The rendering technology of Killzone 2

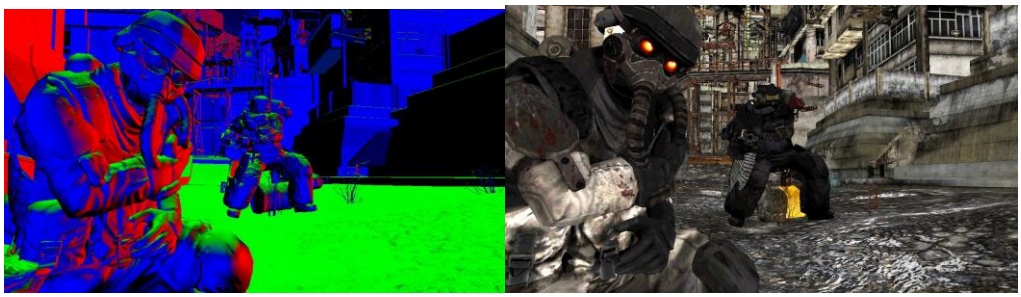
Extended presentation notes

Deferred shading differs from traditional rendering technique by separating lighting from actual rendering of objects. Properties such as position, normal, roughness, albedo of all objects in the scene are rendered into the G-Buffer in geometry pass. Lighting pass then uses screen-space techniques to read the G-Buffer and accumulate light contribution for each screen pixel. In KZ2 there is of course separate (forward rendering) pass for transparent geometry, particles and coronas that comes after lighting pass and also the extensive post-processing pass at the end of the frame, but these are out of the scope of this presentation.

KZ2 rendering engine started as “Light Pre-Pass Renderer” (check ShaderX7) where one fills G-Buffer with minimum amount of properties required for lighting equation (i.e. position and normals), lighting pass then provides accumulated light intensity values. Third pass (we called it shading pass) then re-renders geometry of entire scene, but now it actually computes final pixels using the shader with material properties (all the textures, colors and magic) and it uses accumulated information from the lighting pass as light intensity for the equation. One early iteration of our engine only used diffuse lighting in lighting pass and we picked two specular lights for each object for the shading pass. In theory Light Pre-Pass Renderer offers more freedom with materials and less bandwidth (in lighting pass) as G-buffer does not need to store all material properties so one is not limited by the amount of useable attributes in the shading pass.

Rendering geometry twice was a huge problem for us. We really push a lot of polygons per frame and since we use quite complex shaders, the geometry and shading pass were not that much different – both used a lot of lookup textures for detail maps and layer mixing, fat vertices for texture coordinates and tangent space. The difference was that 1st pass did not read albedo texture while the 3rd pass not only had to read the albedo but also had to read view space normal from g-buffer again (mostly for environmental map reflections). Therefore we actually did quite a lot of duplicate work in both passes. So when we discovered that we can actually get rid of HalfFloat RGBA buffers, fit the entire G-Buffer into 4 plain 32bit RGBA buffers and we can have anti-aliasing (MSAA) with deferred rendering we decided to move to the more traditional deferred renderer engine.

Our G-Buffer contains view space normal (x and y components are stored in FP16 precision, z is computed on the flight), diffuse color (albedo), material roughness, specular intensity, screen space motion vectors (for post processing), sunlight occlusion factor (used for various optimizations, described in-depth later) and accumulated light color (buffer that in the end holds final pixels with accumulated contribution of all lights). Accumulated light color is initialized in the geometry pass with all indirect lighting contributions and all constant terms of material/lighting equation (for example emissive light, ambient light or constant scene reflection). Indirect lighting contribution for static geometry is stored in lightmaps. Dynamic objects use spherical harmonics (SH) – each level has large set of points where we pre-computed indirect lighting and stored it in SH basis. We pick most influential points (and SHs) for each objects and generate unique SH basis of the lighting for that object and frame. We don’t store position in the g-buffer as we reconstruct it from the depth buffer.



