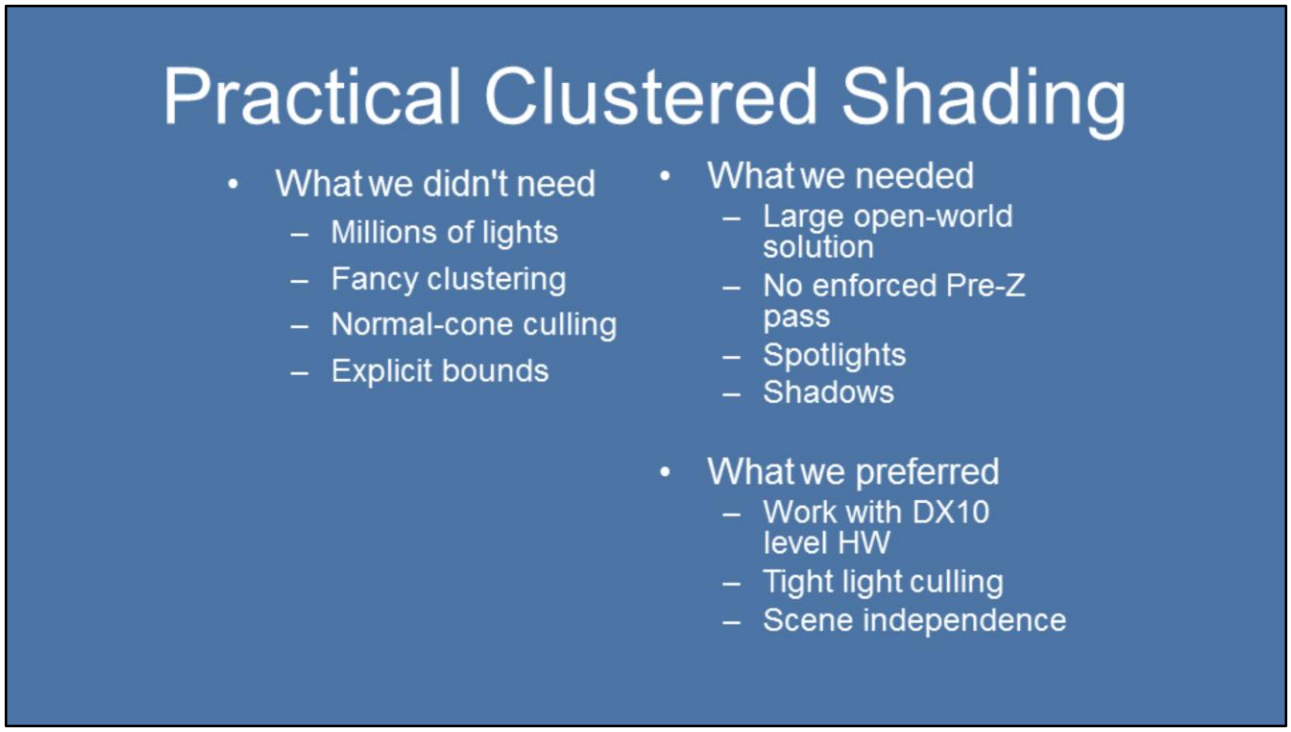
The two tiled solutions need quite a bit of massaging to work reasonable well in all situations, especially with large amounts of depth discontinuities. There are proposed solutions that mitigate the problem, such as 2.5D culling, but they further complicate the code. For Clustered Shading it just falls out automatically and depth discontinuities do not cause performance problems. This allows Clustered Shading to maintain a more stable frame-rate regardless of scene depth complexity.

I will illustrate the point using a random screenshot from Just Cause 3. Now this isn't a hand picked screenshot to show off the worst case, in fact, I wasn't able to pick a screenshot myself. This was hand-picked by marketing for being awesome. But even so, it's representative of what you can expect in the Just Cause series and really shows that this is a real problem in real games, and certainly so in the games that we make.

Depth discontinuities

Here a number of large depth differences have been manually painted over the image to illustrate where you might expect a problem for tiled shading techniques. As you can see, they are fairly common and affect a fairly large part of the screen. One source of pain that's not too well illustrated here, is vegetation, which tends to create lots of nasty depth discontinuities. There are some forests here, but for ground level gameplay you can certainly expect much more of that problem.

Now, if you thought the previous image looked bad, now put that into the context of an actual tiled setup. Here I have illustrated all the tiles that would be affected by the problem, and as you can see, a quite large percentage of the screen suffers from suboptimal lighting from depth discontinuities.

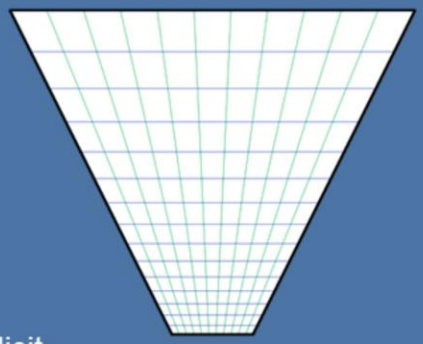


The original paper [Olsson et. al 12] was written by academics, and naturally the direction of their research doesn't match 100% with the requirements of the a game engine. We don't have millions of tiny lights, but between hundreds and thousands of mostly artist placed lights, that are on a human scale. This meant that tight culling, so as to not add lights to more clusters than necessary, became more important to us. The higher-order clustering options the paper explored (and also largely rejected) were also something that we didn't expect to work for us. Deriving the explicit cluster bounds was something that could be interesting, but we found that sticking to implicit bounds simplified the technique, while also allowing the light assignment to run on the CPU. This enables DX10 level GPU compatibility, which initially felt like a good idea, but at this point doesn't add much value to us since we are firmly stuck in DX11 land anyway for other techniques we are using. But the important point is that gives us scene independence. This means that we don't need to know what the scene looks like to fill in the clusters, and this also allows us to evaluate light at any given point in space, even if it's floating in thin air. This could be relevant for instance for ray-marching effects.

The paper only explored pointlights, whereas we need spotlights as well. We also needed a shadow solution, which the original paper also did not explore. However, Olsson et. al. has since continued their research and have now an interesting shadow approach made for clustered shading. We have however stuck with our own simpler approach. Finally, our games are massively large while still being played on human scale, resulting in a depth span from very near to very far, which required some extra fiddling to get rolling with clustered shading.

The Avalanche solution

- Still a deferred shading engine
  - But unified lighting solution with forward passes
- Only spatial clustering
  - 64x64 pixels, 16 depth slices
- CPU light assignment
  - Works on DX10 HW
  - Compact memory structure easy
- Implicit cluster bounds only
  - Scene-independent
  - Deferred pass could potentially use explicit



We are still using a deferred engine, but we could change to forward at any time should we decide that to be better. The important part is, however, that the transparency passes can now use the same lighting structure as the deferred passes, making it a unified lighting solution. Since we are still using deferred, and thus obviously have a complete depth buffer once we get to the deferred lighting pass, we could potentially use explicit bounds there. We still haven't explored that opportunity, but it's an option. It's unclear if computing the explicit bounds, plus an extra round of culling, is going to be outweighed by potentially faster light evaluation.

Currently we are using 64x64 screen-space tiles, and 16 depth slices. This is most likely going to change, primarily because currently the tiles are currently fairly long and thin, and this is not optimal for a culling, in particular for spotlights. We have been experimenting with other setups, such as 128x128 and 32 depth slices. This created more cubical shaped clusters and helped with culling, which helped with culling, especially for spotlights. Another option we have considered, but not yet explored, is to not base it on pixel count, but simply divide the screen into a specific number of tiles regardless of resolution. This may reduce coherency on the GPU side somewhat in some cases, but would also decouple the CPU workload from the GPU workload and allow for some useful CPU side optimizations if the tile counts are known at compile time.

