

The background of the entire image is a vibrant yellow with a complex, abstract pattern of overlapping circles, lines, and organic shapes in shades of green, blue, and brown, creating a sense of depth and movement.

GDC

09 learn
network
inspire

www.GDConf.com

Game Developers Conference®
March 23-27, 2009 | Moscone Center, San Francisco



The Rendering Technology of Killzone 2

Michal Valient
Guerrilla BV





How we made Killzone 2 run 30fps

- » Deferred shading
- » A diet for render targets
- » Dirty lighting tricks
- » Rendering, memory and SPUs
- » Q&A

Killzone 2 has been in the development for just under 4 years.

During that time we tried a lot of rendering techniques.

Some are very Killzone 2 specific but some are general.

This talk is about those rendering optimizations and solutions in KZ2 that we feel could be helpful for your next game.

KZ2 goal - realistic and distinctive lighting.

We wanted technique that allows us to focus on lighting with many dynamic light sources - Deferred shading.

I'll do a quick intro for those who are not familiar with this technique.

How to hit 30FPS?

We used to have very fat render targets and that costed us a lot of bandwidth.

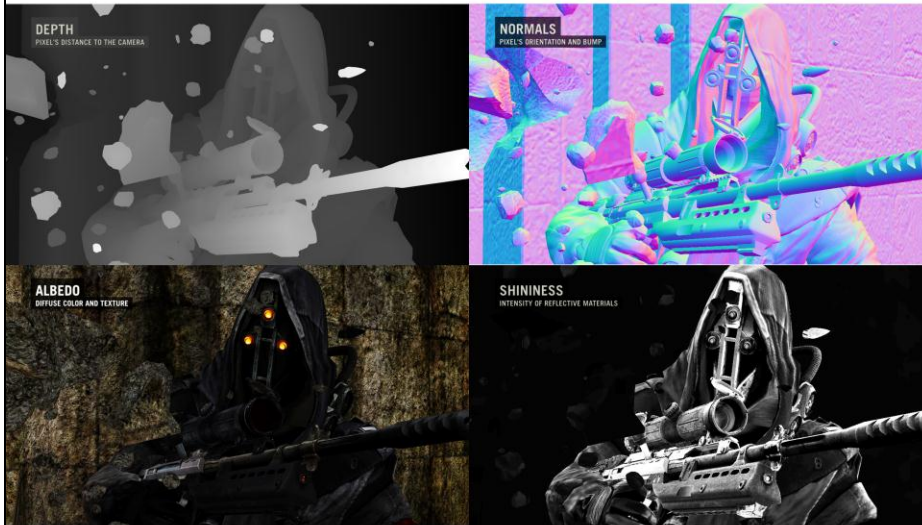
A diet for render targets - how we reduced memory footprint and what had to go.

Dirty lighting tricks - lights are our heaviest spot. So we focused mostly on optimization of this part.

Rendering, memory and SPUs - We generate our push buffer on SPUs

We needed allocator for all the rendering memory. It allows out of order allocations and releases memory as RSX consumes the frame.

What is deferred shading?



1. Geometry pass – fill the G-Buffer

Forward rendering:

Take a triangle

Compute material properties (load all textures, constants) into the shader

Load all light properties into the shader

In the same shader compute final lighting of the pixel.

Repeat for as many lights as needed.

Or... group lights into batches in order to accelerate it - shader combination hell. How do you manage shadow maps.

Deferred shading is different.

Scene geometry rendering is separate from lighting.

Lighting is very similar to image space post-process.

This allows us to use large amounts of lights with much simpler engine.

Instead of directly shading triangles, we store relevant material information for each visible pixel in a structure called G-buffer.

G-Buffer - set of render targets.

We store properties such as position, normal, roughness, albedo...

...pretty much everything we need to know to generate the final appearance of each pixel later on.

This happens in a single pass called geometry pass.

What is deferred shading?



2. Lighting pass – accumulate light info

Lighting pass comes right after geometry pass. On the picture you can see the scene and we're going to add more lights there.

What is deferred shading?



2. Lighting pass – accumulate light info

For each light find all screen pixels that are affected by this light.

For these pixels we read the G-Buffer, compute the lighting equation and add the light contribution to the result buffer.

If the light casts shadows, we generate the shadow map first.

We do this in a loop for every light on screen.

The good:

Shaders which fill G-Buffer are simpler - don't include lighting code.

Light shaders are also simple - don't include material computations.

If multiple lights affect the same region, this does not add any complexity to the code.

This all makes management of the engine much simpler.