G-Buffer – view space normals and albedo

G-Buffer – roughness and specular intensity

G-Buffer – motion vectors and sunlight occlussion

G-Buffer light accumulation (including indirect lighting) and depth buffer

Final composed image including post processing.

Figure shows how how our G-Buffer fits into four 32bit RGBA buffers and single depth buffer.

With g-buffer full of information we can start the lighting pass. For each light we need to find screen space pixels that are affected by this light and perform the lighting equation. The easiest way to do so is to represent the light with a convex volume (i.e. point light is a sphere with radius equal to the range of light influence, spotlight is represented by a cone) and use stencil buffer to mark the pixels inside this volume. First we render back-faces of the light volume and use the greater-equal depth test and mark all pixels that pass the test (stencil test set to write 1 on depth pass, 0 otherwise). These are pixels that could be influenced by the light because they are in front of the (far) light boundary.

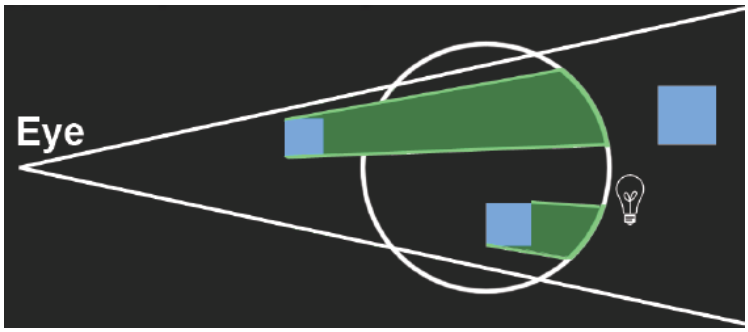


Figure shows the effect of back-face rendering of light volume (side view in 2D). Green areas connect pixels that will be marked in the stencil being potentially lit because they are in front of the far light volume boundary.

If we now render front-faces of the light volume with depth test set to less-equal and only on pixels marked by previous pass (stencil test set to equal 1), the pixels that pass both tests would be exactly pixels inside the light volume. Therefore we commonly use the final light shader for this pass and just render the light contribution into the light accumulation buffer.

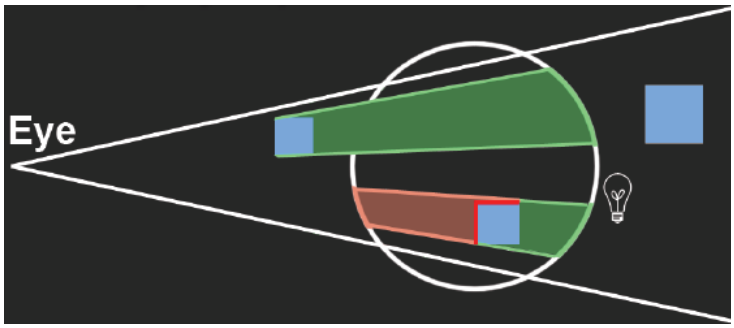


Figure shows the effect of front-face rendering of light volume (side view in 2D). Red areas connect pixels that will be processed by final shader – both marked in 1st stencil pass and passed depth test in 2nd pass.

Note that situation changes a bit if camera is inside the light volume as we only render 1st pass using the final shader (and no stencil marking) and we skip the 2nd pass.
We also have a special "Frankenstein" light that is a mix of a spotlight and a point light – it supports casting of shadows inside a cone just like spotlights and everything outside this cone is rendered as an point light. This special light is used for example for muzzle flashes of the guns and has to be rendered in rather complex way as a combination of spotlight and point light using the stencil buffer magic in several passes.