The Avalanche solution

- Exponential depth slicing
  - Huge depth range! [0.1m – 50,000m]
  - Default list
    - [0.1, 0.23, 0.52, 1.2, 2.7, 6.0, 14, 31, 71, 161, 365, 828, 1880, 4270, 9696, 22018, 50000]
    - Poor utilization
  - Limit far to 500
    - We have a "distant lights" systems for light visualization beyond that
    - [0.1, 0.17, 0.29, 0.49, 0.84, 1.43, 2.44, 4.15, 7.07, 12.0, 20.5, 35, 59, 101, 172, 293, 500]
  - Special near 0.1 – 5.0 cluster
    - Tweaked visually from player standing on flat ground
    - [0.1, 5.0, 6.8, 9.2, 12.6, 17.1, 23.2, 31.5, 42.9, 58.3, 79.2, 108, 146, 199, 271, 368, 500]

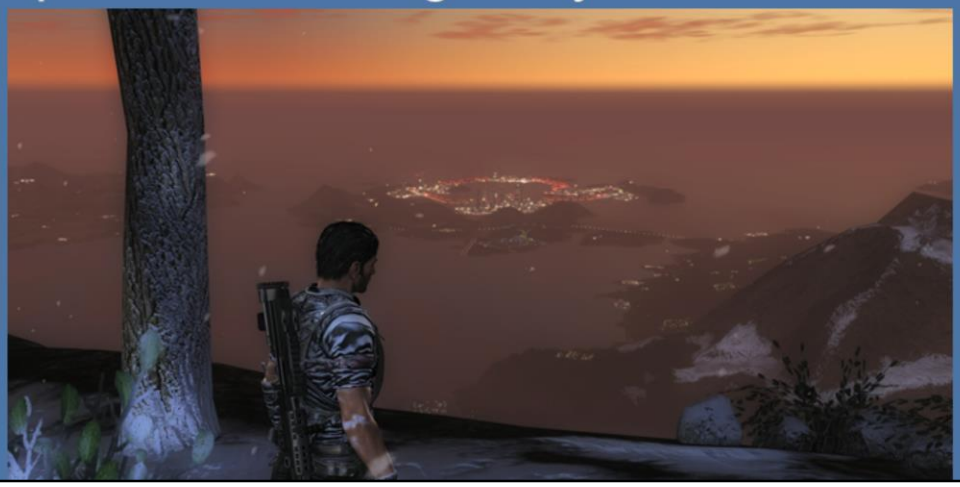
We are using exponential depth slicing, much like in the paper. There is nothing dictating that this is what we have to use, or for that matter that it is the best or most optimal depth slicing strategy; however, the advantage is that the shape of the clusters remain the same as we go deeper into the depth. On the other hand, clusters get larger in world space, which could potentially result in some distant clusters containing a much larger amount of lights. Depending on the game, it may be worth exploring other options.

Our biggest problem was that our depth ratio is massive, with near plane as close as 0.1m and far plane way out on the other side of the map, at 50,000m. This resulted in poor utilization of our limited depth slices, currently 16 of them. The step from one slice to the next is very large. Fortunately, in our game we don't have any actual light sources beyond a distance of 500m. So we simply decided to keep our current distant light system for distances beyond 500m and limit the far range for clustering to that.

This improved the situation notably, but was still not ideal. We still burnt half of our slices on the first 7 meters from the camera. Given how our typical scenes look like, that's likely going to be mostly empty space in most situations. So to improve the situation, we made the first slice special and made that go from near plane to an arbitrary visually tweaked distance, currently 5m. This gave us much better utilization.

The Avalanche solution

- Separate distant lights system



This illustrates our distant light system, which has been around since Just Cause 2. In this screenshot there are likely no actual lights enabled since we're far from civilization on top of a mountain, except perhaps our fake "night light" that slightly illuminates the area around the player at night to help game-play a bit in the darkness. Everything in the distance though, while representing actual artist placed lights, the actual light sources aren't loaded at this distance. They are simply stored as a very compact list of point sprites, resident in memory at all time, and which is very cheap to render. We are at this point still using the same forward rendering solution here as in Just Cause 2, but one option now that we are using deferred is to actually compute real lighting under those sprites instead of just a putting a blob from a texture there.

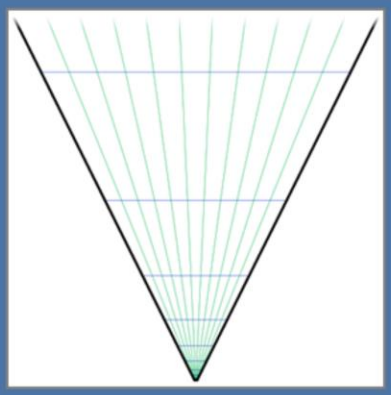
The Avalanche solution

Here the same system can be seen in effect in Just Cause 3. To the right of Rico's arm and in the upper right corner we have instances of the distant light system, but also if you look down into the fog in a bottom left corner.

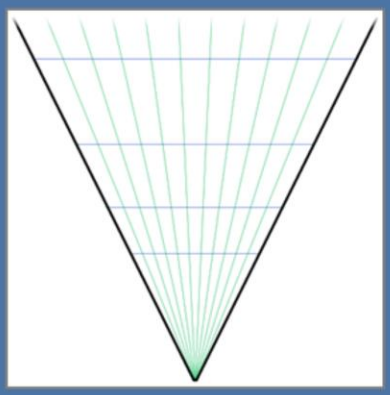
And in this screenshot you can also see the effect as a few lights in the distance in the upper center part of the screen.

The Avalanche solution

- Default exponential



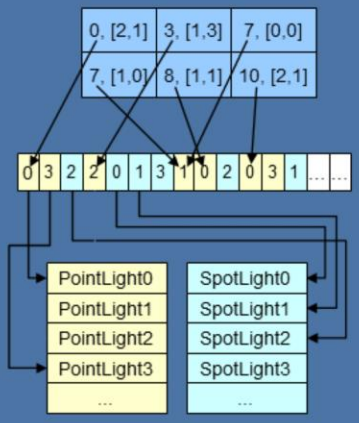
- Special near cluster



This illustrates the benefit of the special near cluster. Fewer slices are wasted, and the cluster shapes aren't quite as long and thin.

Data structure

- Cluster “pointers” in 3D texture
  - R32G32_UINT
    - R=Offset
    - G=[PointLightCount, SpotLightCount]
- Light index list in texture buffer
  - R16_UINT
  - Tightly packed
- Light & shadow data in CB
  - PointLight: 3 x float4
  - SpotLight: 4 x float4



Given a screen position and a depth value (whether from a depth buffer or the rasterized depth in a forward pass) we start by looking up the cluster from a 3D texture. Each texel represents a cluster and its light list. The red channel gives us an offset to where the light list starts, whereas the green channel contains the light counts. The light lists are then stored in a tightly packed lists of indexes to the lights. The actual light source data is stored as arrays in a constant buffer.

All in all the data structure is very compact. In a typical artists lit scene it may be around 50-100kb of data to upload to the GPU every frame.

Shader

```
int3 tex_coord = int3(In.Position.xy, 0);                // Screen-space position ...
float depth = Depth.Load(tex_coord);                     // ... and depth

int slice = int(max(log2(depth * ZParam.x + ZParam.y) * scale + bias, 0)); // Look up cluster
int4 cluster_coord = int4(tex_coord >> 6, slice, 0);     // TILE_SIZE = 64

uint2 light_data = LightLookup.Load(cluster_coord);      // Fetch light list
uint light_index = light_data.x;                         // Extract parameters
const uint point_light_count = light_data.y & 0xFFFF;
const uint spot_light_count  = light_data.y >> 16;

for (uint pl = 0; pl < point_light_count; pl++) {        // Point lights
    uint index = LightIndices[light_index++].x;

    float3 LightPos = PointLights[index].xyz;
    float3 Color    = PointLights[index + 1].rgb;
    // Compute pointlight here ...
}

for (uint sl = 0; sl < spot_light_count; sl++) {         // Spot lights
    uint index = LightIndices[light_index++].x;

    float3 LightPos = SpotLights[index].xyz;
    float3 Color    = SpotLights[index + 1].rgb;
    // Compute spotlight here ...
}
```

This shows the shader code for rendering with this data structure. The input is just the screen-space position and depth. This shows a deferred pass where depth comes from a texture, but in a forward pass the second line of code would simply use In.Position.z instead. Everything else would be identical, which shows how easily this technique adapts to either deferred or forward.

The ZParam.xy here contains the same parameters that you would use to compute a linear depth from a Z-buffer value, except I eliminated the division since that just becomes a negative under the logarithm, i.e. log2(1/(z*a+b)) = log2(z*(-a)+(-b)).

Data structure

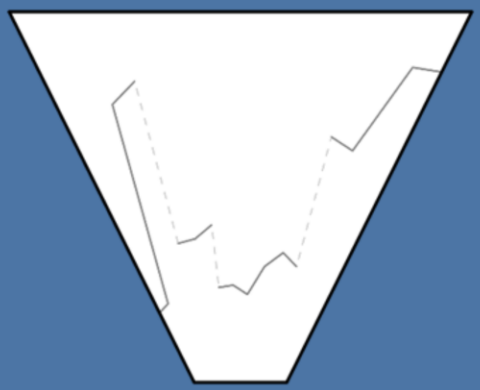
- Memory optimization
  - Naive approach: Allocate theoretical max
    - All clusters address all lights
      - Not likely
    - Might be several megabytes
    - Most never used
  - Semi-conservative approach
    - Construct massive worst-case scenario
      - Multiply by 2, or what makes you comfortable
      - Still likely only a small fraction of theoretical max
  - Assert at runtime that you never go over allocation
    - Warn if you ever get close

The light list could theoretically become huge. Say you have a total of 30*17*16 clusters at 1080p, and allow up to 256 lights per cluster, that would need 4MB, which with double-buffering (because it's updated from the CPU) means you'll need 8MB. Perhaps not a problem on next-gen, but hardly ideal, and who knows how many times these numbers will be bumped before you ship.

