
SIGGRAPH2010
The People Behind the Pixels
Advances in Real-Time Rendering Course Siggraph
2010, Los Angeles, CA

CryENGINE 3:
reaching the speed of light

Anton Kaplanyan
Lead researcher at Crytek

This talk is mostly about improvements in **texture compression** and **deferred lighting** on consoles.

Agenda

- Texture compression improvements
- Several minor improvements
- Deferred shading improvements

The talk consists of two major parts: texture compression and deferred lighting. Also several minor improvements will be mentioned.

TEXTURES

Agenda: Texture compression improvements

1. Color textures
   - Authoring precision
   - Best color space
   - Improvements to the DXT block compression
2. Normal map textures
   - Normals precision
   - Improvements to the 3Dc normal maps compression

There are two subparts of the texture compression part: compression of color/albedo textures and proposed improvements to normal maps compression.

Color textures

- What is color texture? Image? Albedo!
  - What color depth is enough for texture? 8 bits/channel?
  - Depends on lighting conditions, tone-mapping and display etc.
- **16-bits/channel** authoring is a **MANDATORY**
  - Major authoring tools are available in Photoshop in 16 bits / channel mode
- All manipulations mentioned below **don't make sense** with 8 b/channel source textures!

Advances in Real-Time Rendering Course Siggraph 2010, Los Angeles, CA

Nowadays all color textures are mostly used as albedo textures in the lighting pipeline, despites of good old times of Doom and Quake.

Despite there are still many engines with precomputed light maps and even complete prebaked lighting solutions (e.g. Rage from id software), majority of modern engines try hard to stick to the most robust dynamic lighting pipeline in order to provide the most convenient tools for artists, thus simplifying the game development process.

So the color textures for the latter engines mostly represent the albedo of the surface it describe.

Thus, the color depth and the color space of these textures become an important discussion topic in this talk.

The very first and important step is to change the texture authoring pipeline to the higher precision in order to be able to manipulate with source texture with no precision loss after it's been authored once. So we use a 16 bits/channel mode in Photoshop for texture authoring. This allows us to change the color space or histogram of the texture with no consequences both in the Photoshop and in our in-house texture processing tools.

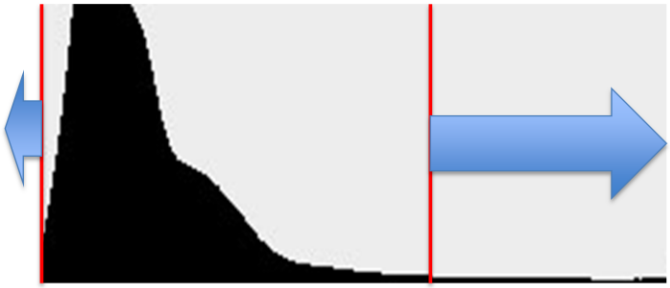
This mode has mostly the same authoring options as the usual 8 bits/channel sRGB authoring mode. That means that the switch is mostly transparent for artists.

The manipulations proposed on the following slides might make the quality **worse** if the source texure is 8 bits/channel! So it is very important to have a source texture

authored in 16bits/channel from the very beginning.

Histogram renormalization

- Normalize color range before compression
  - Rescale in shader: two more constants per texture
  - Or premultiply with material color on CPU



