



**SIGGRAPH**2011

# **More Performance!**

## Five Rendering Ideas from *Battlefield 3* and *Need For Speed: The Run*

John White (NFS)

Colin Barré-Brisebois (BF3)

*Advances in Real-Time Rendering in Games*



SIGGRAPH2011

**BLACK BOX**

**DICE**



Today we're going to present 5 rendering techniques from the Frostbite 2 engine, the engine behind both Battlefield 3 and Need For Speed. So, let's begin.

# Agenda



- Motivations
- The Techniques
  - Separable Bokeh Depth-of-Field
  - Hi-Z / Z-Cull Reverse Reload Tricks
  - Chroma Sub-Sampled Image Processing
  - Tiled-Based Deferred Shading on Xbox 360
  - Temporally-Stable Screen-Space Ambient Occlusion
- Q&A

*Advances in Real-Time Rendering in Games*

# Motivations



- Frostbite 2 is DICE's Next-Generation Engine for Current Generation Platforms
- 5 Pillars:
  - Animation
  - Audio
  - Scale
  - Destruction
  - **Rendering** ←
- Powers *Battlefield 3* and *Need For Speed: The Run*



*Advances in Real-Time Rendering in Games*

So, FB2 is DICE's next-generation engine for current generation platforms. Its design is based on 5 pillars, such as animation, audio, scale, destruction, and rendering.

As we said earlier, this engine is versatile and flexible enough to power both *Battlefield 3* (a first person shooter) and *Need for Speed: The Run* (a racing game).

## More Info



- Lots of FB2 papers on DICE website  
[publications.dice.se](http://publications.dice.se)
- Also see Alex Ferrier's talk on “1000 points of light” Tues, 2pm, East Building, Ballroom A/B



*Advances in Real-Time Rendering in Games*



Here's an in-game shot of BF3



And here's an in-game shot of Need for Speed: The Run.

# **Separable Bokeh Depth-of-Field**

*Advances in Real-Time Rendering in Games*



# Real World Bokeh - Disc



Photo Courtesy of Mohsin Hasan 2011

*Advances in Real-Time Rendering in Games*

Bokeh refers to the out of focus areas of an image when taking a photograph. Specifically it refers to the shape and the aesthetic quality of the out of focus areas.

Though it is effectively an unwanted side effect of a camera lens system it is artistically desirable. DOF as a general rule has great benefits as a storytelling device to divert the eye to points of interest.

Here is a real world image of Bokeh. The disc shape of the lights in the distant scene smear out into a disc due to the light converging to a point before the sensor plane the light therefore separates radially outwards leading to the energy spreading out into a disc shape . The disc shape is because the lenses are circular.

## Real World Bokeh – Pentagonal



Photo Courtesy of Mohsin Hasan 2011

*Advances in Real-Time Rendering in Games*

Here is a shot with a higher F/Stop. This causes the blades in the camera to constrict the size of the opening of the lens. This blocks some of the incoming light leading to a smaller Circle of confusion and gives a pentagonal shape to the bokeh. This lens clearly has a 5 blade aperture.

At high F/Stop rating the blades almost fully cross over leading to an almost pin hole camera which will exhibit very little bokeh.

# Circle of Confusion Calculation



- Calc per pixel CoC from real world camera parameters
  - Lens Length (derive from FOV)
  - F/Stop
  - Focal Plane
- CoC is a simple MADD on the raw Z Depth  
[Demers04][Jones08]

$$\text{CoC} = \text{abs}(z * \text{CoCScale} + \text{CoCBias})$$

$$\text{CoCScale} = (A * \text{focallength} * \text{focalplane} * (\text{zfar} - \text{znear})) / ((\text{focalplane} - \text{focallength}) * \text{znear} * \text{zfar})$$

$$\text{CoCBias} = (A * \text{focallength} * (\text{znear} - \text{focalplane})) / ((\text{focalplane} * \text{focallength}) * \text{znear})$$

*Advances in Real-Time Rendering in Games*

Using real world camera parameters is important

Avoids the toy town look

More intuitive for people with real world camera knowledge

Inputs are Lens length which is derived from the Camera FOV

F/Stop which is artist controlled. Intuitively is the inverse of the aperture. The Higher the F-stop the lower the blur and vice versa.

Focal plane. What you want in focus. Either key frame in cinematic editor or auto-focus on the item of interest (i.e. hero or car)

# Pre Multiplied CoC



- For 16-bit source pre multiply the CoC by the colour
- Store CoC in alpha
- Recover colour by doing  $\text{col.rgb} /= \text{col.a}$
- Ensure CoC always has a small number so colour can always be recovered

*Advances in Real-Time Rendering in Games*

A technique I use is to premultiply the colour by the CoC . This way you can use simple blurs and avoid halo artifacts where in focus pixels bleed into out of focus pixels. Ensure you have the texture address mode to border zero so off screen taps will not influence the blur.

Also the CoC is stored in the alpha channel so the depth buffer is no longer needed.

We do this step during the down sample from the 720p buffer to 640x360

When the bokeh blur is completed the final colour can be recomputed by dividing the RGB by the A channel. Note all colour values are in linear space here.

We always ensure the CoC is set to at least a small number so all colour can be recovered. This is important as the final result of the bokeh blur texture is used as the input to the 320x180 buffer for bloom generation.

# Blur Process



- Gaussian blur
  - Common in DX9 games. Cheap
- 2D Area samples
  - Limited kernel size before texture tap explosion
- GS expanded Point Sprites
  - Heavy fill rate
  - CryEngine3 DX11 and Unreal Engine 3 Samaritan demo

*Advances in Real-Time Rendering in Games*

Gaussian bokeh is common in games because it is fairly cheap as it can be made separable.

Poisson disc and other shapes such as octagonal are popular but is a 2D sampling technique. Doubling the radius quadruples the taps.

Samaritan and CE3 DX11 bokeh is great. Allows artist definable bokeh shapes with texture rather than uniform . But is fill rate heavy.

# Gaussian vs. real world bokeh



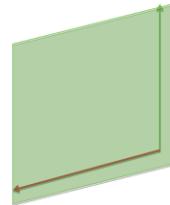
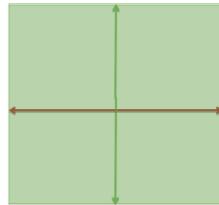
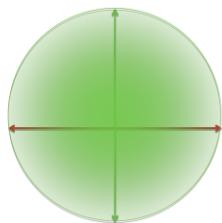
- Arbitrary blurs in image space are  $O(N^2)$
- Gaussian blurs can be made separable  $O(N)$
- What 2D blurs can be made separable?
  - Gaussian
  - Box
  - Skewed Box

*Advances in Real-Time Rendering in Games*

## Other separable blurs



- Gaussian, Box and Skewed Box



*Advances in Real-Time Rendering in Games*

Gaussian. Two passes horizontal and vertical . Weights follow a Gaussian distribution so samples close to the centre have a higher influence. Resultant shape is a bell shape to the energy distribution.

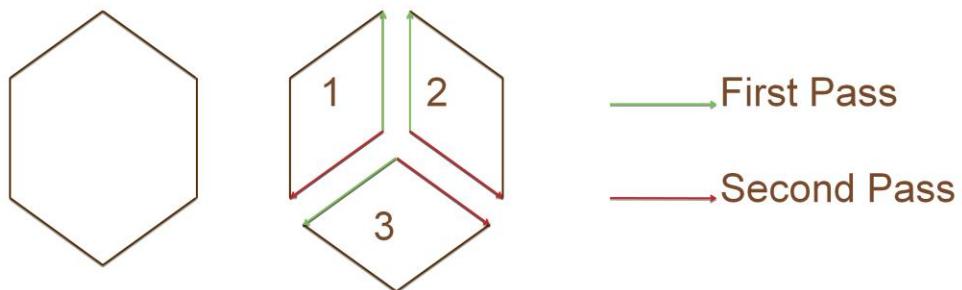
Box. Like the Gaussian but all weights are equal. Stops the mushy look but gives a strange box shape to the bokeh. Looks uglier

Skewed box. Like the box but the horizontal and vertical axes non orthogonal. Note I have deliberately sampled in a simple direction only from the sample being blurred.

# Hexagonal Blurs



- Decompose a hexagon into 3 rhombi
- Each rhombi can be computed via separable blur
- 7 Passes in total. 3 shapes x 2 blurs + 1 combine



*Advances in Real-Time Rendering in Games*

By doing a skewed box at 120deg with equal length we end up with a rhombus shape. By doing this a total of 3 times on the source we have 3 rhombi blurs radiating away from the central point.

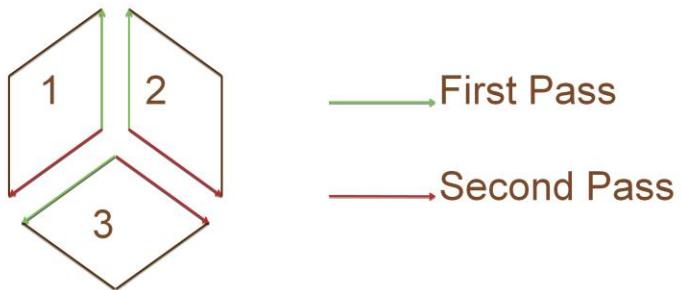
We can then average all 3 results together leading to a hexagon.

We also apply a half sample offset to stop overlapping rhombi. Other wise you'll end up with a double brightening artifact in an upside Y shape.

# Hexagonal Blurs – Pass Reduction



- Hexagonal blur using a separable filter
- But 7 passes and 6 blurs is not competitive
- Need to reduce passes

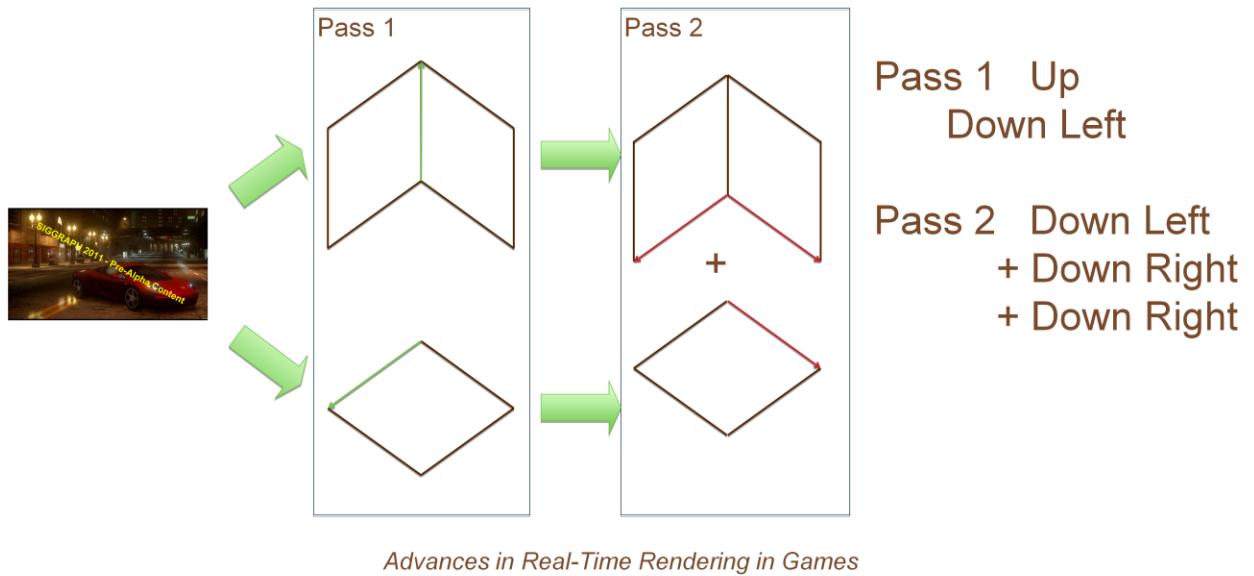


*Advances in Real-Time Rendering in Games*

Looks good but frankly 7 passes is not going to cut it. Gaussian is way ahead in performance and for reasonable filter kernel widths the single pass Poisson disc is going to beat it. A 7 pass linear time algorithm won't beat an  $N^2$  algorithm for suitable small value of N. We need to reduce it.