Normally, not every light affects every cluster in a scene. In fact, it's extremely rare that you get even remotely close to that. So we constructed a somewhat plausible worst-case scenario with loads of large lights jammed in front of the player and recorded the max utilization ever encountered. Then multiplied up that for some extra margin. Even after that, the resulting buffer size we needed to allocate was far smaller. Naturally though, if you go down this path, it's clearly important to add runtime assertions and warnings to make sure you don't ever go above what you actually have allocated. Done correctly, at worst you would have artifacts for that extreme frame where a thousand nukes blew up in the player's face.

Clustering and depth

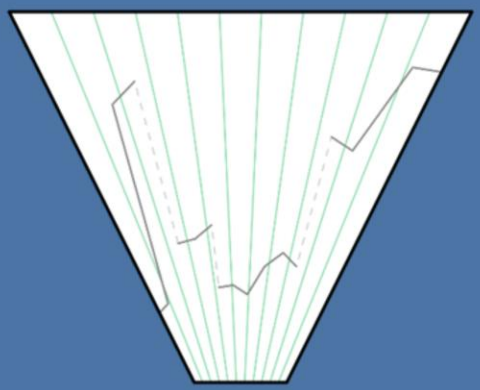
- Sample frustum with depths



Let's discuss the problem of depth discontinuities and illustrate how clustered shading solves it. Here's a sample frustum with some depth values, including a few discontinuities.

Clustering and depth

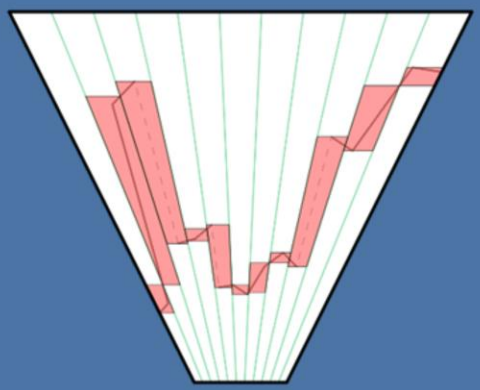
- Tiled frustum



Here we added the tiles.

Clustering and depth

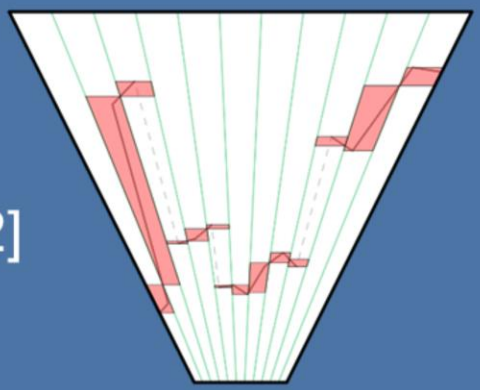
- Depth ranges for Tiled Deferred / Forward+



And this is the depth ranges you would get for a plain tiled shading algorithm. Clearly some ranges are fairly large.

Clustering and depth

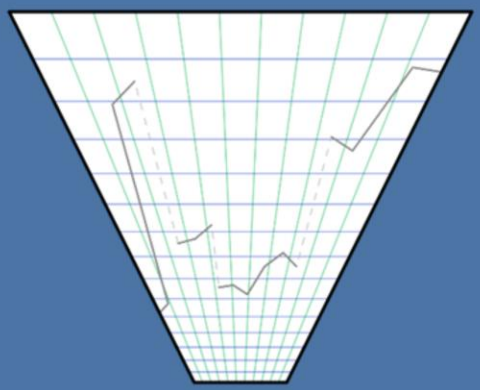
- Depth ranges for Tiled Deferred / Forward+ with 2.5D culling [Harada 12]



With 2.5D culling the situation is notably improved. Now lights in the discontinuity area are not included. However, we do pay the full cost lights at both ends for both sides of the discontinuity. Also note that one very long depth range remains. This is because it's not discontinuous, it's a continuous slope. This situation would happen if you look down a hallway, or the ground plane, or moderately large surface at a grazing angle.

Clustering and depth

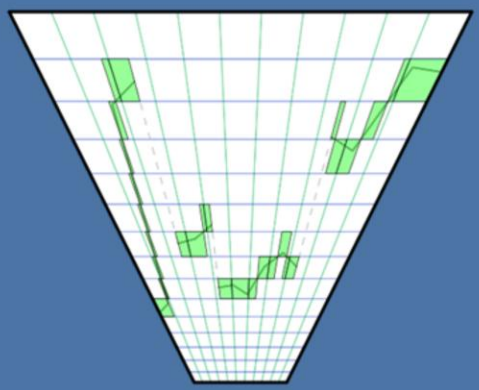
- Clustered frustum



Now let's look at a clustered frustum.

Clustering and depth

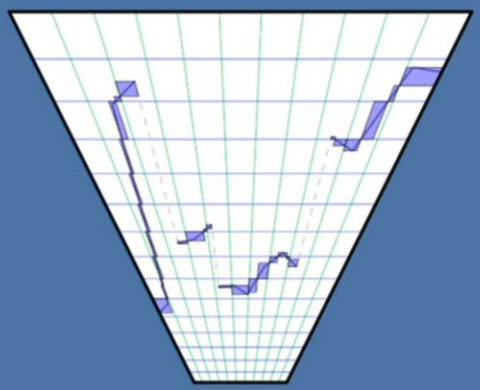
- Implicit depth ranges for clustered shading



These are the depth ranges that we will need to consider. Note that we are paying for exactly one cluster's depth at any given point.

Clustering and depth

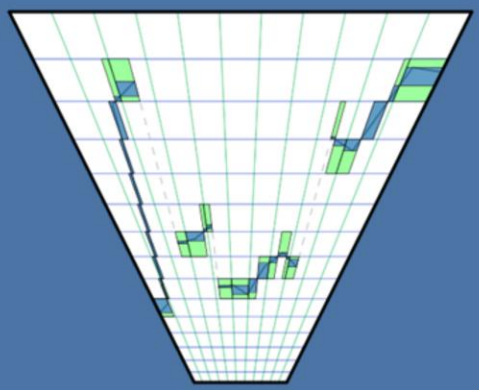
- Explicit depth ranges for clustered shading



If we go to explicit cluster bounds, the situation is even further improved, although in practice there may not be a huge difference between a fairly small range and an even smaller range, depending on the typical size of light sources.

Clustering and depth

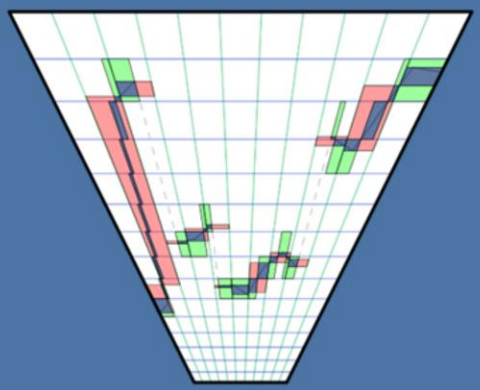
- Explicit versus implicit depth ranges



Here we see the improvement from implicit bounds to explicit.

Clustering and depth

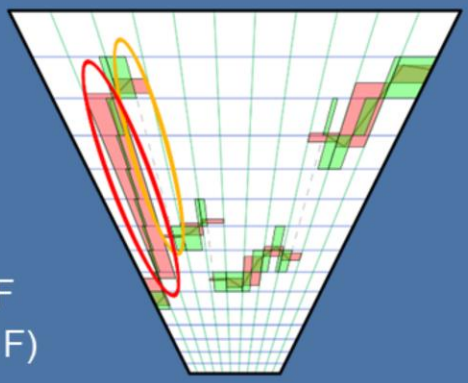
- Tiled vs. implicit vs. explicit depth ranges