Shadow casting lights also need to have shadow map prepared before the final light shader pass. To avoid such extra work for lights that get rasterized but turn out to be invisible (i.e. light is inside view-frustum, but is occluded by a wall), we perform the 2nd pass of light render twice, first with NULL shader (no color/depth output) and occlusion query to determine the amount of screen pixels affected by the light. We then instruct RSX to use this query for conditional rendering of shadow map and final light pass. Conditional rendering feature of RSX allows it to skip any rendering commands if pixel count of given query is 0. It's really fast feature as it does not require any read-back from CPU.

Already at this point we can make several further optimizations for light rendering. We use depth bounds test on GPU to help to reject invisible pixels early. Depth bounds test checks if value that is already in the z-buffer is within specific range and if not, GPU rejects the incoming pixel. In case of light pass we use nearest and furthest z value of the light volume as depth bounds parameters and the test essentially rejects all pixels that cannot be lit by the light because they are not in the volume. The big win of this test comes from utilization of early depth cull unit early in the RSX pipeline to avoid any rasterization/ROP costs for such pixels.

Next we compute approximate size of the light on the screen. To make things simple and fast, we use bounding sphere of the light and distance from the camera to compute the angular size of the volume. If the size is bellow pre-defined threshold, we don't use the stencil pass at all and just rely on the depth-bounds test as the time spent in stencil pass and all extra setup makes it more expensive than any savings we can gain (for Killzone 2 we determined this threshold to be 20% of the diagonal FOV of camera).

Each light also contains information about the minimum on-screen size it has to have to actually cast shadows (usable of course only for lights that have shadows enabled in first place). For such light we gradually fade out the shadow intensity depending on the angular size. When the size

gets bellow given threshold, we switch final light shader to a cheaper version that does not use shadow maps (and we also skip the render of the shadow map). We don't place any other limitation on the number of shadow casting lights on the screen.

With MSAA enabled we have few options how to render the light. For example the simple solution is to treat each sample individually (i.e. 2x2 MSAA mode has 4 samples for each final screen pixel; Quincunx MSAA has 2 samples for each screen pixel). This means that we run full light shader for each sample – essentially performing super-sampling in the light pass.

We chose different option. We run light shader in final pixel resolution (i.e. for 2x2 MSAA mode the shader would run only once for each group of 4 samples and the single result is replicated for all samples). This means that we have to account for MSAA inside the shader – we perform lighting equations for all samples during single shader run and we output the averaged value. This of course is not entirely correct for Quincunx pattern, but it produces good results. Our motivation for this approach was simplicity and also possibility of better optimizations.

One straightforward optimization is to determine if all samples are the same (which is true for all non-edge pixels) and perform only single shading operation. For us this approach did not bring consistent improvement so we removed it (sometimes we got better performance, sometimes slightly worse), but we attribute it mainly to the fact that we only use 2 samples per pixel and the savings do not compensate for the extra work shader has to perform for dynamic branching. With higher MSAA modes, this option becomes more and more viable.

Another small optimization can be achieved by using the same screen position coordinates for all samples – for example during reconstruction of view space position (or view vector) from pixel screen position and depth buffer.

The main optimization possibility however comes from shadow map filtering. We use percentage closer filtering (PCF) with multiple (up to 12) samples. We use Poisson-disk distribution of sample coordinates (one pre-generated set for each number of samples) and each individual sample also uses hardware shadow filtering support of RSX (We tried different methods such as variance shadow maps during prototyping, but PCF proved to be the method that worked reliably in all conditions without special case problems). The straightforward implementation would be to perform full PCF for each sample. This would however become quite expensive with the high amount of shadow map samples (i.e. 12 samples for PCF with 2x2 MSAA mode would mean 48 samples per pixel).