So looking at the above we notice the first pass blur for Rhombus1 and Rhombus2 are the same. Also First pass blur on Rhombus 3 shares the same source so we can reduce this down to 2 passes (With a little MRT help)

# Hexagonal Blurs – Pass Reduction 1

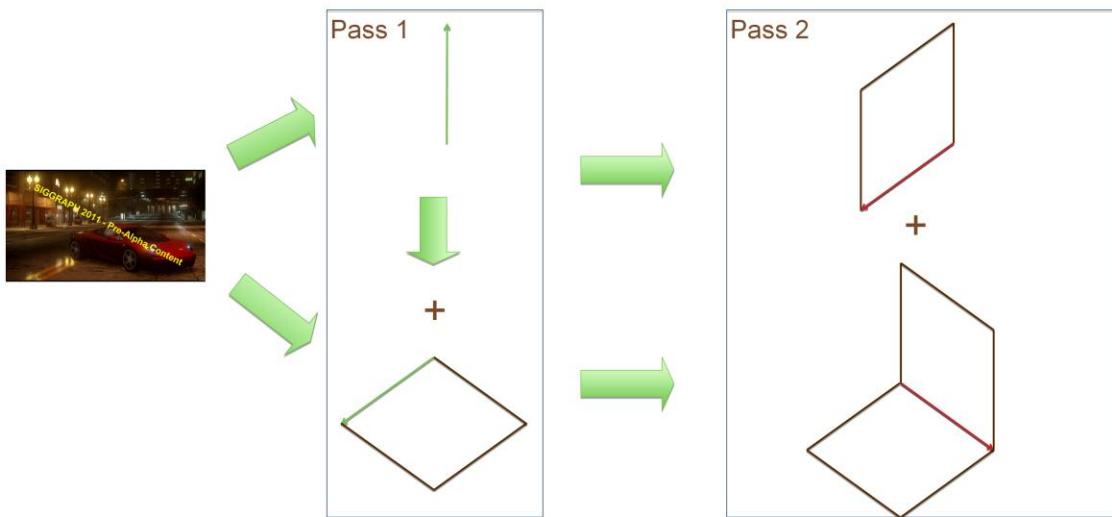


Great 2 Passes in total but Pass 1 has to output two MRTs. And pass 2 has to read from 2 MRT sources.

Total of 5 blurs now which is also 1 reduced and final combine pass is part of pass2.

This is still not optimal. We have 2 identical blurs down and right in pass 2 but they read from different sources. Can we reduce this further?

# Hexagonal blurs – Pass reduction 2



*Advances in Real-Time Rendering in Games*

Now in pass 1 we output the up blur to MRT0 as usual but we also add it onto the result of the down left blur before outputting to MRT1

Now in pass 2 the downright blur will operate on Rhombi 2 and 3 at the same time.

Great 1 blur less!

So Does it actually work

# Hexagonal Bokeh



*Advances in Real-Time Rendering in Games*

# Hexagonal Bokeh



SIGGRAPH 2011 - Pre-Alpha Content

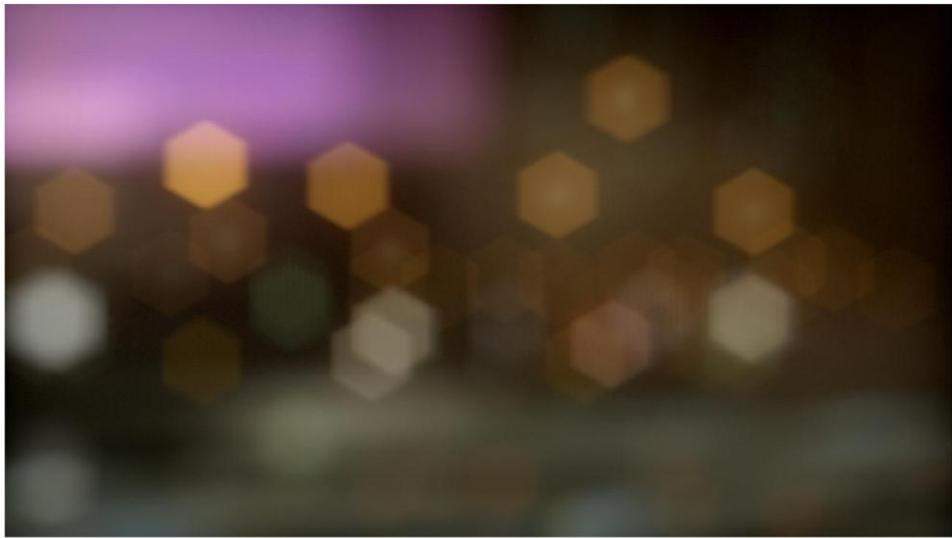
*Advances in Real-Time Rendering in Games*

# Hexagonal Bokeh



*Advances in Real-Time Rendering in Games*

# Hexagonal Bokeh



*Advances in Real-Time Rendering in Games*

Example with extreme filter kernel.

Note there are some bright blobs in some of the hexagons. This is an artifact of a later pass to composite the coronal particles and not of the hexagonal blur process.

# Hexagonal vs Gaussian



- Gaussian

- 2 Passes with a total of 2 blurs

- Hexagonal

- 2 Passes (3 resolves) with a total of 4 blurs
  - BUT each blur only needs half the taps therefore same #taps
  - BUT each tap contributes equally unlike Gaussian so need less taps for a given aesthetic filter kernel width!
  - PLUS We can improve further

*Advances in Real-Time Rendering in Games*

How do we measure up with the Gaussian.

We are doing the same number of taps but our bandwidth has increased. Total of 50% more.

BUT we do have some advantages over Gaussian. We need less taps for a wanted kernel size as each sample contributes equally.

And we can improve this further for very large kernel sizes in sub linear time.

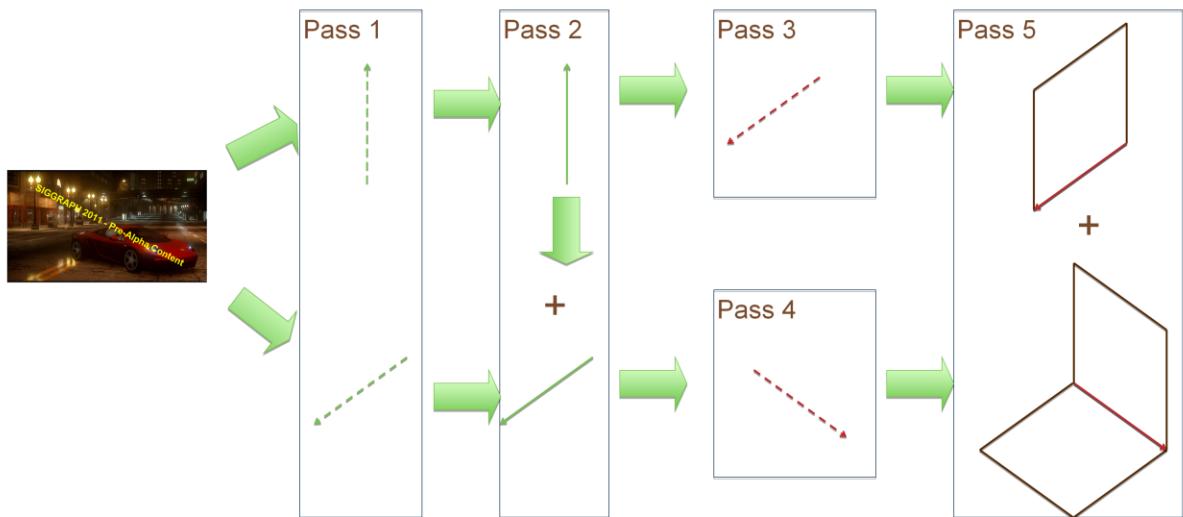
# Iterative Refinement



- Because we have equal weighted blurs can use iterative refinement on the blurring [Sousa08]
- Multiple passes fill in the under-sampling
- Dual iteration blur needs a total of 5 passes with a total of 8 half blurs.

*Advances in Real-Time Rendering in Games*

# Iterative Refinement



*Advances in Real-Time Rendering in Games*

## Pseudo Scatter filter



- Proper bokeh should have its blur scattered to its neighbours
- However pixel shaders are designed to gather they can't scatter the results
- Typical blurs default the filter kernel to the CoC of pixel
- Instead, default to big CoC and reject based on the sampled texel CoC
  - Extra method to stop bleeding artifacts and can sharpen up smooth gradients

*Advances in Real-Time Rendering in Games*

## Hi Z culling



- When downsampling the premultiplied CoC buffer output the computed CoC as depth
- You can then draw the plane at Z depth of 0.001f
- In focus pixels will be quickly rejected by Hi Z
- Same for iterative refinement
- Draw undersample pass at higher Z value, fine at small Z value
  - Requires an explicit copy afterwards to re-fill

*Advances in Real-Time Rendering in Games*

# **Hi-Z / Z-Cull Reverse Reload Tricks**

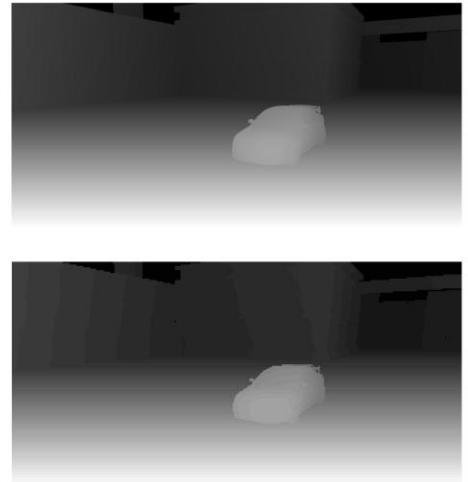
*Advances in Real-Time Rendering in Games*



## Hi-Z (1/)



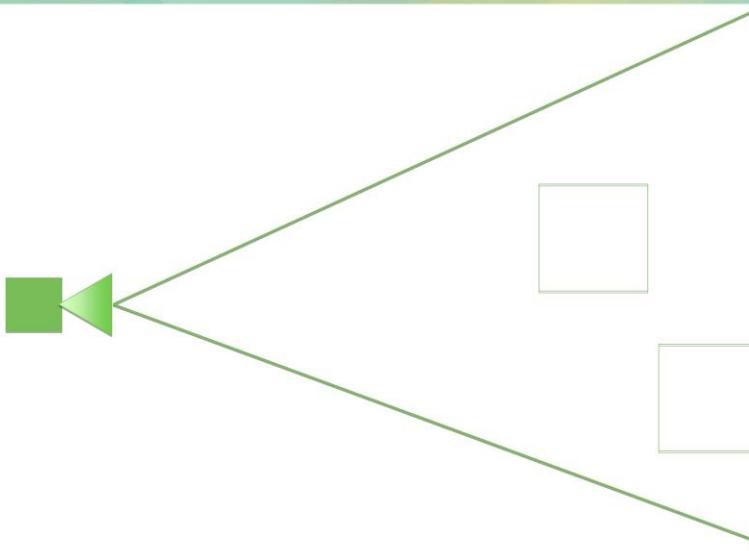
- Ubiquitous on modern hardware
- Stores a Low Res version of the Z buffer
- Can use this to conservatively reject groups of pixels
- Saves fragment shading known occluded pixels