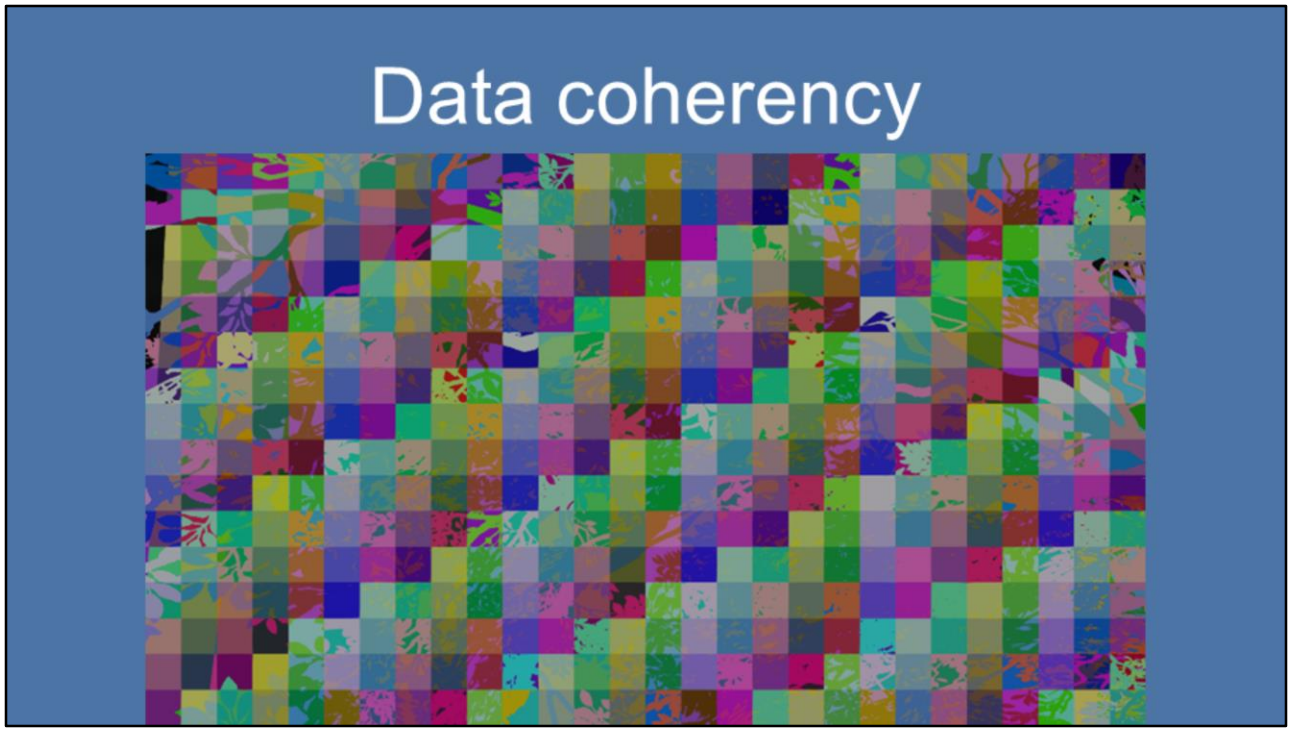
And here all techniques are compared. As you can see, explicit clustered is always the tightest. However, there are definitively areas where tiled with 2.5D culling is tighter than implicit cluster bounds. So in scenes with little depth complexity tiled could very well be faster. However, implicit clustered bounds does not have any areas that are extremely bad, regardless of depth complexity, and would thus perform more consistently. Most importantly, its worst case performance is much better than tiled.

Wide depths

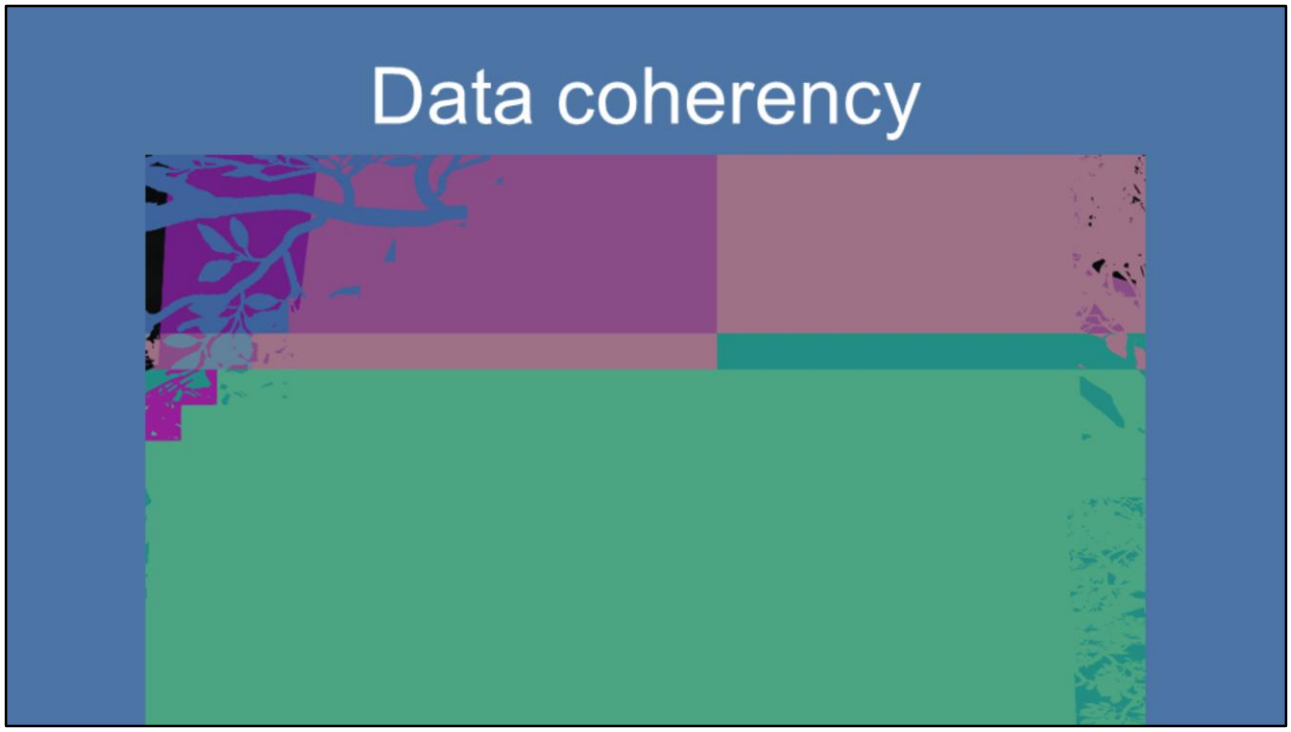
- Depth discontinuity range A to F
  - Default Tiled: A+B+C+D+E+F
  - Tiled with 2.5D: A + F
  - Clustered: ~max(A, F)
- Depth slope range A to F
  - Default Tiled: A+B+C+D+E+F
  - Tiled with 2.5D: A+B+C+D+E+F
  - Clustered: ~max(A, B, C, D, E, F)



Here we can see the impact of adding 2.5D culling to a tiled technique. While it helps in the discontinuity case (although does not reach clustered's performance), it doesn't help much or at all in a depth slope situation.



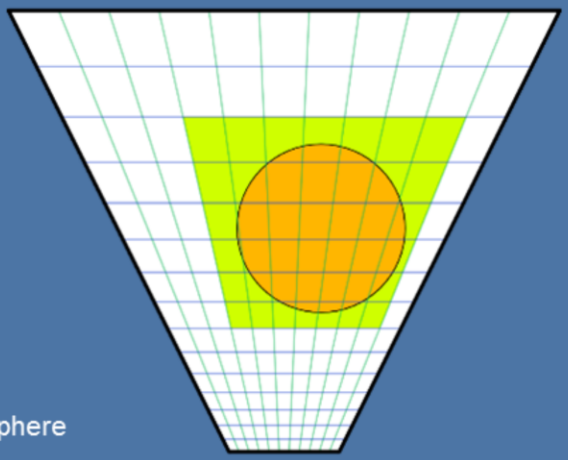
So the difference between tiled and clustered is that we pick a light list on a per-pixel basis instead of per-tile, depending on which cluster we fall within. Obviously though, in a lot of cases nearby pixels will choose the same light list, in particular neighbors within the same tile on a similar depth. If we visualize what light lists were chosen, we can see that there are a bunch of different paths taken beyond just the tile boundaries. A number of depth discontinuities from the foliage in front of the player gets clearly visible. This may seem like a big problem, but here we are only talking about fetching different data. This is not a problem for a GPU, it's something they do all the time for regular texture fetches, and this is even much lower frequency than that.



The thing you might worry about though is divergent branches. However, despite fetching different light lists from pixel to pixel, the situation is not nearly as bad as you might expect from the previous picture. Chances are that the light lists look fairly similar. If you have one light lists with 5 lights and another with 5 lights (that are not necessarily the same as the other ones), branching will still be 100% coherent. You may pay a small overhead from the ideal when the lists have different light count, but that is typically going to be a relatively small overhead. In the worst-case scenario (no coherency at all), the amount of shading essentially boils down to what tiled shading has to shade.