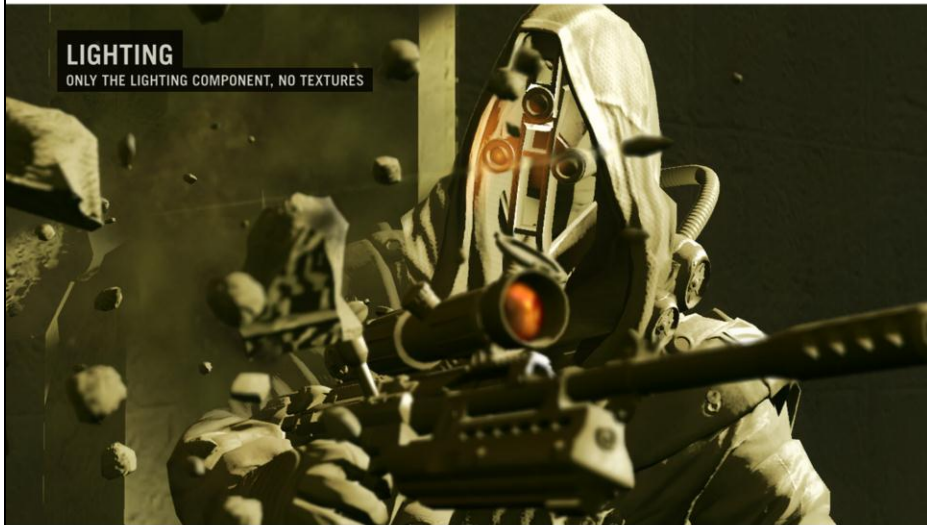
You can have large numbers of lights on screen and the performance only depends on number of pixels they cover and not on the number of polygons the scene has.

Drawbacks

These are mainly related to the fact that you cannot easily make material that would change its appearance based on the lighting that affects the pixel.

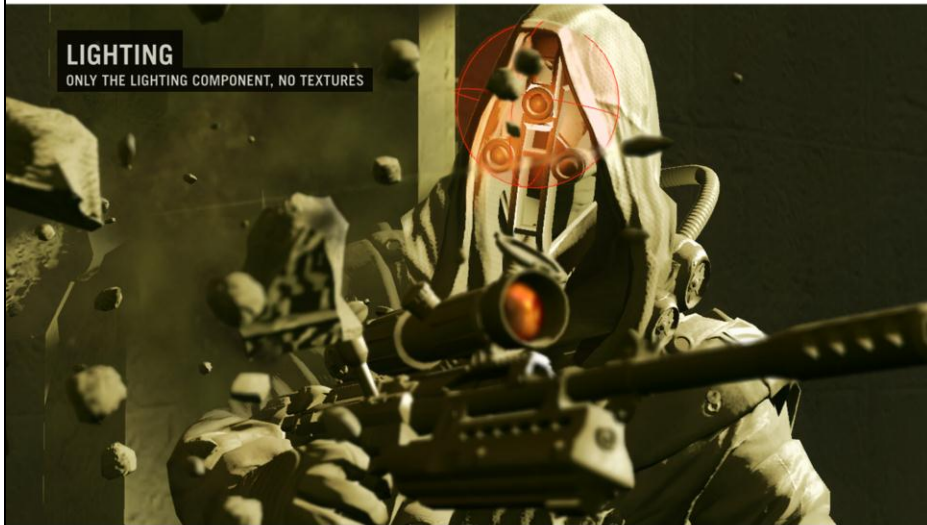
But we're working on it.

What is deferred shading?



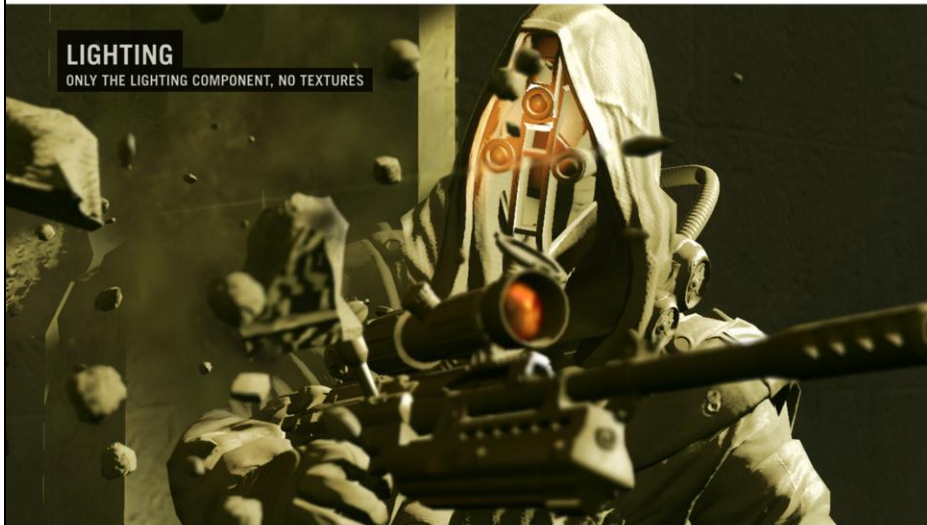
2. Lighting pass – accumulate light info

What is deferred shading?