*Advances in Real-Time Rendering in Games*

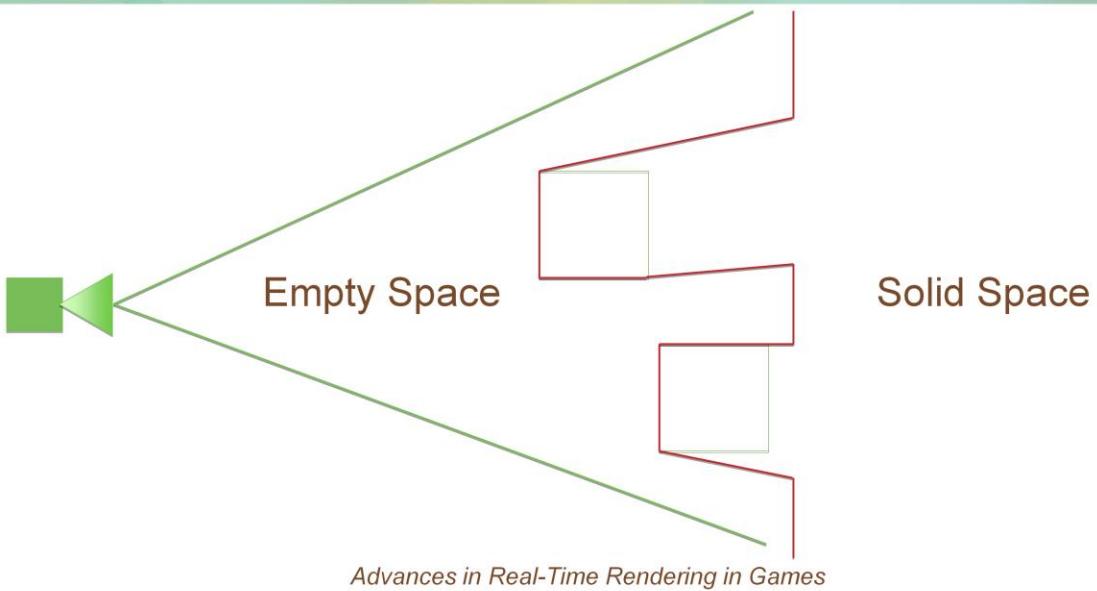
Hi Z is very common on modern graphics hardware. Its an important HW optimisation as it allows pixel groups to be discarded before they are shaded. Without this you'll end up wasting time shading pixels which later are rejected because they fail the depth test.

This Special Z Buffer is usually at a lower resolution than the actual Z buffer so you will end up shading some pixels which are later discarded. Usually this is not huge.



*Advances in Real-Time Rendering in Games*

So here is a top down view on a typical view. In it we have already drawn two cubes.



*Advances in Real-Time Rendering in Games*

The red line denotes how the ZBuffer looks when viewed in perspective space. So everything on the right hand side is in solid occluded space. If we render an object or part of an object here he can ignore it.

# Volume Rendering (1/)



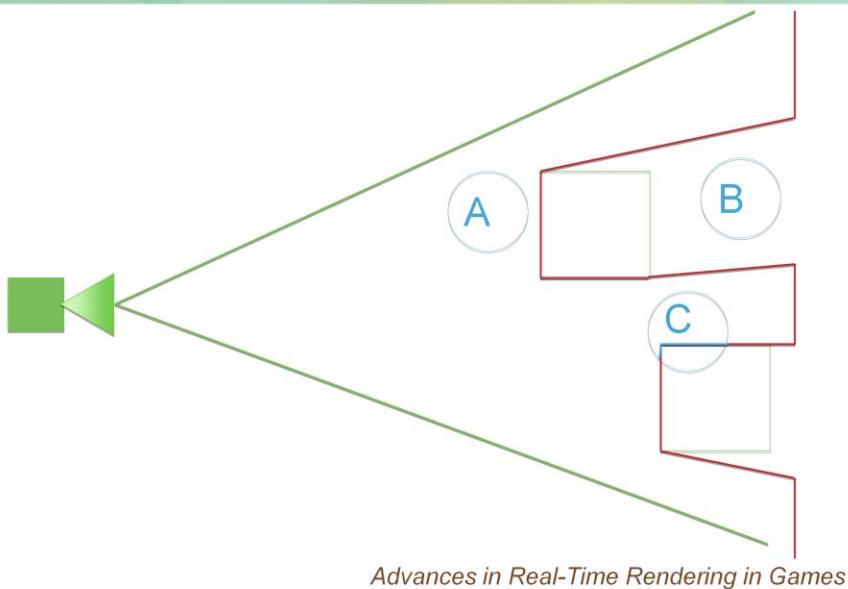
- In Deferred Renderers it is common to reproject screen pixels back into world space
  - Common for lights (point, spot, line)
  - Shadow volumes
- Draw a convex bounding polyhedron projected in screenspace
- In shader, reject pixels which are not in the volume bounds

*Advances in Real-Time Rendering in Games*

It is common in deferred rendering to draw convex polyhedra to represent the volume they enclose. You don't actually care about the surface. So draw the polyhedra to the screen and reproject back into world space from the pixels it covers in screenspace

However not all pixels will reproject into the bounds of the volume

## Volume Rendering (2/)



- A. A will waste fill rate
- B. Hopefully should fail quickly with HiZ. Problem with holes
- C Some wasted fill

Can use Stencil capping and Depth bounds

# Reverse Hi-Z Reload (X360)

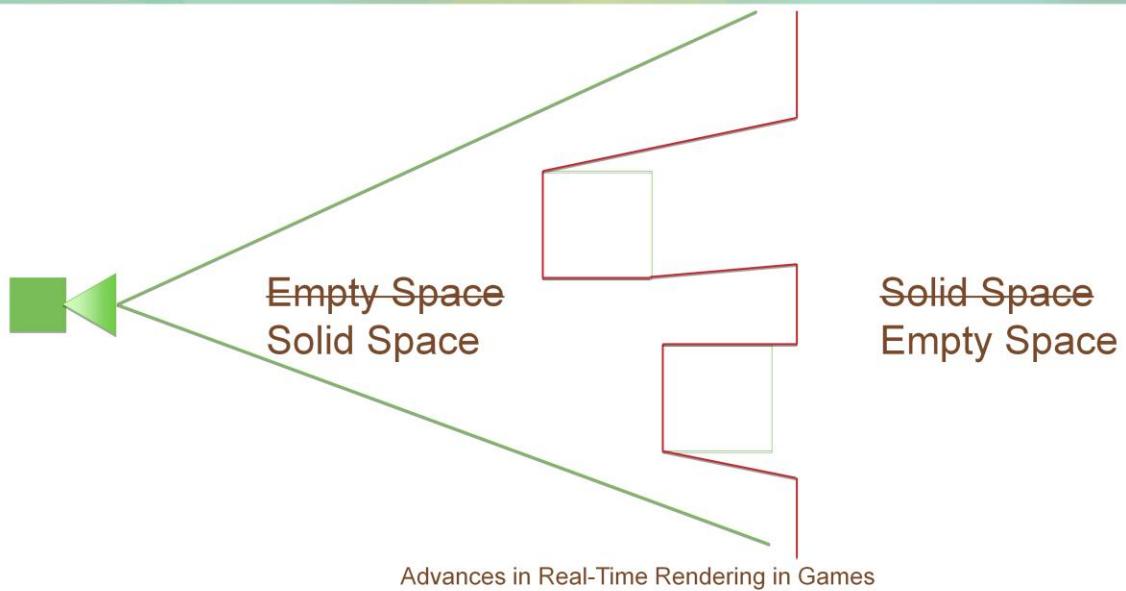


- Alias a Render Target on existing depth buffer
- Init aliased RT to D3DHIZFUNC\_GREATER\_EQUAL
- Draw Full screen quad
  - NULL Pixel Shader
  - Zfunc == Never
- Hi-Z is now primed in reverse
- Similar technique on Playstation3 (See DevNet)

*Advances in Real-Time Rendering in Games*

The Z buffer can be thought of a per pixel delimiter between empty space and solid space. We can make use of this property to reload the Z buffer in reverse. This is super cheap on X360. Bit more expensive on PS3

# Reverse Hi-Z (1/)



So what does our Z Buffer look like now?

We have the space from the camera to the nearest surface as the occluded solid space . And beyond that is empty. How does this help us?

## Reverse Hi-Z (2/)



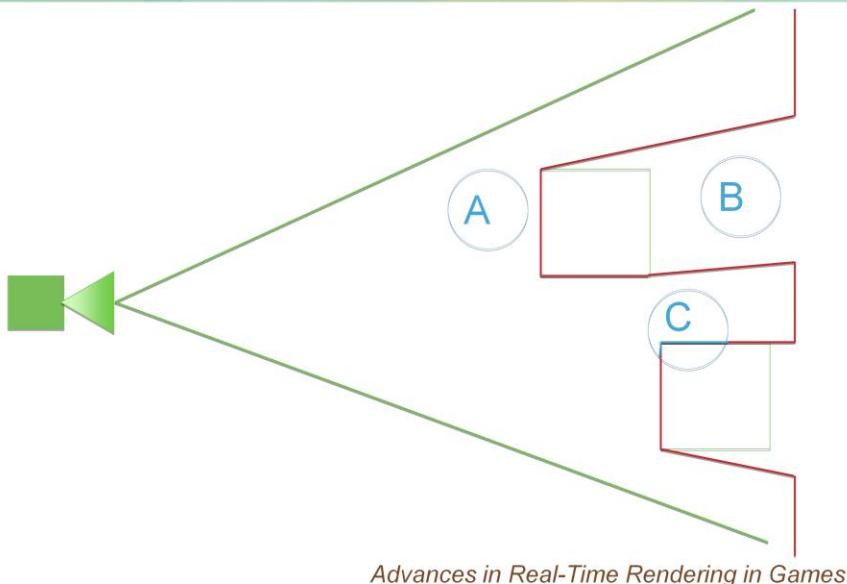
- The GPU will now cull fragments if they are closer than the value in the depth buffer
- By rendering the backfaces of convex polyhedra, pixels beyond the faces will quickly reject
- If the camera is inside the volume then only pixels inside the volume will pass
- Perfect for cascaded shadow maps

*Advances in Real-Time Rendering in Games*

So any the GPU will now quickly cull pixels if they are too close to the camera, and trivially accept distant ones.

Because we are rendering convex polyhedra we can instead render the backfaces. How does this help us? Lets go back to the example from before.

## Reverse Hi-Z (3/)



We render the backfaces of A but they fail the Hi Z as they are closer than the current value. B will pass but this will be wasted fill. C will correctly pass for the pixels inside the volume.

So we have traded the cases but they're are some advantages.