Culling

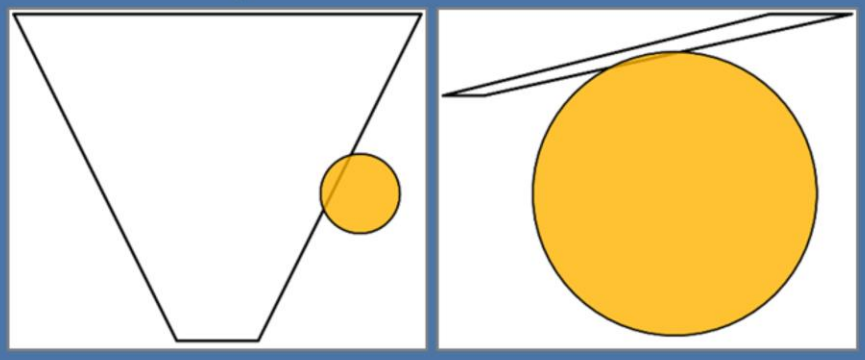
- Want to minimize false positives
- Must be conservative
  - But still tight
  - Preferably exact
    - But not too expensive
    - Surprisingly hard!
- 99% frustum culling code useless
  - Made for view-frustum culling
    - Large frustum vs. small sphere
    - We need small frustum vs. large sphere
  - Sphere vs. six planes won't do



Our light sources are typically artist placed, scaled for human environments in an outdoor world, so generally speaking from meters to tens of meters. So a light source generally intersects many clusters. The typical sphere-frustum tests that you can find online are not suitable for this sort of culling. They are made for view-frustum culling and based on the assumption that the frustum typically is much larger than the sphere, which is the opposite of what we have here. Typically they simply test sphere vs plane for each six planes of the frustum. This is conservative, but lets through spheres that aren't completely behind any of the planes, such as in the frustum corners. The result you get is that green rectangle, or essentially a "cube" of clusters around the light. But that's also the first thing we compute. We simply compute the screen-space and depth extents of the light analytically first, so this test doesn't actually help anything at all after that.

Culling

- Your mental picture of a frustum is wrong!



Most frustum culling code is written with the scenario on the left in mind. We need to handle the scenario on the right.

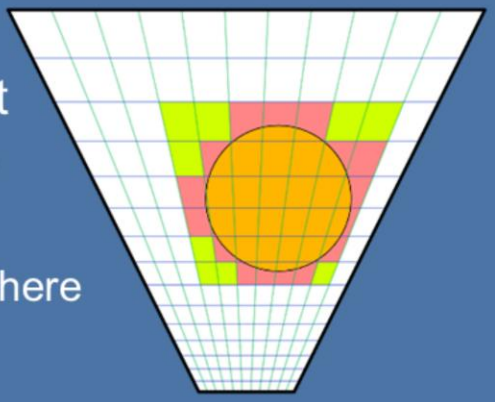
Culling

- "Fun" facts:
  - A sphere projected to screen is not a circle
  - A sphere under projection is not a sphere
  - The widest part of a sphere on screen is not aligned with its center
  - Cones (spotlights) are even harder
- Frustums are frustrating (pun intended)
- Workable solution:
  - Cull against each cluster's AABB

One way to go about frustum culling is testing all planes, all edges and all vertices. This would work, but be too costly to outweigh the gains from fewer false positives. A fast, conservative but relatively tight solution is what we are looking for. There are many approaches that seem fitting, but there are also many complications, which has ultimately thrown many of our attempts into the garbage bin. One relatively straightforward approach is to cull against the cluster's AABB. This is fast and gives fairly decent results, but it's possible to do better.

Pointlight Culling

- Our approach
  - Iterative sphere refinement
    - Loop over z, reduce sphere
    - Loop over y, reduce sphere
    - Loop over x, test against sphere
  - Culls better than AABB
    - Similar cost
    - Typically culling 20-30%

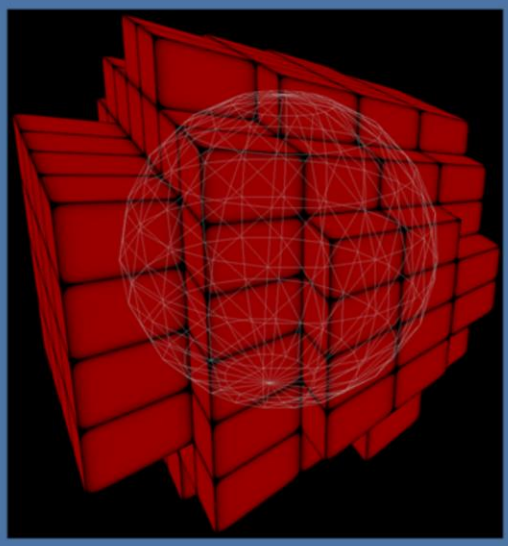


Starting with the ”cube” of clusters around the light, in our outer loop we iterate over the slices in z direction. We intersect the sphere with the slice where it is the widest. This results in a circle of a smaller radius than the original sphere, we thus continue in the y direction using a sphere of this smaller radius and the circle’s midpoint. In the center slice we simply proceed with the original sphere. We repeat this procedure in y and have an even smaller sphere. Then in the inner loop we do plane vs. sphere tests in x direction to get a strip of clusters to add the light to.

To optimize all the math we take advantage of the fact that in view-space, all planes will have components that are zero. A plane in the x direction will have zero y and offset, y direction has zero x and offset, and z-direction is basically only a z offset.

The resulting culling is somewhat tighter than a plain AABB test, and costs about the same. Where AABB culls around 15-25%, this technique culls around 20-30% from the “cube” of clusters.

Pointlight Culling



Here’s the result visualized in 3D.

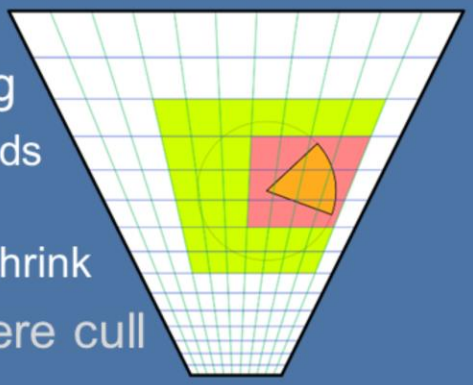
Culling pseudo-code

```
for (int z = z0; z <= z1; z++) {
    float4 z_light = light;
    if (z != center_z) {          // Use original in the middle, shrunken sphere otherwise
        const ZPlane &plane = (z < center_z)? z_planes[z + 1] : -z_planes[z];
        z_light = project_to_plane(z_light, plane);
    }
    for (int y = y0; y < y1; y++) {
        float3 y_light = z_light;
        if (y != center_y) {      // Use original in the middle, shrunken sphere otherwise
            const YPlane &plane = (y < center_y)? y_planes[y + 1] : -y_planes[y];
            y_light = project_to_plane(y_light, plane);
        }
        int x = x0;               // Scan from left until with hit the sphere
        do { ++x; } while (x < x1 && GetDistance(x_planes[x], y_light_pos) >= y_light_radius);

        int xs = x1;              // Scan from right until with hit the sphere
        do { --xs; } while (xs >= x && -GetDistance(x_planes[xs], y_light_pos) >= y_light_radius);

        for (--x; x <= xs; x++)   // Fill in the clusters in the range
            light_lists.AddPointLight(base_cluster + x, light_index);
    }
}
```

This shows the gist of the culling code.

Spotlight Culling

- Our approach
  - Iterative plane narrowing
    - Find sphere cluster bounds
    - In each six directions, do plane-cone test and shrink
  - Cone vs. bounding-sphere cull remaining "cube"

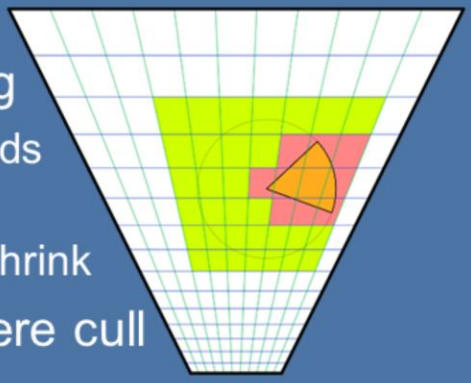


For spotlights we begin by finding the ”cube” of clusters around the light’s sphere, just like for pointlights, except this cube typically is much larger than necessary for a spotlight. However, this analytical test is cheap and goes a long way to limit the search space for following passes. Next we find a tighter ”cube” simply by scanning in all six directions, narrowing it down by doing plane-cone tests. There is likely a neat analytical solution here, but this seemed non-trivial. Given that the plane scanning works fine and is cheap we haven’t really explored that path.

Note that our cones are sphere-capped rather than flat-capped. That’s because the light attenuation is based on distance (as it should), rather than depth. Sphere-capped cones also generally behave much better for wide angles and doesn’t become extremely large as flat-capped cones can get.