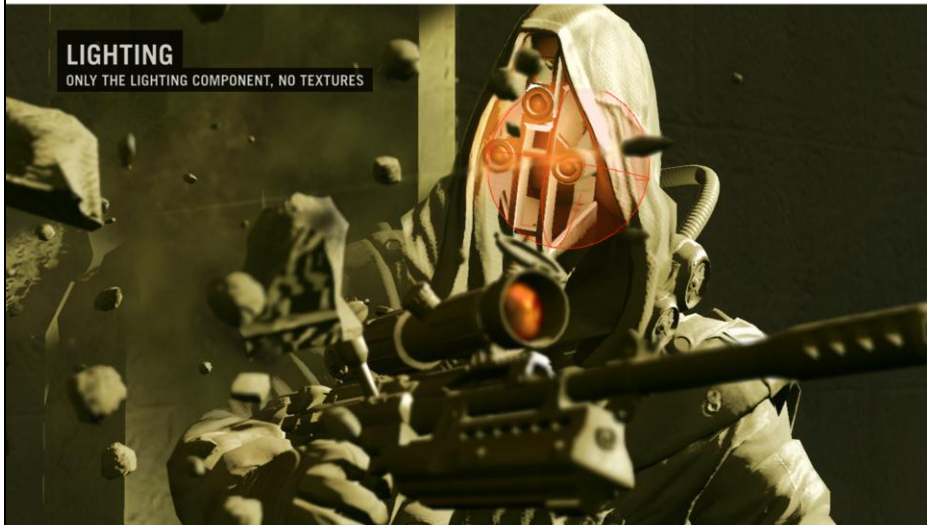
2. Lighting pass – accumulate light info

What is deferred shading?



2. Lighting pass – accumulate light info

What is deferred shading?



2. Lighting pass – accumulate light info

What is deferred shading?



2. Lighting pass – accumulate light info

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.



G-Buffer

DS	Depth (24bit integer)		Stencil
RT0	Lighting accumulation RGB		Glow
RT1	View space normals XY (RG FP16)		
RT2	Motion vectors XY	Roughness, spec. intensity	
RT3	Albedo RGB		Sun shadow

- » RGBA FP16 buffers proved to be too much
- » Moved to RGBA8
 - ↳ 4xRGBA8 + D24S8 - 18.4mb
 - ↳ 2xMSAA (Quincunx) - 36.8mb
- » Memory reused by later rendering stages
 - ↳ Low resolution pass, post processing, HUD

Our G-Buffer (and the back buffer) was at at one point FP16. But FP16 buffers are expensive because of the higher bandwidth requirements so we decided to go for RGBA8.

We already knew that proper HDR is not very important for us - our artists like to be in control of features and automatic tone mapping is not something that you can control too much. So we gave them extensive color correction filter to desaturate the game.

We were able to compress our G-Buffer data to 4xRGBA8 buffers + Z-buffer. This was just above 18mb and therefore we decided that we can actually go with Anti-aliasing which doubled the G-Buffer size to just bellow 37mb.

To not further increase the memory consumption, we reuse G-Buffer memory for any render-target that we need after the lighting pass - low resolution buffers, post processing buffers, HUD.



G-Buffer

DS	Depth (24bit integer)		Stencil
RT0	Lighting accumulation RGB		Glow
RT1	View space normals XY (RG FP16)		
RT2	Motion vectors XY	Roughness, spec. intensity	
RT3	Albedo RGB		Sun shadow

- » View space position computed from depth buffer
- » $\text{Normal.z} = \sqrt{1 - \text{Normal.x}^2 - \text{Normal.y}^2}$
 - ⚠ No negative z, but does not cause problems
 - ⚠ 2xFP16 compressed to RGBA8 on write
 - ⚠ Cg: `unpack_4ubyte(pack_2half(Normal.xy))`
- » Motion vectors – screen space

We don't store position in the g-buffer now (used to have one in FP16 era) - we reconstruct it from the depth buffer. We wanted to use position buffer modifications for some displacement effects, but this never materialized so we just removed it.

Our G-Buffer contains view space normal (x and y components are stored in FP16 precision, z is computed on flight), since we keep the render targets in RGBA8 format, we needed to store 2xFP16 - cg has instructions to do it without quality loss.

We have 2 channels reserved for screen space motion vectors (for post processing).



G-Buffer

DS	Depth (24bit integer)		Stencil
RT0	Lighting accumulation RGB		Glow
RT1	View space normals XY (RG FP16)		
RT2	Motion vectors XY	Roughness, spec. intensity	
RT3	Albedo RGB		Sun shadow

- » Albedo - material diffuse color
- » Roughness - Specular exponent in log range
- » Specular intensity - single channel only
- » Sun shadow - pre-rendered sun shadows
 - ⊗ Mixed with real-time sun shadows


Of course we store the typical material properties too.

Albedo is any combination of diffuse properties (color, textures) - pretty much anything our artists design in Maya (this is actually true for any of g-buffer parameters)

Material roughness - this is how we call the specular exponent. We store it in log range to keep higher precision for low exponents (high roughness).

Specular intensity - just single channel as we discovered that having specular material color is not something our artists were using.

Sunlight occlusion factor - pre-rendered sunlight shadows. We use it to optimize the performance of the real-time sunlight. We mix it with real-time shadows in shader too to achieve nicer look.



G-Buffer

DS	Depth (24bit integer)	Stencil
RT0	Lighting accumulation RGB	Glow
RT1	View space normals XY (RG FP16)	
RT2	Motion vectors XY	Roughness, spec. intensity
RT3	Albedo RGB	Sun shadow

- » Lighting accumulation buffer (LAB)
 - ✎ Geometry pass fills-in indirect lighting terms
 - ✎ Stored in lightmaps and IBLs
 - ✎ Also adds ambient color, scene reflections...
 - ✎ Lighting pass adds contribution of each light
- » Glow – contains HDR luminance of LAB
 - ✎ Used to reconstruct HDR RGB for bloom

www.GDCConf.com

Accumulated light color is a buffer where we add all lights and build the final image. It's pretty much our back buffer.