We are rendering backfaces so we don't care about cases of the camera cutting inside the volume. Also the erroneous ones are projected smaller. The usual tricks of stencil capping and depth bounds still work

Also we can use SW occlusion techniques to cull B before it goes near the GPU. Finally another trick is to render the front faces of the volumes before the reverse reload and count how many pixels pass. This is a conditional query after the reverse reload. This should then discard B

## Reverse Hi-Z for CSM rendering



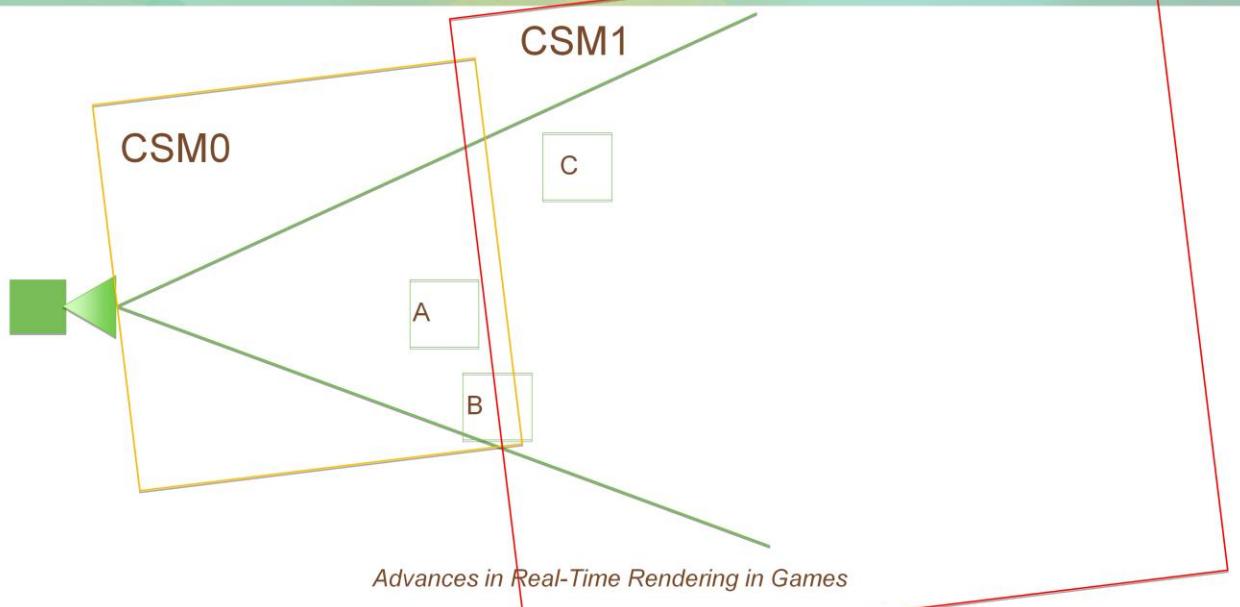
- Each cascade for a directional light is bounded by a cuboid in world space
- Only world space pixels inside the cuboid will project onto the shadow map
- By drawing the cuboid backfaces only these pixels will pass the reverse Z test

*Advances in Real-Time Rendering in Games*

Perfect for CSM. CSM's are designed so the volumes they enclose intersect and enclose the camera near plane.

Not true for later cascades but they will enclose the previous cascade.

# CSM Cuboids



# CSM Cuboids



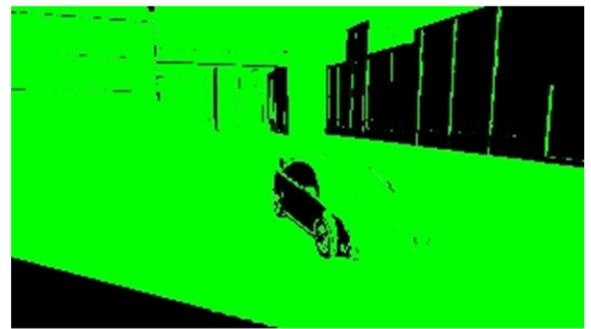
- Evaluate as a separate pass [Sousa08]
  - Input is depth buffer
  - Creates a L8 mask texture input into directional light pass
- Can do a prior full screen pass to tag back facing pixels wrt to light source in stencil
  - Heuristic on sun angle with camera
- Potential for  $\frac{1}{4}$  res with bilateral upsample
- Stencil is updated to denote already processed pixels

*Advances in Real-Time Rendering in Games*

# Reverse Hi-Z CSM (1/)



## ▪ Cascade 0



*Advances in Real-Time Rendering in Games*

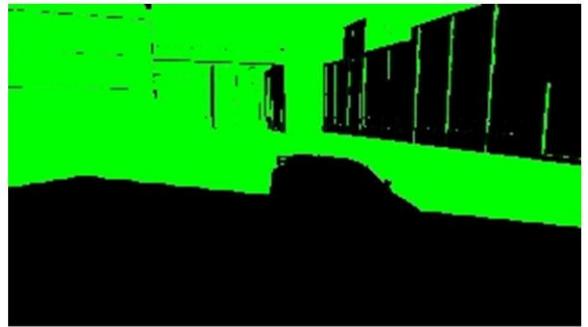
Red denotes the shadow mask value. This is later read in as a way of modulating the diffuse lighting calculation.

Green is the current state of the stencil buffer.

## Reverse Hi-Z CSM (2/)



### ■ Cascade 1

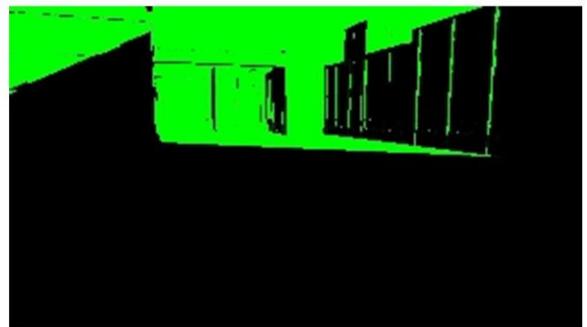


*Advances in Real-Time Rendering in Games*

## Reverse Hi-Z CSM (3/)



### ■ Cascade 2



*Advances in Real-Time Rendering in Games*

## Reverse Hi-Z CSM (4/)



### ▪ Cascade 3

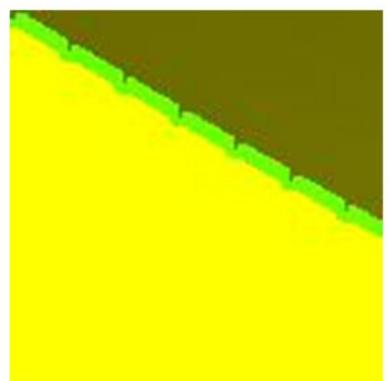
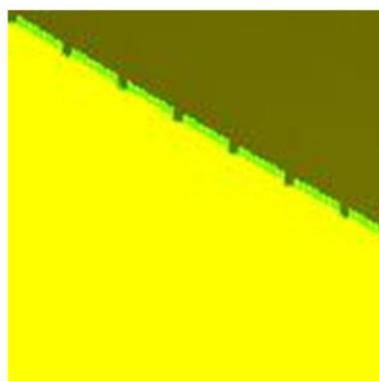
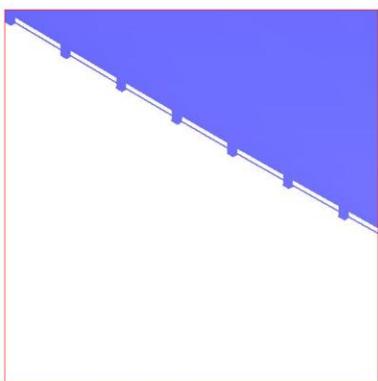


*Advances in Real-Time Rendering in Games*

# Min/Max Shadow Maps (1/)



- Downsample and dilate SM, keeping track of min and max depths



*Advances in Real-Time Rendering in Games*

## Min/Max Shadow Maps (2/)



- Dilated min/max SM allow us to know if a pixel is ...
  - Fully In shadow
  - Fully out of shadow
  - Partially in shadow (conservatively)
- Draw each cuboid twice
- First Pass
  - Single simple tap shader
- Second Pass
  - High quality PCF shader

*Advances in Real-Time Rendering in Games*

# Min/Max Shadow Maps Simple Pass



- If  $Z < \text{MinMax}.min$  return  $(1,1,1,1)$
- If  $Z > \text{MinMax}.max$  return  $(0,0,0,1)$
- If  $Z < \text{MinMax}.min$  return  $(0,0,0,0)$

Mask



Stencil



*Advances in Real-Time Rendering in Games*

Note in this I'll denote white in the mask for the trivially accepted pixels, and red for the complex PCF ones.

Note how the alpha test ensure the stencil remain untouched for the pixels which couldn't be trivially accepted or rejected.

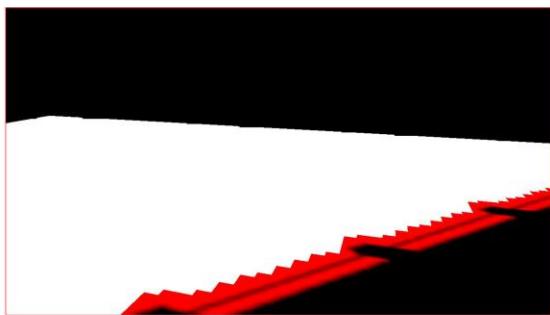
First pass done. Now the second pass

# Min/Max Shadow Maps PCF Pass (1)



- Second pass is standard PCF filter

Mask



Stencil



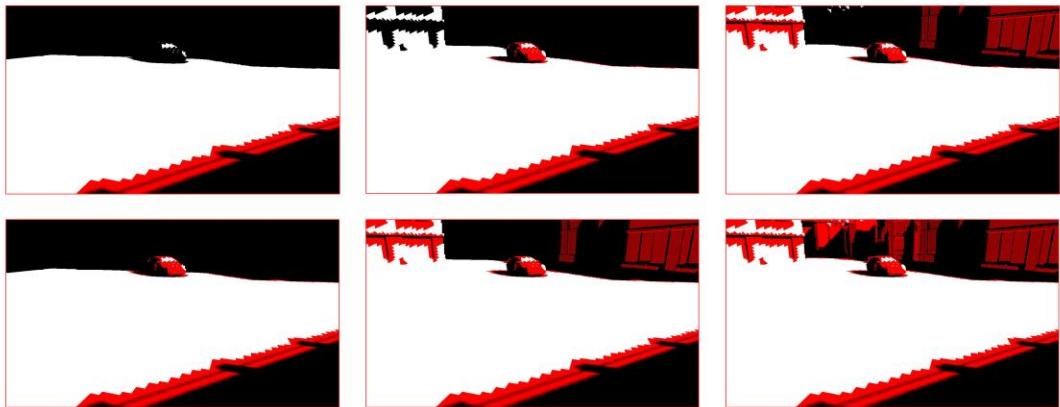
*Advances in Real-Time Rendering in Games*

Here the red shows the newly touched pixels and the stencil is now updated.

## Min/Max Shadow Maps (3/)



- Do for all cascades



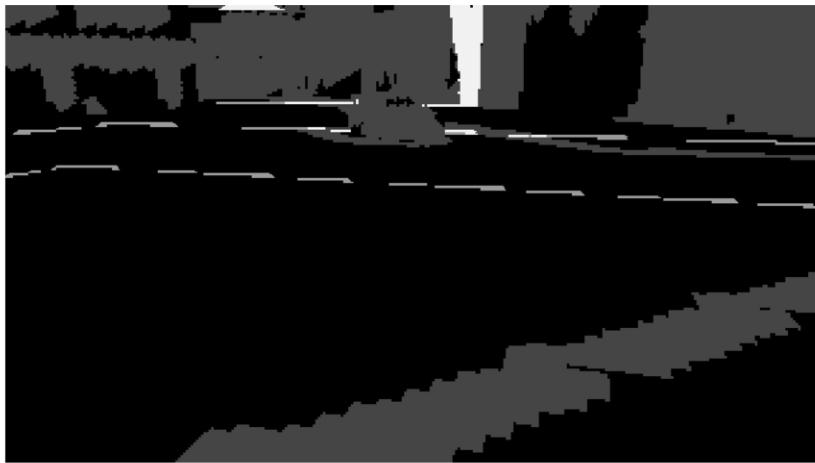
*Advances in Real-Time Rendering in Games*

And here is the update for the later cascades.