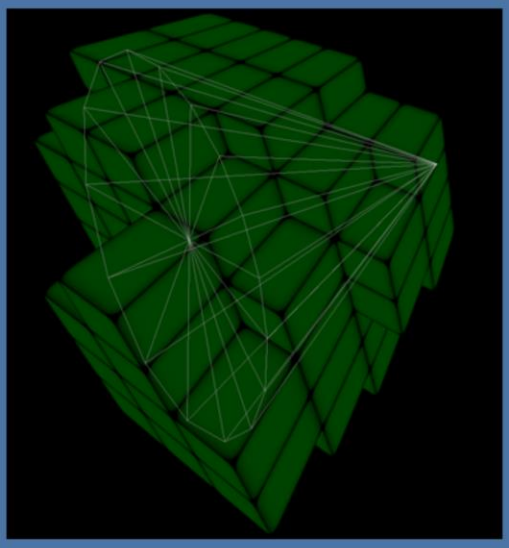
Spotlight Culling

- Our approach
  - Iterative plane narrowing
    - Find sphere cluster bounds
    - In each six directions, do plane-cone test and shrink
  - Cone vs. bounding-sphere cull remaining “cube”



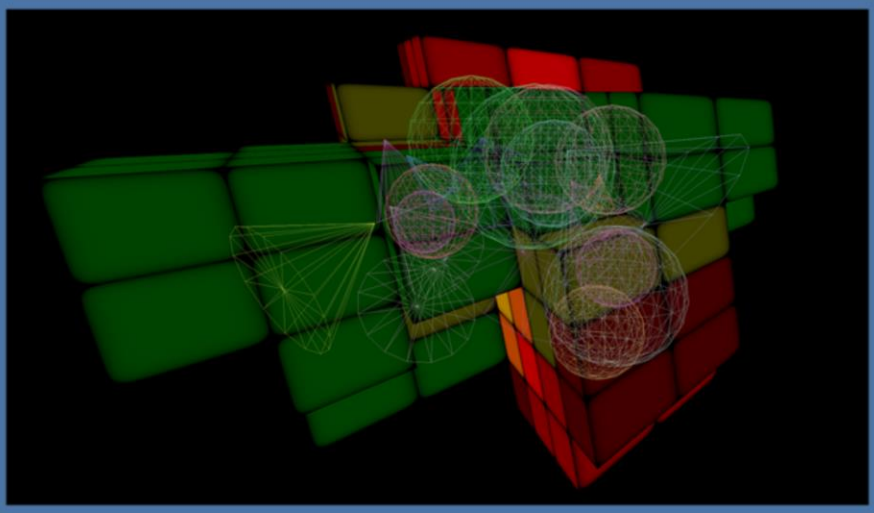
Finally, for the remaining ”cube” of clusters we cull each cluster with a sphere-capped cone vs. bounding sphere test. For this to work well we have to have relatively cubical shaped clusters, otherwise the bounding sphere becomes way oversized. Overall this technique results in a moderately tight culling that is good enough for us so far, although there is room for some minor improvement.

Spotlight Culling



Here's the result visualized in 3D. Although our spotlights are sphere-capped, our debug visualization still draws them as flat-capped. That's why it might look like it's extending a bit outside the clusters.

Pointlights and spotlights



Here's the result with a handful of pointlights and spotlights enabled in a scene. The number of pointlights goes into red, and number of spotlights into green.

Shadows

- Needs all shadow buffers upfront
  - Unlike classic deferred …
    - Memory less of a problem on next-gen
  - One large atlas
    - Variable size buffers
    - Dynamically adjustable resolution
- Lights are cheap, shadow maps are not
  - Still need to be conservative about shadow casters

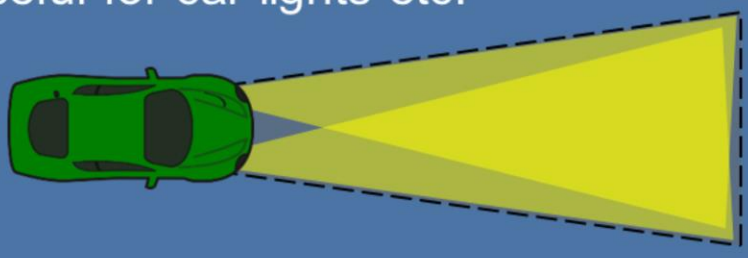
Classic deferred has the advantage that you can iterate light by light, and thus reuse resources such as shadow buffers in between. This saves some memory, which may be needed on current generation consoles. On PC and next-generation consoles this is not nearly as big a problem.

With the switch to clustered shading the cost of adding a new light to the scene is small. Artists can now be moderate ”wasteful” without causing much problems performance-wise. This is not true for rasterizing shadow buffers. They remain expensive, and relatively speaking going to be more expensive going forward since it’s often a ROP-bound process, and ROPs aren’t getting scaled up nearly as much as ALU. So we still need to be a bit conservative about how many shadow casting lights we add to the scene.

An observation that was made is that artists often place very similar looking lights close to each other. In some cases it is to get a desired profile of a light, in which case the two lights may in fact be centered at the exact same point. But often it is motivated by the real world, such as two headlights on car. Some vehicles actually have ten or more lights, all pointing in the same general direction. Rendering ten shadow buffers for that may prove to be far too expensive.

Shadows

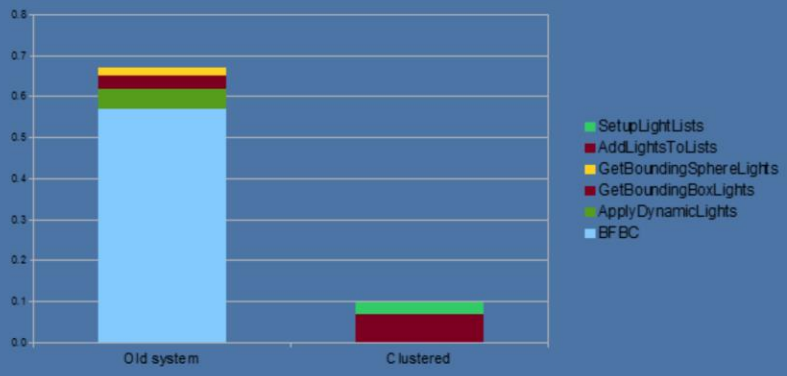
- Decouple light and shadow buffers
  - Similar lights can share shadow buffers
  - Useful for car lights etc.



Often it works just fine to share a single shadow buffer for these lights. While the shadow may be slightly off, this is usually not something that you will notice unless you are specifically looking for it. To make this work the shadow buffer is decoupled from lights and the light is assigned a shadow buffer and frustum from which to extract shadows. The shadow frustum has to be large enough to include all the different lights that uses it.

CPU Performance

- Time in milliseconds on one core. Intel Core i7-2600K.

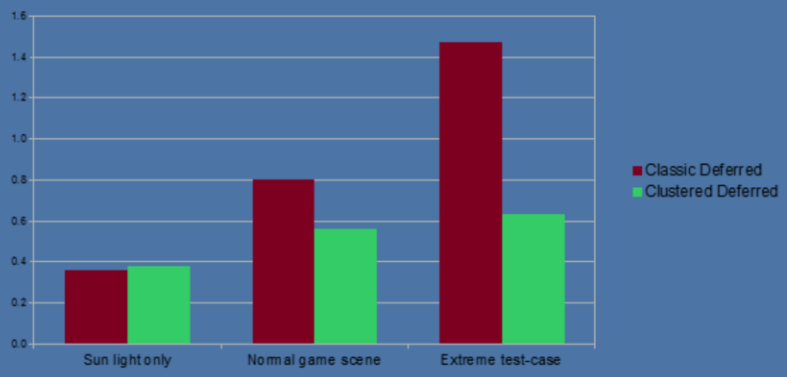


Given that we are doing the light assignment on the CPU, one may suspect that this will become a significant burden for the CPU. However, our implementation is fast enough to actually save us a bunch of CPU time over our previous solution. In a normal artist lit scene we recorded 0.1ms on one core for clustered shading. The old code supporting our previous forward pass for transparency that was still running in our system was still consuming 0.67ms for the same scene, a cost that we can now eliminate.

As of this writing, further optimizations have been made resulting in another 30-50% lower CPU cost than previously.

GPU Performance

- Time in milliseconds. Radeon HD 7970.



When we have nothing but the sun light in our scene, we incur a small overhead compared to classic deferred shading from looking up our empty light list and looping zero times. Once a light or two has been entered into the scene clustered shading is typically faster, and in regular artist lit scenes significantly so. Once we go to extreme artificial test cases with hundreds of lights sprinkled randomly in front of the player, clustered scales really well whereas classic deferred gets significantly slower. We have observed cases as large as 5x more expensive, whereas typically for heavy scenes it's around 2x. The difference is generally about how slow we can make classic deferred, rather than how fast clustered can be, as the clustered performance stays quite consistent, whereas classic's performance can very quite a lot depending on the scene.

Future work

- Clustering strategies
  - Screen-space tiles, depth vs. distance
  - View-space cascades
  - World space
    - Allows light evaluation outside of view-frustum (reflections etc.)
  - Dynamic adjustments?
- Shadows
  - Culling clusters based on max-z in shadow buffer?
- Light assignment
  - Asynchronous compute

We did some prototyping on a distance based clustering strategy instead of depth. While this allowed pointlights to be culled efficiently and exactly, this also made the cluster lookup slightly more expensive. The performance gain from an exact test was small enough that only with extreme workloads did we gain back what we lost from slower cluster lookup and we were hard-pressed to find a case where it ended up being faster in practice.