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). Everything is multiplied with appropriate texture to produce final term (i.e. diffuse lighting with albedo)

Indirect lighting contribution for static geometry is stored in lightmaps (YUV space for better quality in S3TC5).

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 view dependent texture (IBL - image based lighting) of the lighting for that object and frame.

Reflection texture - cube map specified for each level zone. Cheap speculars.

Glow buffer is updated in sync with LAB and contains scaled luminance of LAB. This way we can have HDR input colors for the post processing filters such as Bloom, but we use it also in motion blur or depth of field.

Light accumulation buffer



» After geometry pass - no real-time lights

Image from the real-time version of the KZ2 'Bullet' commercial. Contains only indirect lighting.

Light accumulation buffer



» After lighting pass - with real-time lights

Note - left side of the screen is without the textures, right side of the screen is with textures. Spot the difference...



Lighting pass

- » The most expensive pass
 - ⌚ 100+ dynamic lights per frame
 - ⌚ 10+ shadow casting lights per frame
 - ⌚ Anti-aliasing means more of everything
- » Optimization strategies
 - ⌚ Avoid hard work where possible
 - ⌚ Work less for MSAA
 - ⌚ Precompute sun shadow offline
 - ⌚ Approximate

In some cases as much as 300 lights on screen.

Optimizations on the next few slides.

We're trying to be lazy here

Avoid hard work (running the expensive shaders) where possible. Ideally we try to avoid any shading at all.

Work less for MSAA. MSAA means more samples to be lit, but that does not mean we cannot be smart and we can actually get the cost closer to the non-MSAA case.

Precompute Sun shadow offline - the Sun shadow buffer mentioned earlier and how we use it to accelerate sunlight rendering.

Approximate - if everything else fails and it's still slow, we can try to hack our way towards 30fps.



Avoid hard work where possible

- » Don't run shaders
 - ⌚ Use your early z/stencil cull unit
 - ⌚ Depth bounds test is the new cool
 - ⌚ Enable conditional rendering
- » Optimized light shaders
 - ⌚ For each combination of light features
- » Fade-out shadows for small lights
- » Remove small objects from shadow map

Don't run shaders.

Since lighting pass is done in screen-space you have to be careful what to light and what not.

When the light is behind the pillar, it is still in the screen space, but not visible. So you want to skip shading here.

Similarly, when the light just affects the air in the hallway - it is on screen, visible, but it actually affects zero pixels because all are in the greater distance.

The easiest way to avoid the work is to use early cull units of RSX (and we got burned many times on this).

Therefore it's best to use some actual 3D representation of the light in the lighting pass (such as sphere for the point light) and use existing scene depth buffer to help with visibility tests and utilize the cull units.

First render back-faces of the light representation with NULL shader, depth test set to 'greater'.

This allows you to stencil mark pixels that could be affected by the light because they are either inside the volume or even closer to the viewer.

Then front-faces of the light representation with final shader with 'less-equal' depth tests + stencil test enabled to pass only pixels marked in prev. step. Resulting affected pixels are exactly the ones inside the light volume.

Depth bounds test - very cool feature. Test will actually reject pixels if **existing** depth buffer value is not within given min-max range. This means it can be done very early in the pipeline and works great with early depth cull.

Compute min and max z that the light can affect and setup the depth bounds test before rendering the light representation.

Conditional rendering - for shadow casting lights. Run pixel query during the stencil pass to determine if the light affects at least some pixels. Conditional rendering will instruct RSX to skip any render commands if query is zero - and is enabled for both shadow map rendering and final expensive shader.

Stencil/queries are not necessary for the small lights (less than 20% of the screen, no shadow casting) - depth bounds test is enough there.

Optimized light shaders

Since light shaders are simple in the deferred shading, it's easy to have several versions of them for each light - different shadow filtering quality, no-specular, no-projector texture. The amount of combinations is still quite small so it's easy to run optimal shaders for each light type.

Fade out shadows...

Since we don't have any hard limit on number of shadow casting lights we try to fade-out the real-time shadows for distant (small) lights. After the shadow is faded-out, we can switch to the non-shadow casting shader.

We have default values set-up for this, but artists are free to use their own overrides.

Remove small objects...

Further optimization to lower vertex processing costs is to remove small objects (small both in shadow map and on-screen) from the shadow map. These objects usually don't contribute to the shadow too much.

Again, we have some defaults, but artists can specify the thresholds both per light and per object.



Lighting pass and MSAA

» MSAA facts

- ⌚ Each sample has to be lit
- ⌚ Samples of non-edge pixel are equal

» KZ2 solution – in-shader supersampling

- ⌚ Run at 1280x720 not 2560x720
- ⌚ Light two samples in one go

» Where is the gain?

- ⌚ Shadow filtering can be much cheaper

MSAA

Pixel consists of individual samples (and these samples are also stored in the g-buffer)

Each sample has to be lit.

Samples are the same for non-edge pixels and we try to use this.