Therefore we decided to distribute the single set of Poisson disk coordinates among multiple samples. Let's say we have Poisson disk with 12 coordinates and we have 4 samples per pixel (2x2 MSAA mode) - then each sample gets exactly 4 unique coordinates for shadow map filtering during the lighting equation computation. For non-edge pixels where all samples are exactly the same we basically get multiple lighting equations that only differ by used shadow map filtering coordinates (but use the same normal, albedo, specular components…). Averaged value of these results is then the same as performing single lighting equation with the full PCF kernel. Therefore for non-edge pixels we automatically get full quality PCF. For edge pixels where the samples are different, we of course get lower quality PCF (less samples per unique sample) but since the final color of each sample is usually different from the others in the group and the pixel shader result is an averaged value of all sample colors, the visible error is minimal.

Sunlight is our global infinite light (we also use it for flash effects of lightning strikes in some levels) and is rendered as a full-screen effect instead of a convex hull representation. This makes it our single most expensive light type and we developed several shortcuts and optimizations to make it render fast.

First simplification takes advantage of the fact that sunlight in the level does not move. Therefore we can pre-render sunlight shadows of the static level geometry and store it in the alpha channel of the lightmaps for each static object. Later we write it into G-Buffer as the sunlight occlusion term during geometry pass. We use this value before rendering the actual sunlight to mark all pixels that do not have to be processed by the expensive sunlight shader because they are in full shadow. We render full screen quad with a simple shader that just reads sunlight occlusion part of the G-buffer and cancels pixel rendering (texkill in DX or clip in CG) when the value is 0. We disable actual color writes during this pass so the pixels that do not get canceled in the shader would only change the stencil value to 1. The final sunlight shader runs only on these marked pixels. Some objects write 0 to the sunlight occlusion channel to avoid unnecessary expensive sunlight processing – for example for skybox and distant background objects we can include the sunlight contribution directly in lightmaps (during offline rendering) and we do not have to spend GPU time with expensive shader on such pixels.

The sunlight occlusion value of G-Buffer is also used during final sunlight rendering - we mix the pre-rendered shadows with real-time shadows:
final_sun_shadow = min(real_time_sun_shadow, pre_rendered_sun_shadow). This way we can achieve constant look and filtering of large shadow areas (for example large shadows of buildings) even in the distance (so we can drop real-time shadows without visual glitch).

We use cascade shadow maps for real-time sunlight shadows – we split camera frustum into several sections (cascades) along the view distance and render shadow map for each of these sections (improvements of actual shadow map rendering are described in more detail later).

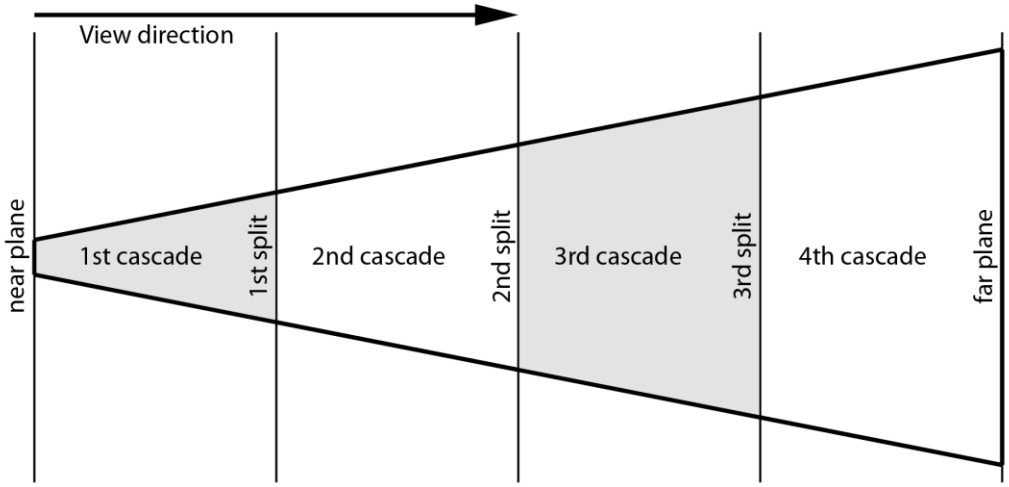


Figure – 2D visualization of camera frustum split (uniformly in this case) into separate cascades.