Another possible approach is clustering on view-space cascades. This would allow for exact AABB tests. One could argue that if you are going to test using an AABB, then you might just as well shape your clusters that way.

World-space clusters is another interesting option. While this would utilize the available clusters worse, the light distribution might match real world better. The other advantage is that you could evaluate light outside of the view frustum. This would allow for instance a reflection pass (such as a rear-view mirror) to use the same lighting structure for light evaluation.

There may be performance gains to be had if we consider the actual lights we have when clustering. For instance, we could tighten the cluster bounds if the most distant light active is closer than 500m, and the closest one more distant than 5m. This would allow for better cluster utilization.

Finally, a quick conservative reduction of the depth values in a shadow buffer could allow us to cull some clusters based on a conservative maximum-z value over some range. Whether this would result in any actual performance gains is unclear though.

Conclusions

- Clustered shading is practical for games
  - It's fast
  - It's flexible
  - It's simple
  - It opens up new opportunities
    - Evaluate light anywhere
    - Ray-trace your volumetric fog

What are you waiting for? Start writing your clustered shader today! ☺

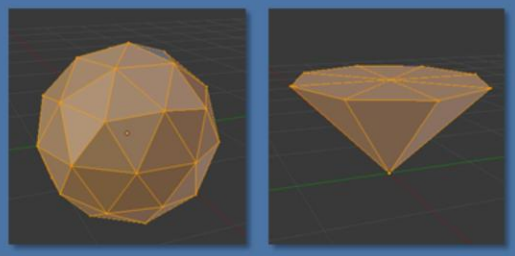
The new stuff

Research

- *Clustered shading: Assigning convex light shapes using conservative rasterization in DirectX12*
  - Research project with master thesis student
  - Improves light assignment for Clustered Shading
    - "Perfect clustering"
  - To be published in GPU Pro 7

Algorithm

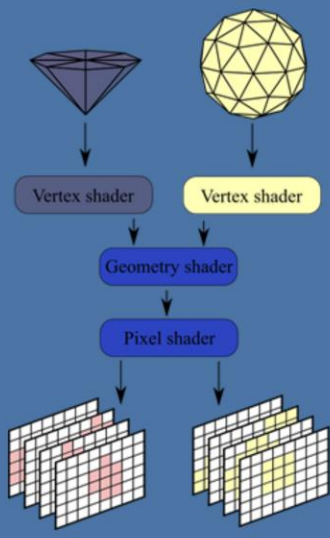
- Lights as meshes
  - Typically very low-res
  - Can be LODed
- Shell pass
  - Conservative Rasterization
- Fill pass
  - Compute shader
- Shading



The algorithm consists of two passes for the light assignment, and then of course there is a shading pass. The first pass is the shell pass. This establishes the min and max cluster slice touched by a light source within each tile. This is done by rasterizing the light volume as a low-res mesh, using conservative rasterization to make sure all relevant pixels (corresponding to tiles) are touched. The pixel shader computes the exact depth range of the triangle within each pixel, and overlap is resolved using min blending. Actually we want min blending for the lower bound, and max blending for the upper bound, but we can only use one blend mode. This is accomplished by inverting one channel, so that min blending actually picks the inverted max. Finally a fill pass is run as a compute shader. It simply collects all the depth ranges and fills in the clusters for each light.

Shell pass

- Vertex shader
  - Unit mesh
  - One draw-call per light type
- Geometry shader
  - Assigns array ID
    - Can be done in VS now though
- Pixel shader
  - Compute exact depth range
- Texture Array, e.g. 24x16xN, R8G8 format



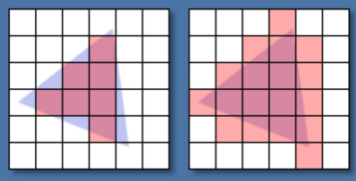
We need one draw-call per light type. So if you support say pointlights and spotlights you get a grand total of two draw-calls, using instancing.

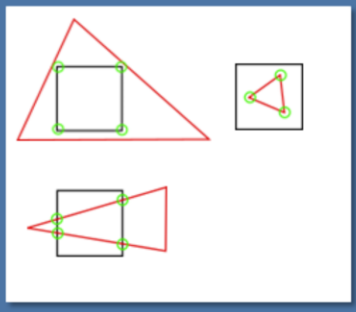
The geometry shader is only really needed for be able to assign an output texture array slice. This can actually be done using a vertex shader now in DX12, but wasn't possible at the time when this research was conducted. We expect a healthy performance boost for moving this to the vertex shader instead and dropping the geometry shader entirely.

The output is a small but deep texture array. There are probably other arrangements that would better utilize the memory, given that rendertargets tend to have hefty alignment requirements. Combined with a viewport array this could probably be improved, but we haven't had time to explore that.

Shell pass

- Conservative Rasterization
  - Touch all relevant tiles
- Compute exact depth range within pixel
  - Triangle fully covers pixel
    - Compute min & max from depth gradient
  - Pixel fully covers triangle
    - Use min & max from vertices
  - Partial coverage
    - Compute min & max at intersections
- MIN blending resolves overlap
  - Output 1-G to G to accomplish MINMAX

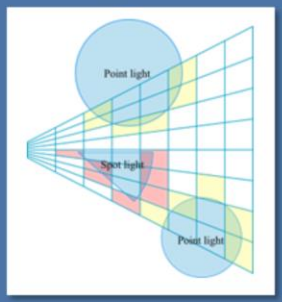


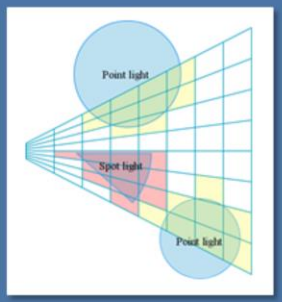


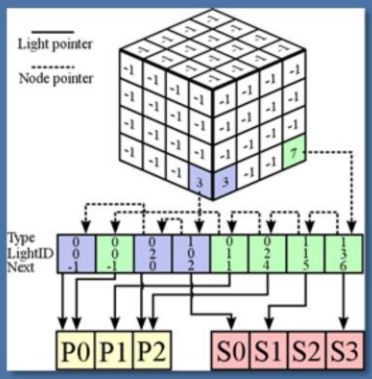
The pixel shader is relatively complicated, needing to compute the exact depth range for the intersection with the pixel, with many different cases. Fortunately, while the shader is long, it only needs to run on very few pixels.

Fill pass

- Compute shader
  - 1 thread per tile per light
  - Light linked-list



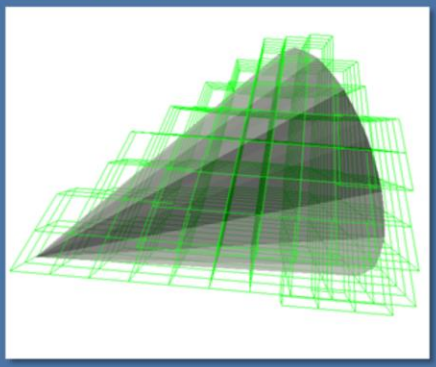




The fill pass fills out the clusters with the relevant lights. This was done using a linked list. There are probably other options that would also work, but this was never the bottleneck.

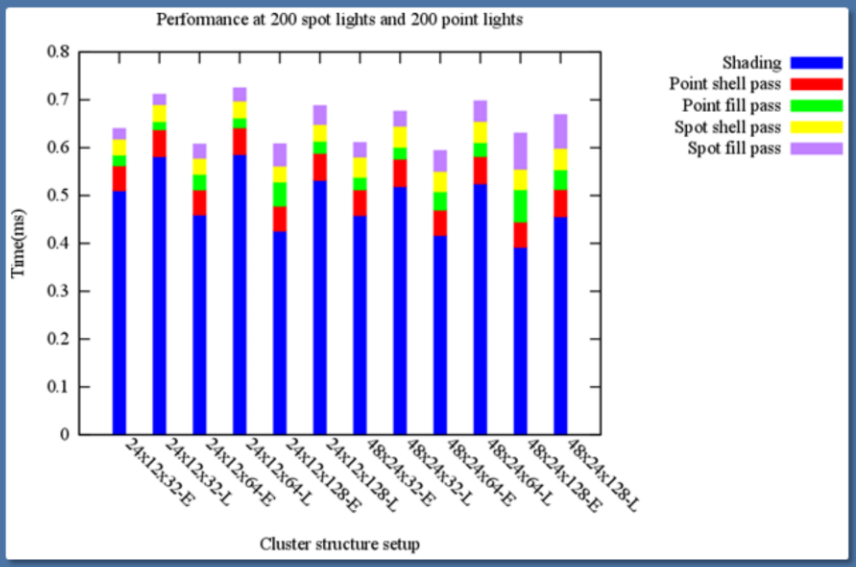
Results

- Works for any convex light type
- Tightest light assignment out there
  - Saves shading time



The main takeaway is the we can create really tight light assignment, with essentially no false positives.