Light two samples in one go

run shader once per pixel,
read g-buffer for both samples,
perform two lighting equations
output the average value.

Gain compared to regular supersampling (lighting samples individually)

Lighting of course has similar cost, but we can actually save on shadow map filtering.

Next.



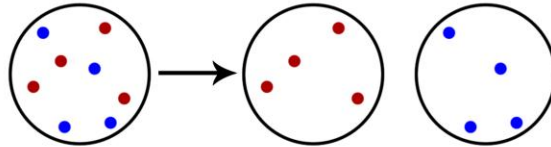
Shadow map filtering distribution

» Motivation

- Define filtering quality per pixel rather than per sample.

» Split filter coordinates into disjoint sets

- One set per pixel sample



» MSAA is almost as fast as non-MSAA

Motivation

We use percentage closer filtering for shadows.

Artists don't specify filtering quality per sample (they don't care about how the light will be rendered on screen - they just want kernel of certain size). Therefore we can just do it per pixel.

Example - light has 8 shadow filtering taps.

We split filter kernel to 2 disjoint sets (4 taps each)

Use one set for shadow filtering for one sample, the other set for the second sample.

For non-edge case we have equal g-buffer samples, but different shadow filter taps.

So we get the same result as performing full filtering (8 taps in example) for the non-msaa case and for the same speed.

Edge case - lower quality, but not noticeable.



Sunlight

- » Fullscreen directional light
- » We divide screen into depth slices
- » Each depth slice is lit separately
 - ⌚ Different shadow properties
 - ⌚ Use depth bounds test
- » Use 'Sun shadow' from G-Buffer
 - ⌚ Stencil-mark pixels completely in shadow
 - ⌚ Skip expensive sunlight shader!
 - ⌚ Also mixed with real-time shadows

Fullscreen + MSAA = Panic. It can affect every pixel/sample of the screen.

To get at least some optimization - divide screen into depth slices.

Each slice has different shadow properties - number of shadow filter taps, shadow map size.

Allows performance vs. quality scaling.

Sunlight is rendered as fullscreen quad and we use depth bounds test to limit it only to given depth slice.

We finally get to the Sun shadow buffer. We use it to stencil-mark pixels that are completely in shadow and therefore can be skipped by expensive real-time sun shader. This is huge saving for us - most of the time close to 50%.

We also use the 'sun shadow' buffer in the final shader and we mix it with the real-time shadows. This way we can drop real-time shadows for last depth slice and it also hides some artifacts that can happen on depth-slice boundary where we have different shadow settings.

Sunlight rendering - example



» Final scene - Corinth River level

Scene from Corinth river level.

Sunlight rendering - example



» Sun Shadow channel

Sun shadow channel - notice the low resolution.

Sunlight rendering - example



» Red – skipped pixels

Skipped pixels

Notice characters - IBLs are not part of the sun shadow - we write white.

Exception - dark areas can have IBLs that have sun shadow in the alpha color.

Sunlight rendering - example



» Depth slice 3 - background, no shadow

Sunlight rendering - example



» Depth slice 2

Sunlight rendering - example



» Depth slice 1

Sunlight rendering - example



» Depth slice 0

Sunlight rendering - example



» Only real-time shadows

Real time shadows only. You can see shadowing errors (halo around the block caused by depth bias of shadow map) and also some rather un-pleasant sharp bump mapping caused by light being almost parallel to the block surface. All this was hidden by the 'sun shadow' from the g-buffer.

Sunlight rendering - example



» Sun shadow channel - free ambient occlusion!

Mixed again with 'Sun Shadow'.

Notice the free ambient occlusion effect (on the ground) we get with this.



Sunlight rendering – Fake MSAA

- » Used only in the distance
- » We cut down on lighting cost
 - ⌚ By doing shadows properly, but...
 - ⌚ running lighting equation on closest sample only!
- » But isn't it wrong?
 - ⌚ It's a hack. We hope you don't notice it.
 - ⌚ Works correctly against background
 - ⌚ The edges are still partially anti-aliased
 - ⌚ Distant scenery is heavily post-processed

We load albedo, 'sun shadow' and position for each sample.

We do shadowing as usual (separate shadow filter for each sample, mix with 'sun-shadow' too).

Then we do only single lighting equation to get diffuse and specular lighting - for closest sample, and we use this result also for the farther sample.

We multiply this equation with per-sample albedo and shadow (just like if we had result for each sample).

Works for non-edge case (g-buffer samples are the same).

Works on for foreground-to-background edge because background has 0 in 'sun shadow' channel. Since we pick the nearest sample, we always use the foreground object to do lighting equation and 0 in the 'sun shadow' channel ensures that this result is not applied to the background.

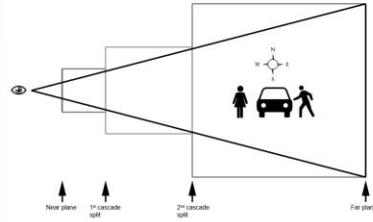
Edges are still partially anti-aliased - from the geometry pass + usually different albedo and/or shadow helps to hide the sharp edge.

If that fails, we assume that the distant scene is heavily post-processed and hidden by the depth of field or particles or color correction.