## Min/Max Shadow Maps PCF Pass (2/)



- Final Overdraw



*Advances in Real-Time Rendering in Games*

This image shows the total pixel overdraw. A lot of it is 1 with a smaller percentage at 2. Note for the 2 overdraw case 1 of them was the fast single tap shader

# Conditional Tests



- When we render the cuboid we can count how many pixels pass
- If Zero then no pixels will sample from the shadow map
  - So why even render the shadow map!
  - Draw the cuboid first and only if pixels pass draw the actual shadow map
- Zero passed pixels occur for two reasons
  - All pixels are further away
  - All pixels have been touched by a closer cascade (stencil cleared)

*Advances in Real-Time Rendering in Games*

Also by doing this we can reuse the shadow map for each cascade. Nice memory saving. Caveat is problems if you later need them to shadow fwd rendered transparent.

# **Chroma Sub-Sampled Image Processing**

*Advances in Real-Time Rendering in Games*



# Chroma Sub-Sampling



- Not a new idea. Used in TV broadcasts as well as Jpeg/Mpeg compression
- Decompose image into luminance and chroma
- Store Luma at full res only. Chroma at lower



*Advances in Real-Time Rendering in Games*

# Chroma Sub-Sampling - Motivation



- Post processing requires lots of bandwidth
  - Easy to optimise ALU down
  - Quickly hit a performance ceiling, especially for 16bpp pixels
- Reading and writing a 720p image with 16bpp components is 14MB bandwidth
- Assuming 14GB/Sec Bandwidth and perfect cache usage this is 1ms for a single pass

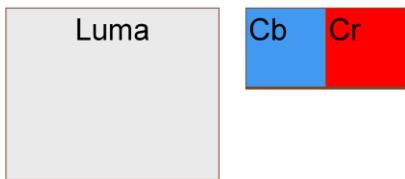


*Advances in Real-Time Rendering in Games*

# Chroma Sub Sampling - Motivation



- Instead reduce down to Luma only
- $\frac{1}{4}$  of the bandwidth required
- Requires extra processing on the Colour
  - But this can be 2 channel at  $\frac{1}{4}$  res ( 1/8 original size )
  - Also can get away with less taps



*Advances in Real-Time Rendering in Games*

# Chroma Sub Sampling



- Bandwidth is reduced to 1/4
- So, are the shaders now 4X quicker?
- No. We are ALU bound again
  - Texture units and ALU are designed for 4 component SIMD
  - We are only using 1 component
  - Need to pack 4 luma values together and process together

*Advances in Real-Time Rendering in Games*

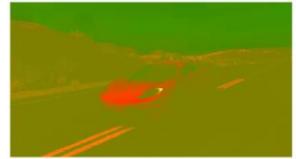
# Chroma Sub Sampling



- Pack 4 adjacent pixels together into one RGBA pixel
- Only need 1 texread to get 4 luma values
- So a 1280x720 luma buffer is a 320x720 ARGB buffer
- With the packed buffer bilinear filtering is not correct
- Have to manually filter horizontally using DOTP
  - Colin will go into this later

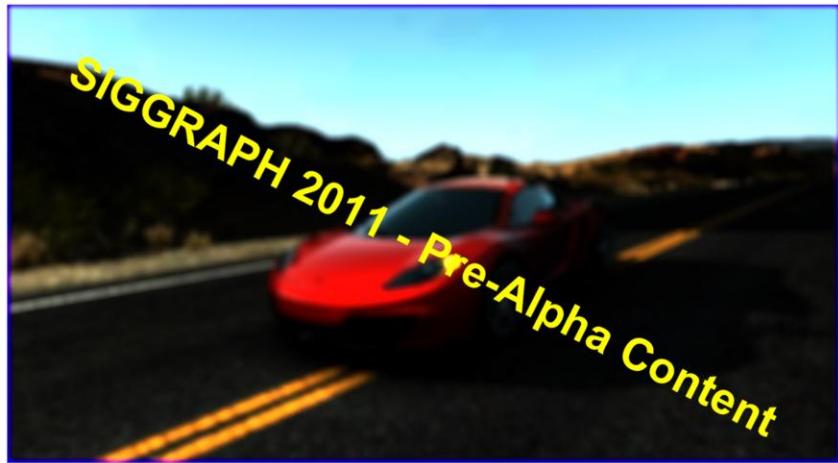
*Advances in Real-Time Rendering in Games*

# Chroma Sub Sampling



*Advances in Real-Time Rendering in Games*

# Chroma Sub Sampling



*Advances in Real-Time Rendering in Games*

# Butterfly Packing



- Overlay each quadrant into ARGB
- Mirror around the image center point
- Bilinear now works except across the boundaries
  - Re-draw a strip with additive blend and swizzling
  - R<->G and B<->A for horizontal
  - R<->B and G<->A for vertical
- Radial blurs just work

*Advances in Real-Time Rendering in Games*

# Butterfly Unpacking



- When rendering fullscreen quad, need two attributes

0,0 640,-640,-360,-360	2,0 -640,640,-360,-360
0,2 -640,-640,360,-360	2,2 -640,-640,-360,360

Use UV in Mirror Mode  
Dot with saturated second component

*Advances in Real-Time Rendering in Games*

Also need a half texel offset before saturate. Omitted for clarity.

## Future Work



- Use for hexagonal blurs
- Output packed tonemap
  - Only perform temporal AA for luma
  - Packed luma used for MLAA passes

*Advances in Real-Time Rendering in Games*

# Tiled-based Deferred Shading on Xbox 360

*Advances in Real-Time Rendering in Games*



Let's now talk about our tiled-based approach to deferred shading, but this time for Xbox 360.

Many of you have probably attended or read Christina Coffin's GDC talk about tiled based deferred shading on SPUs. This time, Christina, Johan Andersson and I teamed-up and came-up with a somewhat different approach, tailored for the Xbox 360.

# Tiled-based Deferred Shading? (1/)



- Want more interesting lighting with more dynamic lights!
- Platform is fixed → better usage of *rendering resources*
- [Swoboda09] and [Coffin11] on Playstation3™, by [Andersson09] in DirectCompute, and other hybrids
- Christina, Johan and I teamed-up for this version on 360
- Load-balance and compute lighting where it matters:
  1. Divide the screen in screen-space tiles
  2. Cull analytical lights (point, cone, line), per tile
  3. Compute lighting for all contributing lights, per tile

*Advances in Real-Time Rendering in Games*

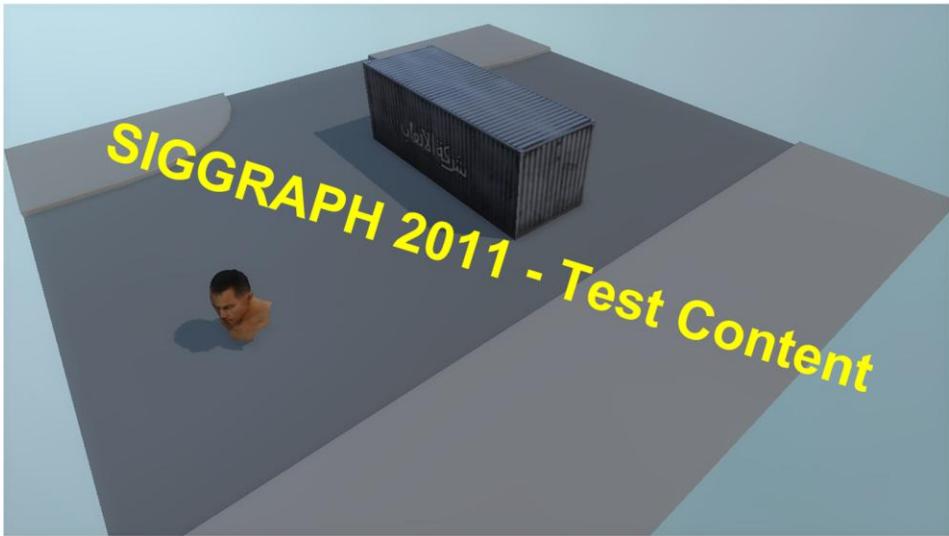
Again, why tiled based deferred shading? Basically, we want more interesting lighting. One way to achieve this is via more dynamic lights, and this is where deferred shading shines!

Unfortunately, the 360 won't change. We need to figure out ways to better use the rendering resources that we have. This is where tiled-based deferred shading comes in.

This has been done before, by Matt Swoboda and Christina Coffin on PS3, by Johan Andersson on DirectCompute and others have come up with hybrids. Here, we'll focus on our approach for the 360.

A quick recap, tiles allow us to load-balance and compute lighting where it matters. First, we divide the screen in tiles. Then, we cull all analytical/pure-math lights, such as point, line and cone lights (here, no lights that project a shadow or a texture). Once this is done, we can compute the lighting only for the valid lights in each tile.

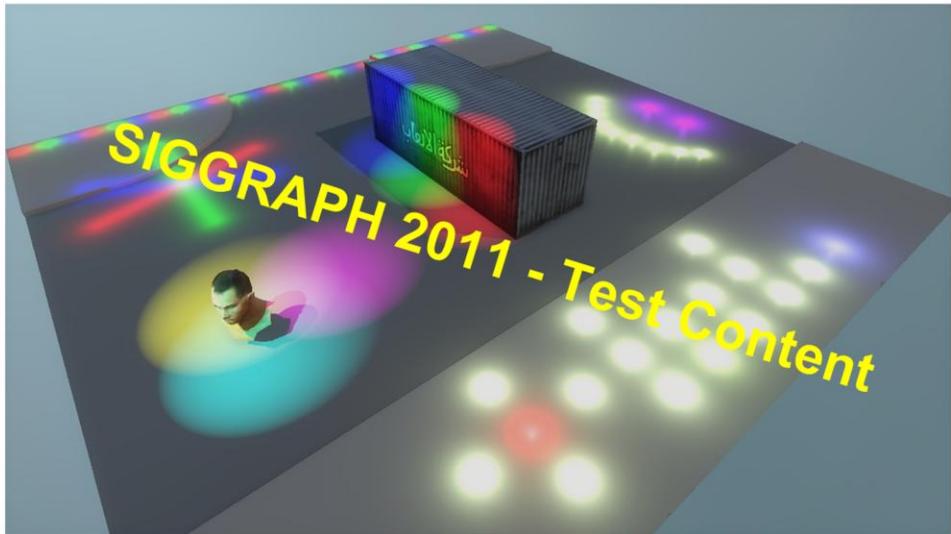
## Tiled-based Deferred Shading? (2/)



*Advances in Real-Time Rendering in Games*

Here's a very complex programmer-designed scene :)

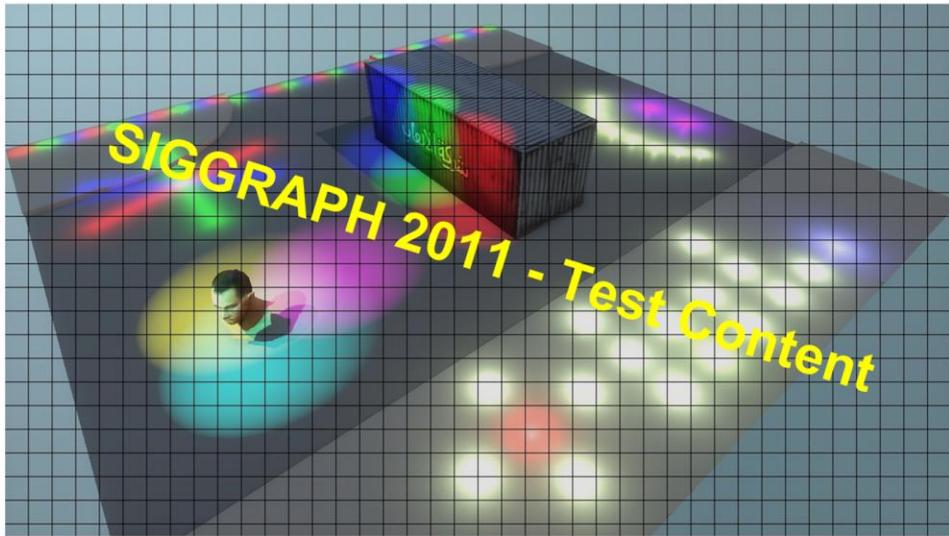
## Tiled-based Deferred Shading? (3/)



*Advances in Real-Time Rendering in Games*

We then add some lights to the scene, but this not optimal...