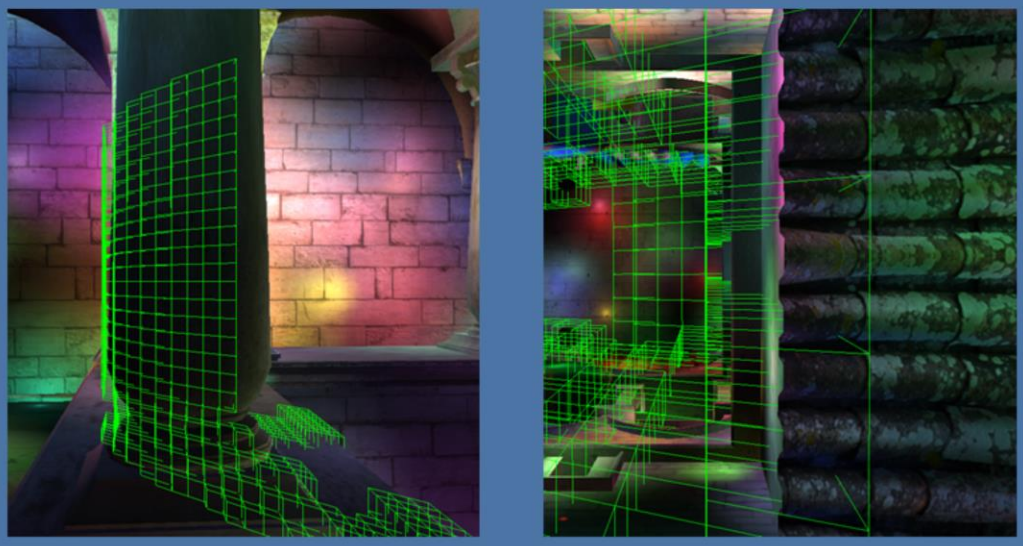
Performance



48x24, 64 slices, exponential, is fastest.

24x12, 64 slices, exponential, is best choice due to memory overhead.

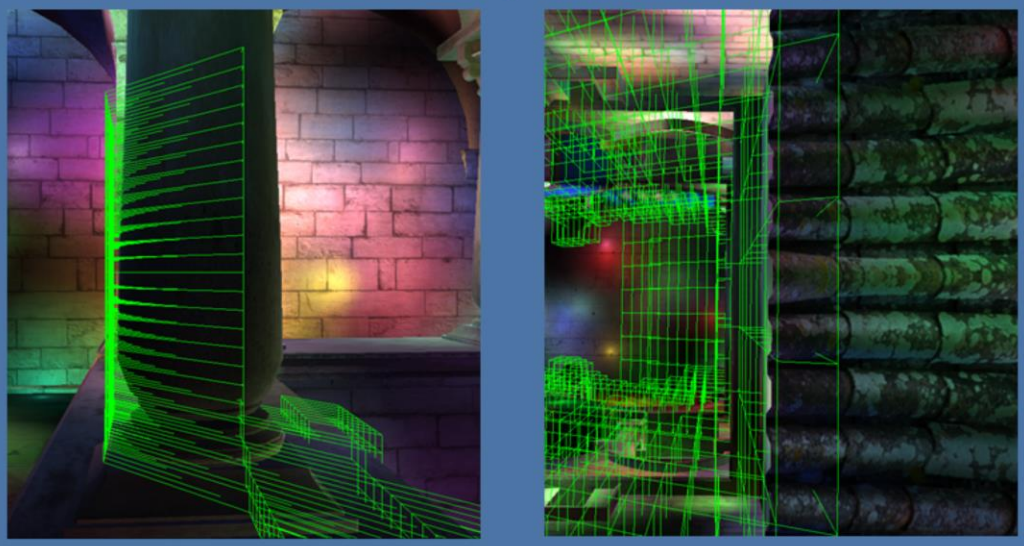
Exponential depth slicing



The other contribution of this work is a comparison between exponential depth slicing and linear. While the results are mixed, exponential tended to be the winner, but there were also definitively cases where linear came out on top. While exponential is a clear winner for large open world games with very deep depth ranges, the story may be different for say an indoor shooter, and most likely for top-down games with very narrow depth range.

In the screenshot here we can see how the clusters maintain their shape across different distances; however, close up one may argue that the clusters are wastefully small compared to the light sources, where in the distances they tend to be overly large.

Linear depth slicing



And in the linear case the depth ranges stays the same, thus keeps a constant ratio compared to the typical light sources, but on the other hand gets either very long and thin up close, or very squished in the distance.

Alternative Implementations

This is a demo application I did and released on my website (www.humus.name). It shows one way to implement Clustered Forward. It also has a reference Classic Deferred mode, which basically represents a demo I did back in 2008 and was really proud of at the time, especially with the stencil optimizations that clearly improved MSAA performance. However, 7 years later, classic deferred is showing its age. In this sample I am seeing far better performance with clustered forward, with around 5x better framerate with MSAA.

Alternative Implementations

- Alternative clustering scheme
  - World-space, fixed grid
- Alternative light list
  - Bitfield of lights
    - Single fetch
    - Constant and low memory requirements
  - Suitable with low to moderate light counts

It's important to point out that what we do in the game isn't the only way to do things, and depending on your needs you may want to explore other options. In this demo I explored some alternative ways of implementing clustered shading, in particular using a fixed world-space grid for the clusters, and representing the light lights as bitfields. It is also done with forward shading instead of deferred.

Shader

```
// Compute cluster and fetch its light mask
int4 coord = int4(In.WorldPos * Scale + Bias, 0);
uint light_mask = Clusters.Load(coord);

while (light_mask)
{
    // Extract a light from the mask and disable that bit
    uint i = firstbitlow(light_mask);
    light_mask &= ~(1 << i);

    // Lighting
    const Light light = Lights[i];
    // ...
    // ...
}
```

With a bitfield representation, a single fetch is enough, and it's really quick and easy to parse the light list. For games with only a small number of lights, this is likely the best option. Certainly for anything below 32 lights, maybe for some case up to say 128 where it's still only a single fetch, but beyond that it really depends on your situation, the overall light density and other factors.

Alternative Implementations

- Clustered lightmapping?
  - Old school lightmaps meets Clustered Shading
  - LightMap stores light bitfield per texel
  - Shadow fetched for enabled lights
    - Dead space optimization?
- Limited dynamic lighting support
  - Turn lights on/off
  - Vary light color, intensity, falloff
  - Reduce radius

Another things I've been prototyping is an old-school lightmapping approach, but for modern lighting solutions where we can't reduce all lighting into a single color value for each lightmap texel. Instead shadows are precomputed for each light, nice and soft, corresponding to you regular old-school lightmaps, but with only a shadow factor, one for each light. Then another "light map" stores a bitfield of the active lights for each texel. This can be generated from the already created shadow factor maps. At shading time the bitfield is first fetched, then lighting is computed for the enabled lights as usual, and of course, only the active lights get their shadows fetched.

This method gets all the benefits of old-school lightmaps, while supporting modern lighting features, and even including a limited amount of dynamic lightings support, despite being mostly precomputed.

Conclusions

- Lots of unexplored ground
  - Opportunities for future research
- Clustered Shading come in many flavors
  - Best choice depends on requirements

References

- [Andersson 11] DirectX 11 Rendering in Battlefield 3.
  http://dice.se/publications/directx-11-rendering-in-battlefield-3/
- [Harada et. al. 12] Forward+: Bringing Deferred Lighting to the Next Level.
  https://amd.box.com/s/9g70td498gmxz5lq8py5
- [Harada 12] A 2.5D Culling for Forward+.
  https://sites.google.com/site/takahiroharada/storage/2012SA_2.5DCulling.pdf
- [Intel 14a] Forward Clustered Shading.
  https://software.intel.com/en-us/articles/forward-clustered-shading
- [Intel 14b] Clustered Shading Android Sample.
  https://software.intel.com/en-us/blogs/2014/07/30/clustered-shading-android-sample
- [Olsson et. al. 11] Tiled Shading.
  http://www.cse.chalmers.se/~olaolss/main_frame.php?contents=publication&id=tiled_shading
- [Olsson et. al. 12] Clustered Deferred and Forward Shading.
  http://www.cse.chalmers.se/~olaolss/main_frame.php?contents=publication&id=clustered_shading
- [Persson 10] Making it Large, Beautiful, Fast and Consistent: Lessons Learned Developing Just Cause 2. In *GPU Pro*. pp 571-596
- [Persson 12] Graphics Gems for Games - Findings from Avalanche Studios.
  http://www.humus.name/index.php?page=Articles&ID=5
- [Thomas 15] Advancements in Tiled-Based Compute Rendering
- http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/Advancements-In-Tiled-Rendering.ppsx

Questions?

@_Humus_
emil.persson@avalanchestudios.se

AVALANCHE STUDIOS