Sunlight – shadow map rendering

- » We generate shadow map for each depth slice
- » Common approach
 - ⌚ Align shadow maps to view direction
 - ⌚ Pros - Maximize shadow map usage



Think - cascaded shadow maps.

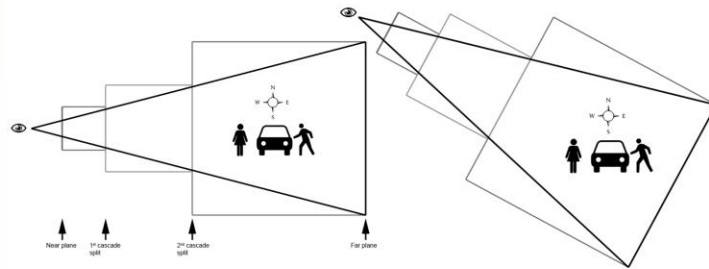
Image - shadow maps aligned to the view (top view, light comes from the top)



Sunlight – shadow map rendering

» Shadow map changes each frame...

- ⌚ ...when viewer moves or rotates
- ⌚ Smooth movement causes sub-pixel changes in shadow map
- ⌚ Result – shadow shimmering



Shadow map changes per frame

As viewer moves or rotates.

Smooth movement - sub-pixel changes in shadow map

Manifests as shimmering during rendering

Sunlight – shadow map rendering

» [Shadow shimmering example video](#)



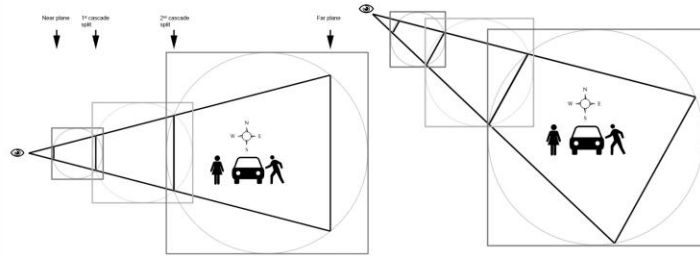
Shadow shimmering example.

Note that this is modified code to show the problem - there is no additional shadow filtering performed.



Sunlight – shadow map rendering

- » Our fix - remove shadow map rotation
 - ⚙ Align shadow maps to **world** instead of **view**
 - ⚙ Remove sub-pixel movement
 - ⚙ Cons – unused shadow map space



rotation independent -

Find the bounding sphere of the depth slice region

Any view rotation will then fit into the sphere.

remove sub-pixel movement -

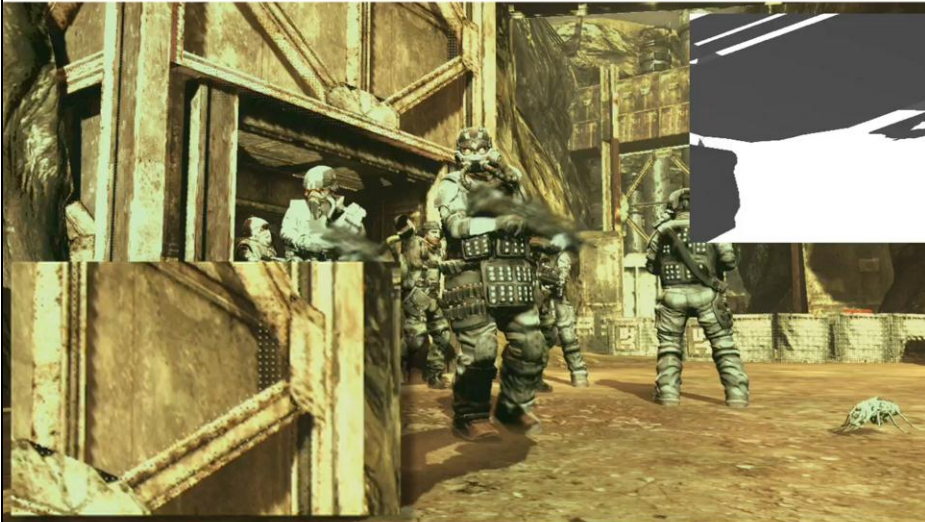
Round the smooth shadow map movement to the next shadow map pixel.

This will assure that even if we generate the shadow map every frame, it will be the same unless viewer moves by more than one shadow map pixel.

Check ShaderX6 for details.

Sunlight – shadow map rendering

» [Stable shadows example video](#)



stable shadow example.

No shimmering of shadows

See how shadow map “jumps” when camera moves (jumps by 64pixels to show the effect, in reality it jumps by 1 pixel exactly).

Note that this is modified code to show the stable solution - there is no additional shadow filtering performed and real in-game scene looks much better.



» GPU driven memory allocation system



Push buffer (PB) building

- » Multiple SPU's building PB in parallel
- » Additional SPU's generating data
 - ⌘ Skinning, particles – vertex buffers
 - ⌘ IBL interpolation - textures
- » Common solutions for memory allocation
 - ⌘ Ring buffering
 - ⌘ Issues with out-of-order allocations
 - ⌘ Double buffering
 - ⌘ Too much memory

The problem.

We have a lot of SPU's either building push buffer or building data for the push buffer. This all happens at the same time, even out of order (data generated in different order than the order in which RSX consumes them).