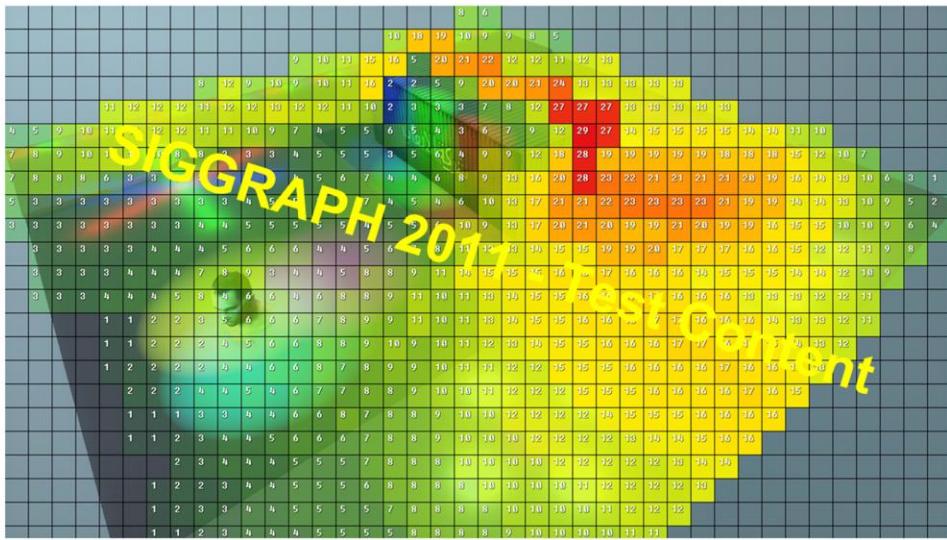
## Tiled-based Deferred Shading? (4/)



*Advances in Real-Time Rendering in Games*

So we divide the screen in tiles...

# Tiled-based Deferred Shading? (5/)



*Advances in Real-Time Rendering in Games*

And we cull the lights. Once the culling is done, we know exactly how many lights should be rendered in each tile. Here, we provide a colored grid to help our amazing lighting artists make sure the lighting is in budget.

# How Does This Fit on Xbox 360?



- We don't have DirectCompute nor SPUs on 360...
- Fortunately, *Xenos* is powerful, and will crunch ALU
- For maximal throughput, data at rendering time has to be cleverly pre-digested
- If timed properly, we can also use the CPUs to help the GPU along the way...
- GPU is better at analyzing a scene than CPUs...
- Let's use it to classify the scene

Advances in Real-Time Rendering in Games

So, how does this fit on 360? Of course, we don't have DirectCompute nor SPUs...

Fortunately, Xenos is a pretty powerful GPU, and will crunch ALU if asked to do so

If we pre-digest data, maximum throughput can be achieved at rendering time

Also, if we time things properly, we can use the CPU to help the GPU along the way

One thing to not forget is that the GPU is definitely better than the CPU at analyzing a scene. This means we can definitely use it to classify our scene.

# GPGPU Culling (1/)



- Our screen is divided in 920 tiles of 32x32 pixels
- Downsample and classify the scene from 720p to 40x23 (1 pixel == 1 tile)
  - Find each tile's Min/Max depth
  - Find each tile's material permutations
  - Downsampling is done in multi-pass and via MRTs
- Similar to [Hutchinson10]

*Advances in Real-Time Rendering in Games*

This is why we have a GPGPU approach to culling.

First, our scene is divided in 920 tiles of 32x32 pixels

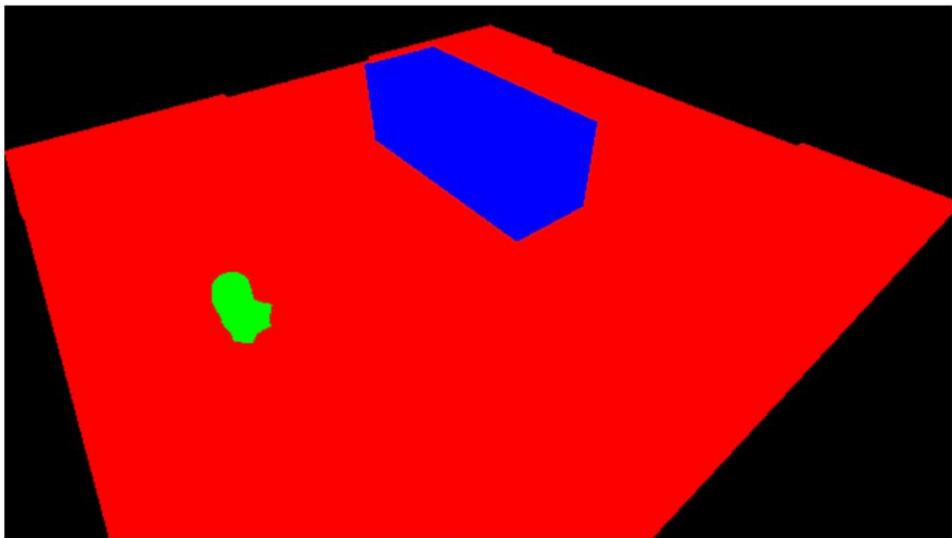
We downsample and classify the scene from 720p to 40x23. Which means, in the end, 1 pixel of classification equals to 1 tile on screen.

Classification is done to:

1. Identify the min and max depth value for each tile
2. Identify all the material permutation in each tile

This is a multi-pass process which can definitely be achieved via MRTs.

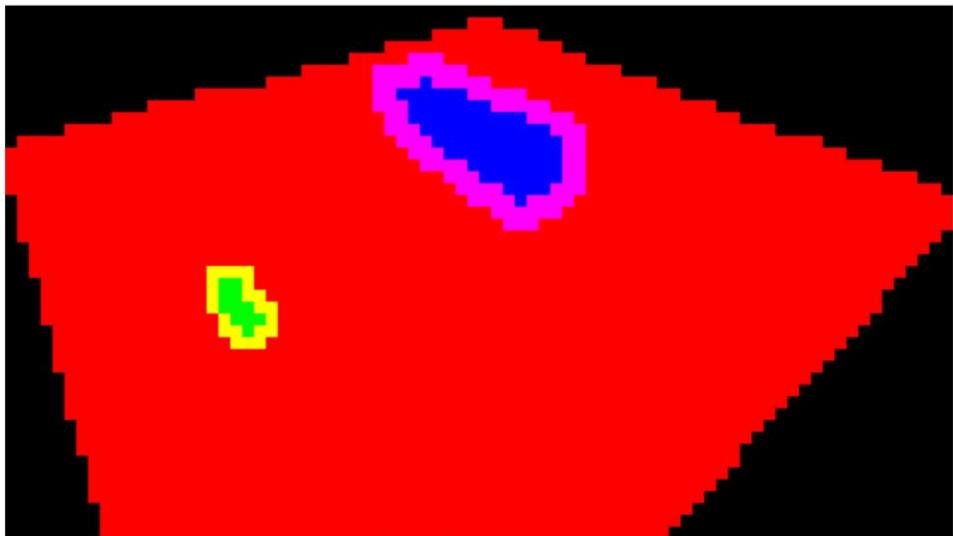
## GPGPU Culling (2/)



*Advances in Real-Time Rendering in Games*

Here's an example of the material classification. If we go back to our scene, we have 3 materials. Our default material in red, skin shading in green, and metallic in blue.

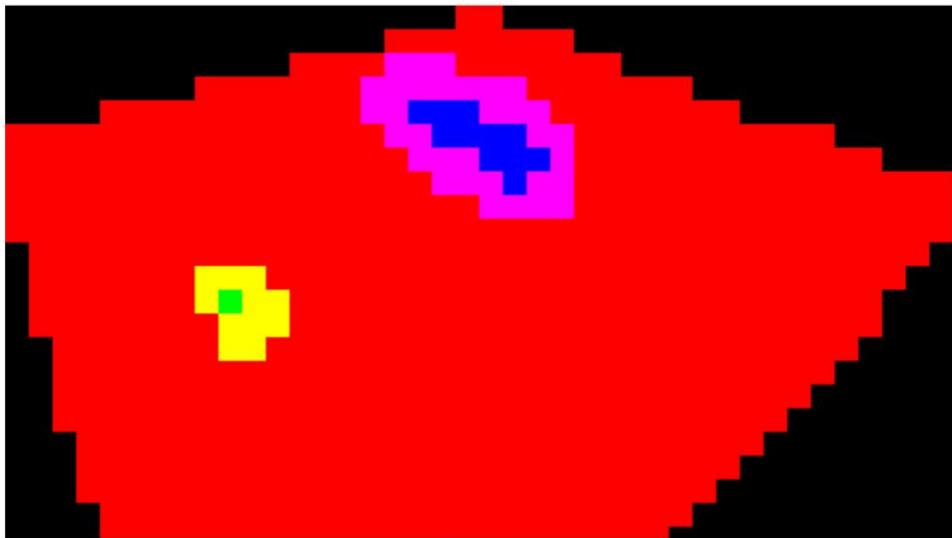
## GPGPU Culling (3/)



*Advances in Real-Time Rendering in Games*

As we downsample (by doing manual bilinear taps), we combine the materials in the final output color. We can see to the right how the downsampling around the head gives yellow, which is the combination of red and green. Same for the red and blue, giving magenta.

## GPGPU Culling (4/)



*Advances in Real-Time Rendering in Games*

Skipping a couple steps here, but this is what we get in the end. This 40x23 texture will tell us exactly which combination of materials we have in each tile. In a parallel MRT, we also downsampled and stored the min/max depth.

With this information we are now ready to use the GPU to cull the lights since:

- We know all lights in the camera frustum
- We know depth min/max and permutation combinations for each tile (and sky), mini-frustums

- Build mini-frustas for each tile
- Cull lights against sky-free tiles in a shader
- Store the culling results in a texture:
  - Column == Light ID
  - Row == Tile ID
- Actually, 4 lights can be processed at once (A-R-G-B)
- Read back the contribution results on the CPU and prepare for lighting!

*Advances in Real-Time Rendering in Games*

First, we quickly read back this texture on the CPU and we build mini-frustas for each tile. We know which tiles are sky, from the previous classification pass, so those can be skipped since they never will get lit.

We can then use the GPU to cull all lights against the tiles.

We store the culling results in a texture, where each column represents the light ID and the row the tile ID.

Actually, we can do 4 lights at a time, and store the results in ARGB. Once this is done, we fast-untiling using the Microsoft XDK sample and read-back on the CPU. We're basically ready for lighting.