Deferred rendering makes it very easy to render final sunlight contribution in multiple passes, one pass for each cascade. We render fullscreen quad for each pass and use the already mentioned depth bounds test (with the lowest and highest distance of a cascade as parameters) to make sure the shader only touches screen pixels that are affected by given cascade. This also makes it possible to use different optimization strategies and different shadow map sizes for different parts of the scene. Closest pixels (the closest cascade) will be lit by full quality sunlight shader (for example high number of PCF samples for shadow filtering). Further cascades can be rendered with lower amount of PCF samples (the penumbra regions on screen are not that large so less samples is sufficient). The furthest cascade (distant background pixels) can be rendered without real-time shadows. MSAA quality can be also selected per cascade. Full quality MSAA means the same rendering and shadowing as mentioned early in this text. Lower quality MSAA mode has higher performance because it will compute only single light equation for the closest sample

(using averaged albedo value of all samples in a group). Shadow filtering (including sunlight occlusion from G-buffer) is still computed correctly for each sample. Again, for non-edge pixels this provides proper results. The motivation for edge pixels is that in distant cascades (where this mode is used) we have high probability that most of samples would belong to the background where sunlight occlusion is 0 and that difference in lighting at these distances is not that prominent. Also the potential error is less visible because of post processing (lower contrast, possible distance blur).

Rendering of shadow maps has few common optimization concepts for all light types we use. We cull objects that do not contribute with shadow to the visible scene – this means that we test each object not only if it appears in the shadow map, but also if it casts shadows into the camera frustum. We also determine approximate object size in the shadow map and on the screen (using the same angular size algorithm mentioned earlier for lights) and we skip objects that are either very small on screen or very small in shadow map. Both thresholds are defined per object (so for example important object can cast shadows at any given size and non-important background objects will stop the shadow casting quite early). Each light can additionally bias these thresholds to either include more objects in shadow map (i.e. muzzle flash of first person weapon) or lower the amount of objects in shadow map (some large scale spotlight that is in the background and only need to have few very big objects casting shadows).

Shadow map rendering for each sunlight cascade additionally uses quality improvement we published in the book ShaderX6 ("*Stable rendering of cascaded shadow maps*"). Since area covered by shadow map for single cascade directly depends on camera position and orientation (each cascade is just a region of camera frustum starting at given distance and ending at the distance where next cascade starts) it changes every for frame. Continuous movement and rotation of the camera (i.e. player movement in the level) cause sample changes of shadow map rendering area and will produce visible shimmering of shadow edges on screen thanks to polygon rasterization rules of GPU. We avoid the shimmering by removing the sample changes during camera move and also by avoiding rotation of shadow map area during camera rotation. Next figure shows main principle of the algorithm:

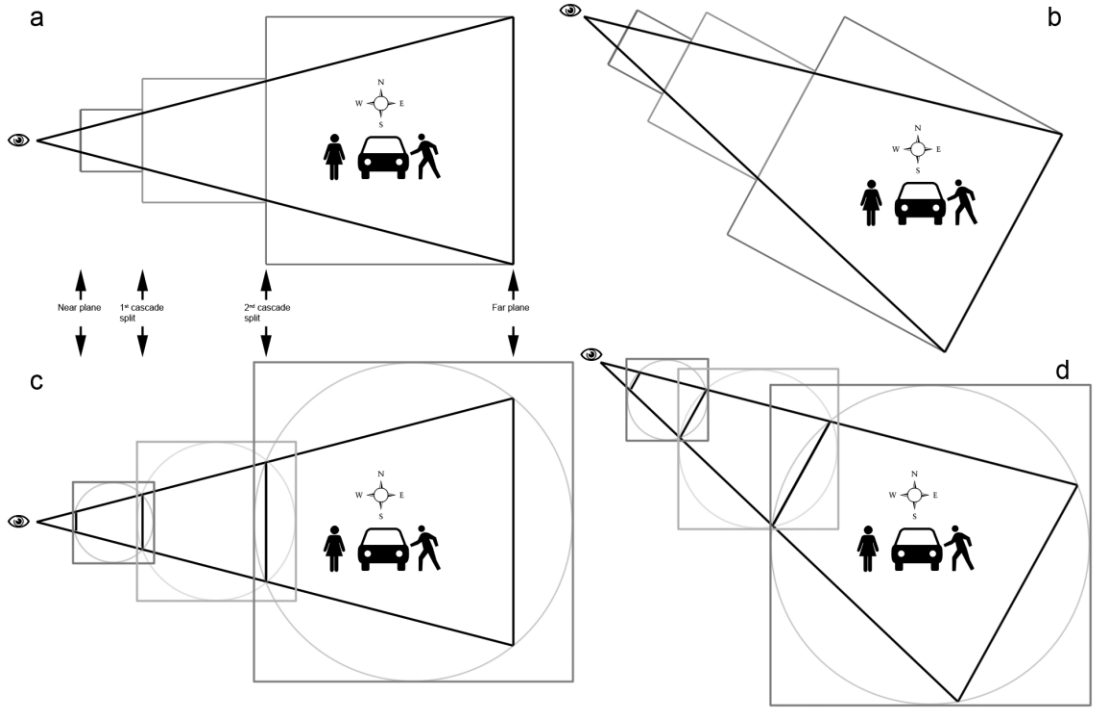


Figure shows view frustum in a world space split into three cascade frustums and corresponding shadow map coverage for each cascade. For simplicity of 2D representation, we use a top view with light direction pointing straight down to the horizontal world plane. Part a) shows the traditional shadow map generation method where the u (or v) direction in shadow map space is parallel to the view direction to maximize the shadow map coverage of the cascade. Part b) shows how the shadow map is rotated (relative to the world space) when the camera rotates. This will cause objects in the shadow map to be rasterized differently every frame. Part c) shows our approach, where the shadow maps are generated using Minimum Enclosing Sphere of the corresponding cascade frustum. Part d) shows how our approach allows shadow maps to not rotate (relative to the world) when the camera changes the direction.

SPU memory allocation.

One common way to generate display list is to use double buffering (generate display list for frame N+1 while GPU is consuming display list for frame N). This method potentially requires a lot of memory (all dynamic resources such as vertex arrays of skinned objects have to be double buffered too). The other solution is to use ring-buffering – CPU is generating display list only slightly ahead of the GPU and when CPU reaches end of the buffer, it jumps to the beginning of the ring buffer and continues generation there. This method requires more synchronization work to prevent CPU from overwriting data that GPU did not consume yet (or again double dynamic buffered resources).

None of these approaches solves problems of out of order parallel display list generation – we want to generate most of the display list just-in-time for GPU and out of order on multiple SPUs (good example is light pass display list where we need to spawn multiple SPU tasks to generate display list for shadow map rendering – we cannot stall any SPU and wait for other tasks to finish). Our solution for this problem was to treat memory allocated for display list and rendering resources more like regular dynamic memory allocations with GPU signaling what parts of memory are already free and can be reused.

So we created block based render memory allocator that manages fixed amount of blocks of memory (all of which are the same size in our case). We use Fence mechanism of GPU to determine whether given memory block is free or not. Suppose the SPU fills block of memory “A” with display list commands. When it reaches the end of the block, it asks render memory allocator for a new block “B”. The first command written into this new block is a Fence command that signals that when GPU reaches this point, block “A” is free and can be reused by some other SPU.

The allocation method alone on SPU is just a walk over all registered blocks and query whether GPU already passed a fence of this block or not. If it passed the fence, block is free, SPU (atomically) resets the fence value and can use the memory.