None of the common solutions for memory management worked for us.



KZ2 render memory allocator

- » Fixed memory pool
 - ↳ 22mb block - split into 256k blocks
- » Each block has associated AllocationID
 - ↳ Specified by client during allocation
 - ↳ Only whole block can be allocated
 - ↳ Fine grained allocation stack on top of this
- » Global FreeID identifies free blocks
 - ↳ Updated as RSX consumes 'Free' marker
 - ↳ If (AllocationID ≤ FreeID) -> Free block

Our solution

Take fixed pool of memory and split it into 256k chunks. Each of the chunks represents single allocated block (but we have fine grained allocator working on top of these 256k chunks).

We have a table with single slot for each 256k block. This slot contains allocationID - a number specified by the client (i.e. skinning code, rendering code) during the allocation.

Allocator also contains the FreeID - single number that identifies which 256k blocks are free. When client calls “free(allocationID)” it does not free the memory immediately. It just inserts the marker to the push buffer and when RSX passes that point, it copies the value to the FreeID.

Next call to “allocate” will walk through the table and compare all stored allocationIDs with the FreeID. If (table[idx].AllocationID ≤ FreeID) => block is free and can be re-used.



KZ2 render memory allocator

- » We've got strengths of ring-buffering
 - ⌚ With none of the weaknesses
- » Lockless, out-of-order, memory allocation
 - ⌚ From PPU and/or SPU
 - ⌚ Simple table walk (fast!)
- » Allows immediate memory reuse
 - ⌚ We generate push-buffer just-in-time for RSX
 - ⌚ Block can be reused right after RSX consumption
- » Can allocate memory for skinning early...
 - ⌚ ... and still free at correct point in frame

out of order - the order in which blocks appear in memory is irrelevant, blocks that would be consumed first (i.e. geometry pass) can be allocated later than lighting pass blocks. We'll also have lighting pass spawning multiple shadow mapping tasks, each allocating first free block it finds.

How does this work?

The only requirement is to increase the AllocationIDs in the order in which RSX will consume the buffer.

Therefore we generate a fixed budget of AllocationIDs for a frame and each pass uses IDs only from its range (let's say for this frame, geometry pass uses IDs from 1000 to 1100, lighting pass will use 1200 to 2000 [and will distribute the range among shadow map tasks] and forward pass will use 2100 to 2200. Next frame would be for example 2300 to 2400 for geometry pass and so on...).

... of course there is some extra management for AllocationID overflows.

Immediate memory reuse...

We'll show on next few slides how we build the push buffer for RSX. The main point is that we try to be only few blocks ahead of RSX. In ideal case if RSX consumes push buffer from block N, SPU's are generating block N+2. This allows us to reuse blocks within the same frame as soon as RSX consumes these.

Allocate memory for skinning early...

We know how the frame will look - what would be the AllocationID at the end of each logical block (geometry pass, lighting pass). Therefore depending on the objects features (transparent yes/no, shadow casting yes/no) we can determine the latest point where the skinned vertex arrays are still needed (geometry pass, lighting pass, forward pass, end of frame).



Push buffer generation - example

Memory blocks

AllocID: N



Example:

Allocate block with ID N (let's say for the geometry pass).

Again - it does not matter what is the block position in memory.

What matters is that we gave it allocationID N

THINK

09

learn
network
inspire

www.GDCConf.com

Push buffer generation - example

Memory blocks

AllocID: N

⌛

WAIT

Write the RSX stall to prevent RSX entering the block until it is full and we start to add the render commands.

We usually don't hit the stall because SPU's are quick enough to fully fill the block and unlock it before RSX hits this point (will be shown later).

THINK

09

learn
network
inspire

www.GDCConf.com

Push buffer generation - example

Memory blocks

AllocID: N

WAIT

AllocID: N+1

When the block is full, allocate new one with ID N+1 to continue with the push buffer generation.

Block N+1 - again - does not matter what is the memory position relative to block N.



Push buffer generation - example

Memory blocks

AllocID: N



AllocID: N+1



First instruction of the new buffer is be “Free block N” to free the previous block. This means when RSX hits this point, block N will become available for any allocations.

THINK

09

learn
network
inspire

www.GDCConf.com

Push buffer generation - example

Memory blocks

AllocID: N

WAIT

AllocID: N+1

FREE
N

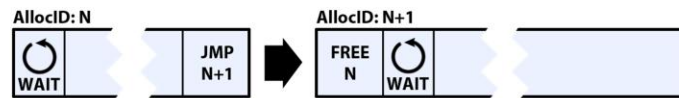
WAIT

Second instruction is RSX stall just like in the previous case.
This way RSX can free the old block as soon as possible and will wait until the new block is filled.



Push buffer generation - example

Memory blocks



Next we connect old and new block with the jump - we instruct RSX to continue with block N+1.



Push buffer generation - example

Memory blocks



... and we remove the RSX stall so if the RSX was fast enough and was actually stalling on that point, it can consume commands from block N and then just continue on block N+1 (possibly stall in N+1 until the wait command is removed).