# I Need a Light



- Parse the culling results texture on CPU
- For each light type,  
For each tile,  
For each material permutation,
  - Regroup & set the light parameters for the PS constants
  - Setup the shader loop counter\*
  - Additively render lights with a single draw call (to the final HDR lighting buffer)

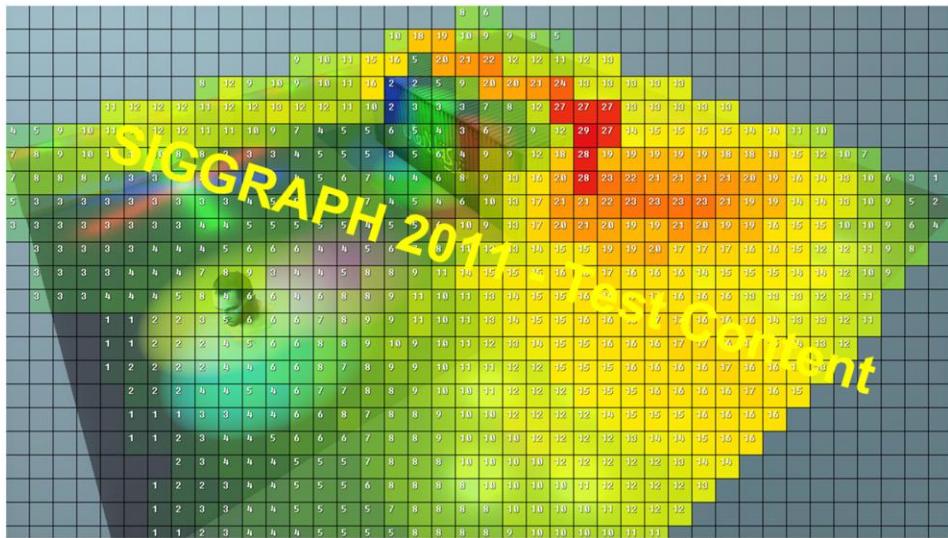
*Advances in Real-Time Rendering in Games*

As mentioned previously, we parse the culling results texture on the CPU.

For each light type, for each tile, and for each material permutation, we regroup and set the light parameters to the pixel shader constants, we setup the shader loop counter and we additively render lights with a single draw call to the final HDR buffer.

Since we don't have an infinite number of constants, we have a set limit on how many lights we can render per-tile. Since we do this per light type, and per material permutation, it's a good load of lights (i.e. 32-64 lights of each type, per tile) :)

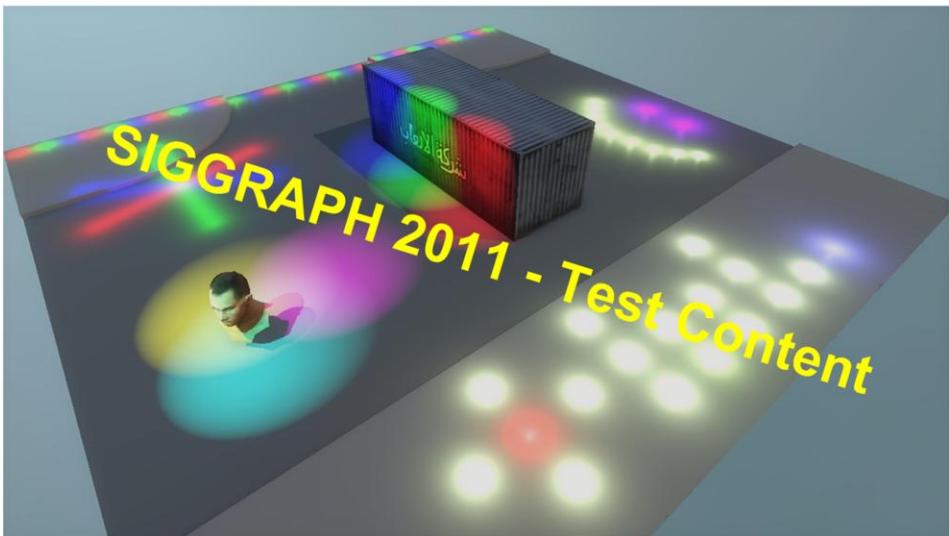
# Results (1/)



Advances in Real-Time Rendering in Games

Again, our final result, with tiles.

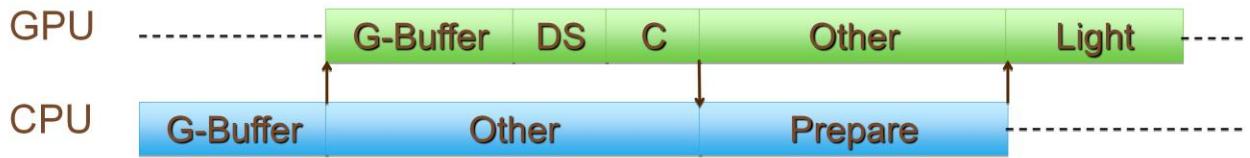
## Results (2/)



*Advances in Real-Time Rendering in Games*

And without the tiles

# Timeline



- DS: Downsample / Classify
- C: Cull
- Light: Lighting pass
- We kick CPU jobs from the GPU using a MEMEXPORT shader (i.e.: write token at specific address, job starts)

*Advances in Real-Time Rendering in Games*

Quickly, here's a timeline of our CPU/GPU synchronization. I collapsed all the CPU threads into one for simplicity.

To make sure our CPU jobs start right after the GPU is done with the culling, we basically kick-off our CPU job which prepares for the actual lighting using a MEMEXPORT shader.

Typically, people use callbacks to do this, which is what Microsoft recommends, and not MEMEXPORT, which is not endorsed. In our case, our job system works in such a way that you can kick off jobs by writing a token at a specific address, which is what MEMEXPORT allows you to do.

# Don't Upset The GPU (1/)



- Constant Waterfall sucks!
  - This WILL kill performance
  - To prevent, use the aL register when iterating over lights [Pritchard10]
  - If set properly, ALU / lighting will run at 100% efficiency
- In C++ Code

```
int lightCounter[4] = { count, start, step, 0 };
pDevice->SetPixelShaderConstantI(0, lightCounter, 1);
```

*Advances in Real-Time Rendering in Games*

Here's a bit more about performance, and things you should do to not upset the GPU

First, constant waterfall sucks! This will kill performance, even with the most optimized shaders (approx. 33% perf, if not more)

To prevent this, you should use the aL register when iterating over lights. If this is set properly, your lighting code will run at maximum ALU efficiency. This is because if the GPU knows how loops will behave, it can schedule constant accesses in a orderly fashion (therefore waterfall-free)

In c++ code, this is basically how you want to set it.

First parameter defines the count of how many iterations, then the start value and then the step. Last parameter unused.

## Don't Upset The GPU (2/)



```
int tileLightCount : register(i0);
float4 lightParams[NUM_LIGHT_PARAMS] : register(c0);
[loop]
for (int iLight = 0; iLight < tileLightCount; iLight++)
{
    float4 params1 = lightParams[iLight + 0]; // mov r0 c0[0+aL]
    float4 params2 = lightParams[iLight + 1]; // mov r1 c0[1+aL]
    float4 params3 = lightParams[iLight + 2]; // mov r2 c0[2+aL]
    ...
}
```

start                          count\*step                          step

Advances in Real-Time Rendering in Games

I've highlighted the syntax required to make this happen on the HLSL side.  
This is similar to Cody's presentation from gamefest, but for our case.

This allows us to iterate over all of the lights in a tile, and access our lighting constants in a waterfall-free fashion.

# Don't Upset The GPU (3/)



- Use Dr.PIX, and check shader disassembly!

- These shaders are ALU bound

- Simplify your math, especially in the loops!
- Get rid of complicated non 1:1 instructions (e.g. smoothstep)
- Play with microcode:  $-\text{normalize}(v)$  is faster than  $\text{normalize}(-v)$
- Move code around to help with dual-issuing:

```
/* 14 */    mul r5.xyz, r4.yzx, r4.yzx  
this →      + mulsc r0.w, c254.y, r0.z
```

- Use shader predicates to help the compiler ([flatten], [branch], [isolate], [ifAny], [ifAll]), and tweak GPRs!

*Advances in Real-Time Rendering in Games*

Also, you need to use Dr.PIX and check shader disassembly. No need to be an expert here, but some basic microcode knowledge is a good idea when it comes to this level of optimization.

These shaders are definitely ALU bound, so make sure to simplify your math, especially in the loops.

(list rest of stuff)

# Don't Upset The GPU (4/)



- Use GPU freebies
  - Texture sampler scale/bias (\*2-1)
- Simplify / remove unneeded code via permutations
- Upload constants via the constant buffer pointers
- We use async pre-compiled command buffers (APCBs)
  - Keep them lean & mean (check contents in PIX)
  - For more info, check out Ivan's awesome presentation from Gamefest 2011 [Nevraev11]



*Advances in Real-Time Rendering in Games*

Use GPU freebies, such as the texture sampler scale bias

Also, simplify and remove unneeded code via compile-time permutations

One super important thing is to set your shader constants via constant buffer pointers. Basically, you allocate some memory where all your constants are, and tell the GPU: Hey, my constants are right there! `SetPixelShaderConstant` is slow, because it does some internal memcopies, which we definitely don't need here.

Finally, we use async pre-compiled command buffers, which also need to be kept lean and mean. You can easily see their contents in PIX. For more info about this, you should definitely check out Ivan Nevraev's awesome Gamefest presentation.

# Performance



Light Type (8 lights/tile, every tile)	Performance
Point	4.0 ms
Point (with Spec)	7.8 ms
Cone	5.1 ms
Cone (with Spec)	5.3 ms
Line	5.8 ms

- Classification: 1.35 ms (with resolves)

*Advances in Real-Time Rendering in Games*

Here is some of the performance numbers. In the case where we have a budget of 8 lights for every tile (some can be unique, some can be covering multiple tiles) here are some of the numbers.

This includes support for the full lighting model of BF3. This can be optimized even more, depending on how much you want to support. This can definitely be optimized and tailored for your BRDF.

# **Temporally-stable Screen-Space Ambient Occlusion**

*Advances in Real-Time Rendering in Games*



Let's now talk about our temporally-stable approach to screen-space ambient occlusion

# SSAO in Frostbite 2 (1/)



- SSAO for mid-range PC & consoles, HBAO for high-end PC
- Line sample [Loos10], with **linear depth** in a texture
- Linearize depth for better precision/distribution
  - $kZ = -\text{far} * \text{near} / (\text{far} - \text{near})$ ;
  - $kW = \text{far} / (\text{far} - \text{near})$
  - $\text{linearDepth} = kZ / (z - kW)$
- Sample linear depth texture with linear sampling
- Scale SSAO parameters over distance
- Final compositing with Hi-Stencil, reject sky pixels
- 4x4 random noise texture is sufficient, 1:1 (texel:pixel)

*Advances in Real-Time Rendering in Games*

SSAO in Frostbite 2 is for mid-range PCs and consoles. High-end PCs run Horizon-based AO, which looks awesome.

Rather than just handing out some of the code, since pretty much everyone here has toyed with AO, we're gonna focus on some of the tricks we do in order to get an alternative to HBAO, which works well for consoles and mid-range PCs.