Push buffer generation - example

Memory blocks

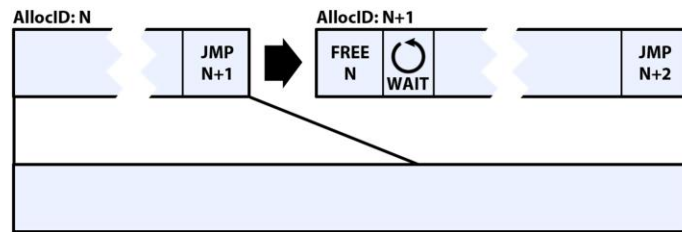


Then we start to fill the new block and if it's full, we proceed as before.



Push buffer generation - example

Memory blocks

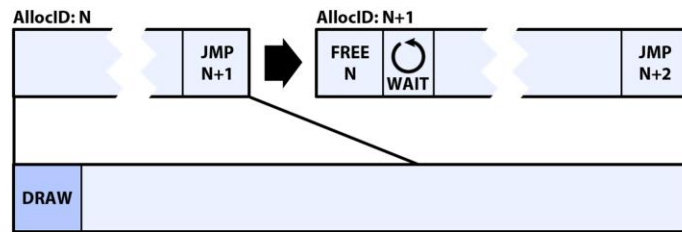


But it's not that simple...



Push buffer generation - example

Memory blocks

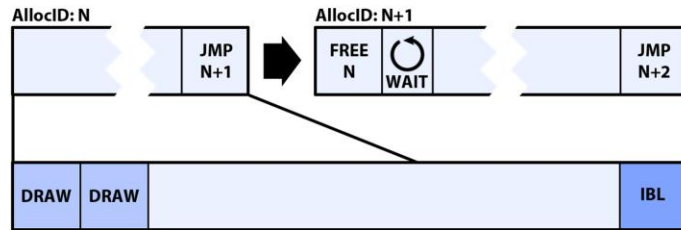


While we generate the draw command to the push buffer...



Push buffer generation - example

Memory blocks

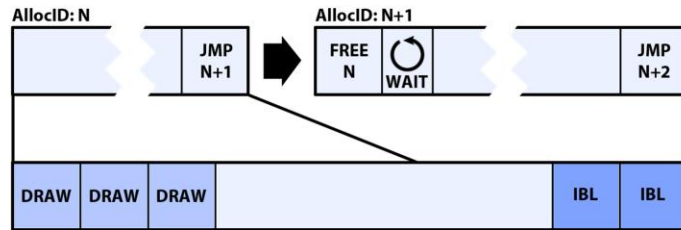


... we also generate the additional data. In this case IBL textures (8x8xRGBA8). And we don't store these into separate memory block, but we use the other side of the same block used for push buffer.



Push buffer generation - example

Memory blocks

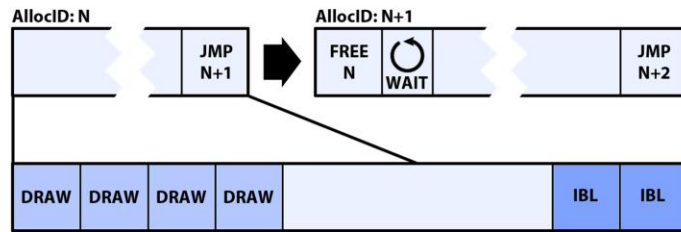


As we add more and more commands, gap between the blocks becomes smaller...



Push buffer generation - example

Memory blocks

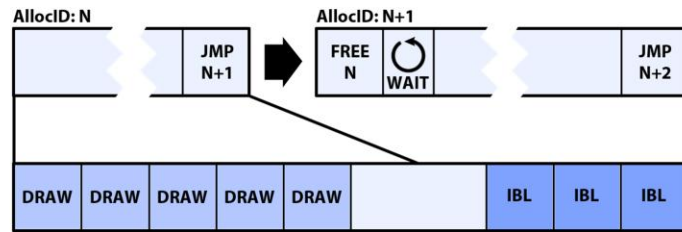


... and smaller ...



Push buffer generation - example

Memory blocks

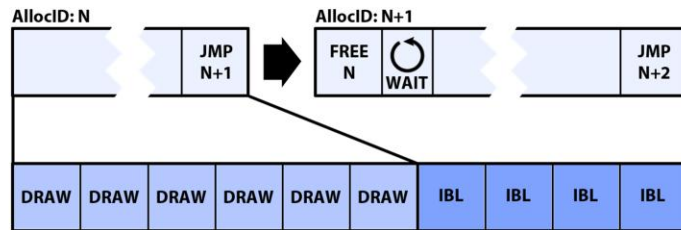


... and even smaller ...



Push buffer generation - example

Memory blocks



... until it closes. Then the block is full and we actually allocate block N+1.

We do this to promote memory locality - IBLs are generated just-in-time and the memory this uses will be freed with the command that uses it.



Conclusion

- » My advice - keep it simple
- » I hope you learned something useful for your next game.
- » Thank you!

Keep it simple - this is what Deferred shading allowed us to do – we have simple rendering pipeline with few optimizations in place to make it run fast & within desired memory budgets. Forward rendering would not allow us to have this level of predictability with all possible material/light combinations on screen.

With deferred shading it's easy to make project specific optimizations that might not be exactly general and reusable for all future projects but are so simple to include/exclude that it makes it worth it.



Questions?

?