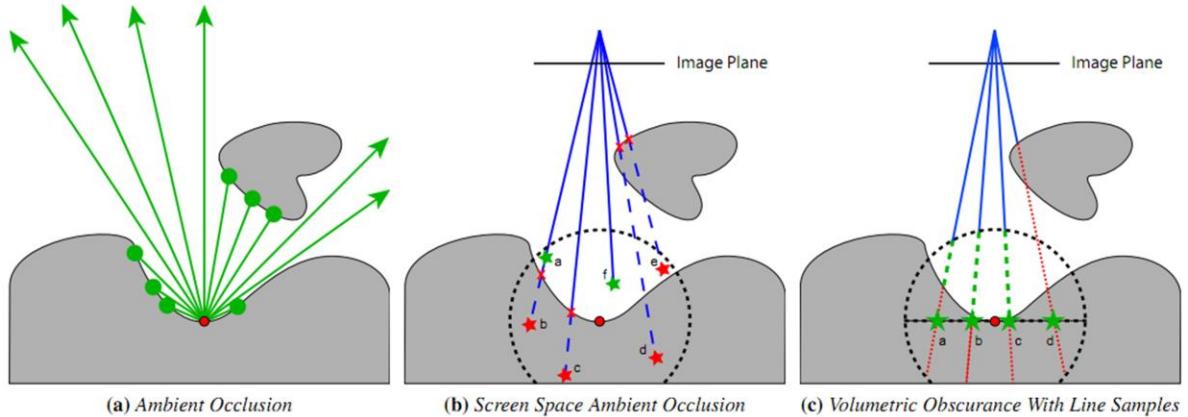
First, we use line sampling, and store linear depth in a texture. We basically linearize depth for better precision and distribution. A quick way to do this is with the following code, which only requires a single division ( $kZ$  and  $kW$  is computed on the CPU).

Once we have the linear depth, we also sample this texture in our AO pass using linear sampling, which is not something people usually since it's depth, but this goes hand-in-hand with line sampling and helps reducing jittering.

Another important thing is to scale AO parameters over distance, this will help with performance. Also, make sure to use HiStencil when compositing the final AO on screen, to make sure to reject sky pixels and such.

Finally, with line sampling, a 4x4 random noise texture is sufficient, no need to go bigger.

## SSAO in Frostbite 2 (2/)



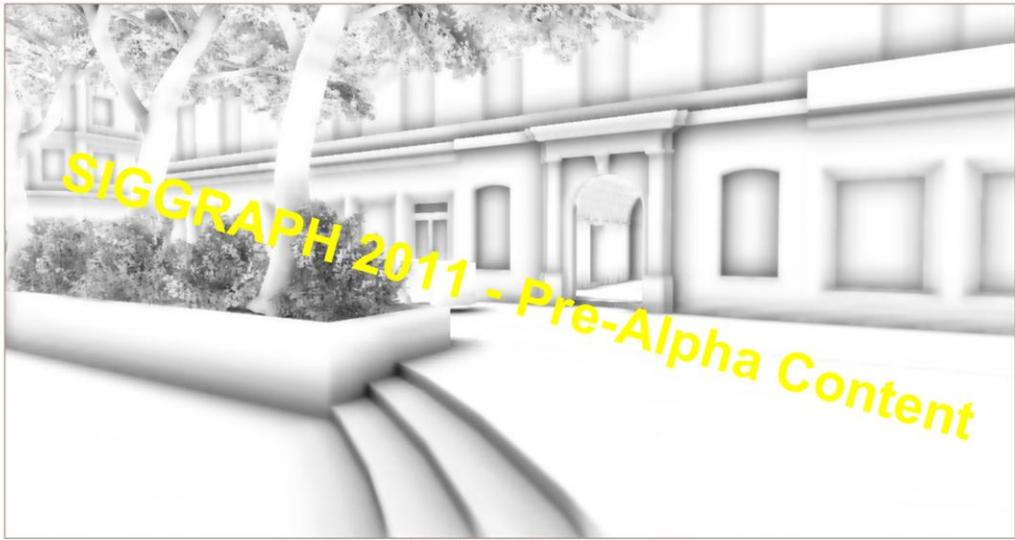
Line sampling, from *Volumetric Obscurrence* [Loos10]

In case you missed this paper by Loos and Sloan, here's an example of line sampling from *Volumetric Obscurrence*.

Line sampling is basically the analytic integral of the occupancy function times the depth extent of the sphere at each sample point on a disk.

Line sampling is also more efficient and stable due to the fact that all 2D samples will project to distinct points in the z-buffer, whereas the usual approach of random point sampling in 3D space, two point samples may project to the same location.

## HBAO in Frostbite 2



*Advances in Real-Time Rendering in Games*

If we compare HBAO to SSAO, here's an example with HBAO

# SSAO in Frostbite 2 (1/)



*Advances in Real-Time Rendering in Games*

And here's an example with SSAO. It definitely looks different (since it's in half-res), but this runs in 1/3 of the time, is pretty stable and there is almost no false-occlusion. It also responds well to foliage.

## Blurring the line (1/)



- Dynamic AO is done best with edge-preserving blur / bilateral filtering
- On consoles, we have really tight budgets
- Scenes are pretty action-packed, halos not too noticeable
- AO should be a subtle effect
- We need to find the fastest way to blur AO, and has to look soft! (e.g.: 9x9 Gaussian, with bilinear)

*Advances in Real-Time Rendering in Games*

When it comes to blurring, Dynamic AO is done best with bilateral / edge-preserving blurs

On consoles, we have tighter budgets. Our scenes are also pretty action-packed, so skipping on bilateral upsampling (which usually costs around 2.0-2.5ms) is not that big of a deal. We don't notice the halos too much. Then again, this works well when AO is a subtle effect.

To fit in budget, we need to find the fastest way to blur the AO, and it has to look soft (typically, a 9x9 separable bilinear Gaussian blur)

# Fast Grayscale Blur - 8 as 8888 (1/)



- Reduce the number of taps: aliasing the AO results from R8 as A8R8G8B8
- 1 horizontal tap == 4 taps (ARGB)
- Combine with bilinear sampling (vertical pass only)
- 9x9 Gaussian = 3 horizontal taps and 5 vertical taps
- On PS3: alias the memory directly
- On 360: Formats are different in memory, use resolve remap textures. See *FastUntile* XDK sample.

*Advances in Real-Time Rendering in Games*

Additionally, we can accelerate the blurring of the AO by aliasing the results, which is an 8-bit texture, as a ARGB8888 texture.

This means that 1 horizontal tap == 4 taps

We can also combine this trick with bilinear sampling, so reduce even more taps (but this only applies to the vertical pass) since bilinear sampling doesn't make sense, you can't bilinearly interpolate between "channels"

This means that a 9x9 Gaussian can now be done with 3 horizontal point sampled taps, and 5 vertical bilinear taps

On PS3, we can easily do this by directly aliasing the memory.

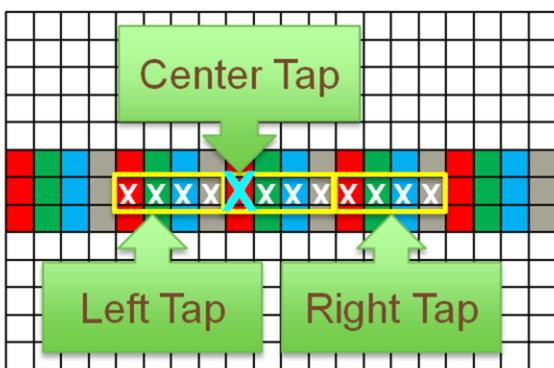
On 360, formats are different in memory, which means you have to use resolve-remap textures. See the *FastUntile* XDK on how to define and use those textures.

# Fast Grayscale Blur (1/)



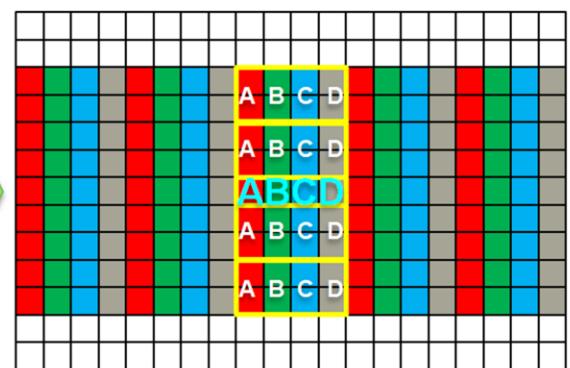
## Horizontal

9 “samples” → 3 point taps



## Vertical

9 “samples” → 5 bilinear taps



*Advances in Real-Time Rendering in Games*

If we visualize this blur, this is how it looks like for the horizontal pass, and then the vertical pass.

## Fast Grayscale Blur (2/)



For a 640x360 SSAO Buffer (720p / 2)

Technique	PlayStation 3	Xbox 360
9x9 Gaussian	0.5 ms	0.65 ms (0.52 ms + 0.132 ms resolve)
9x9 Gaussian (Bilinear, as R8)	0.40 ms	0.43 ms (0.3 ms + 0.132 ms resolve)
9x9 Gaussian (Bilinear, as A8R8G8B8)	<b>0.10 ms</b>	<b>0.18 ms</b> (0.143 ms + 0.034 ms resolve)

Average (total) AO performance (compute + blur + blot) :  
(360: 1.25-1.5 ms ; PS3: 1.5-2.0 ms)

*Advances in Real-Time Rendering in Games*

Here are the performance numbers for the consoles. As you can see, this definitely gives back performance, where a 640x360 grayscale blur only takes 0.1ms on PS3 and 0.18ms on 360!

# Thank You



- Christina Coffin
- Johan Andersson
- Ivan Nevraev
- Daniel Collin
- Khalid Khalkhouli
- Andrew Routledge
- Stephen Hill
- Aurelio Reis
- Alex Ferrier
- Fredrik Seehussen
- Alex Fry
- Natalya Tatarchuk
- Mohsin Hasan



**BLACK BOX** **DICE**

*Advances in Real-Time Rendering in Games*

There is obviously no “i” in team, and this wouldn’t have been possible without the help of these amazing people.

# Questions



John White – Bokeh, Z Cull Reverse Reload, Chroma Subsampling

jwhite@ea.com

@ZedCull

Colin Barré-Brisebois – Tile-based Deferred Shading, SSAO

cbbrisebois@ea.com

@ZigguratVertigo

*Advances in Real-Time Rendering in Games*

# References (1/)



ANDERSSON, J., "Parallel Graphics in Frostbite - Current & Future", Beyond Programmable Shading, SIGGRAPH 2009.

BAVOIL, L., SAINZ, M., and DIMITROV, R., "Image-space horizon-based ambient occlusion", SIGGRAPH 2008.

COFFIN, C., "SPU Based Deferred Shading for Battlefield 3 on Playstation 3", GDC 2011.

DEMERS, J., "Depth of Field : A Survey of Techniques", GPU Gems, Ch.23.

HUTCHINSON, N. et al., "Screen Space Classification for Efficient Deferred Shading", SIGGRAPH 2010.

JONES, M., Optimal CoC Calculation, Rendering @ EA internal mail, 2008.

*Advances in Real-Time Rendering in Games*

## References (2/)



- LOOS, B. J., and SLOAN, P-P., “Volumetric Obscurrence”, 2010.
- NEVRAEV, I., “Xbox 360 Precompiled Command Buffers”, Microsoft Gamefest London 2011.
- PRITCHARD, C., “Xbox 360 Shaders and Performance: How Not to Upset the GPU”, Microsoft Gamefest Seattle 2010.
- SOUSA, T., “Crysis Next Gen Effects”, GDC 2008.
- SWOBODA, M., “Deferred Lighting and Post-Processing on PLAYSTATION®3”, GDC 2009.
- KAWASE, M., “Frame Buffer Postprocessing Effects in DOUBLE-S.T.E.A.L (Wreckless)”, GDC 2003.

*Advances in Real-Time Rendering in Games*