Advances in Real-Time Rendering Course Siggraph 2010, Los Angeles, CA

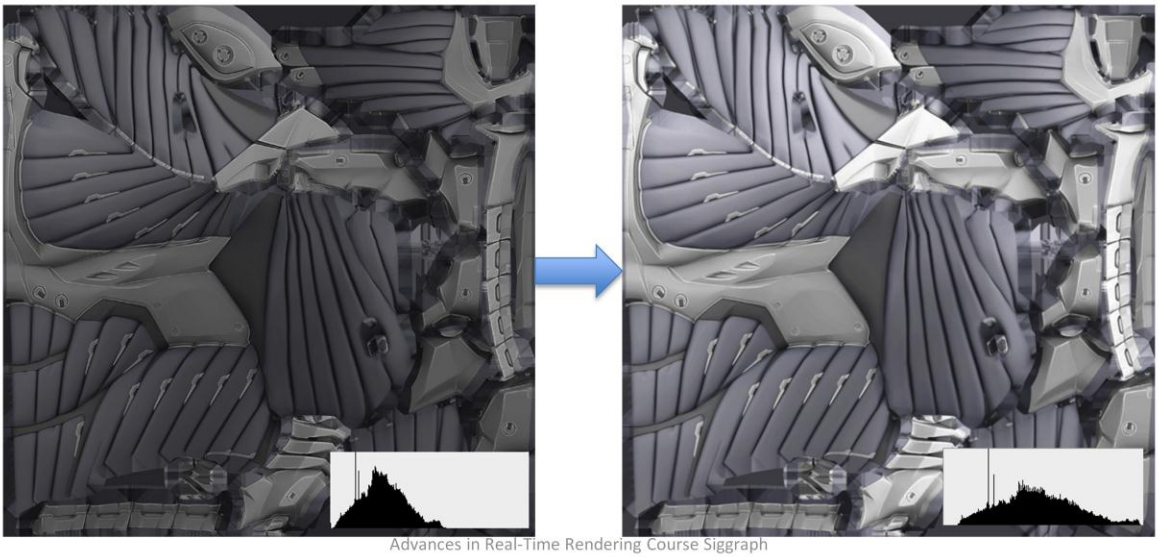
The first proposed manipulation is the color range renormalization.

Usually artists do not care about the quantization artifacts produced by block compression and/or low-precision quantization (that happens e.g. in DXT block compression). The most important point in authoring process is to achieve a similar looking texture in a usual game lighting. However dark textures might show very noticeable and disturbing artifacts under strong HDR lighting conditions.

Thus we decided to introduce a range renormalization for color textures behind the scenes. That significantly reduces the color banding and deviation introduced by 5:6:5 quantization and block compression of DXT format.

We store the original minimum and maximum limits into the resulting texture in order to be able to reconstruct the source texture created by the artist in the shader. That means that for each renormalized texture we set two additional shader constants in order to rescale the results of texture lookup.

Histogram renormalization

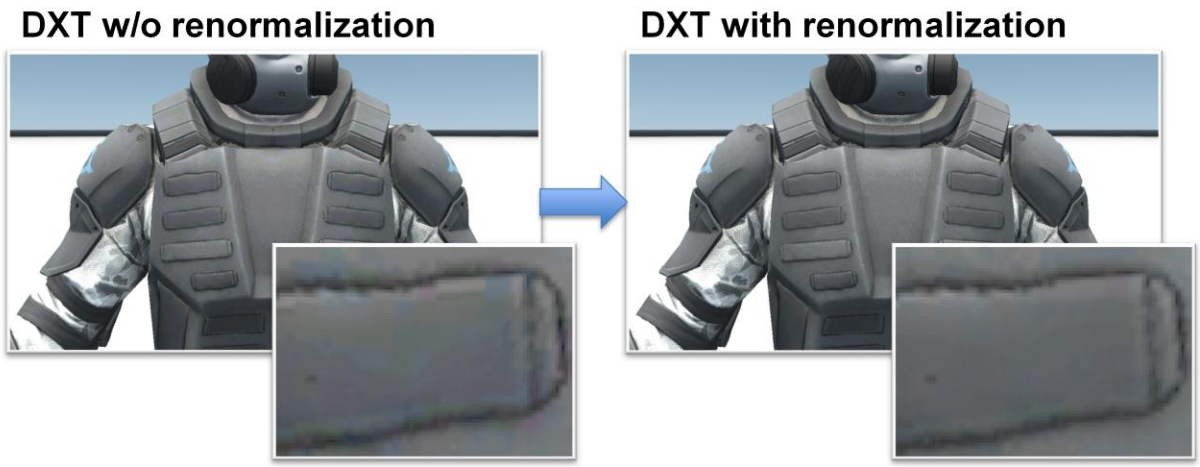


Advances in Real-Time Rendering Course Siggraph
2010, Los Angeles, CA

Here is the result of renormalization and the corresponding histograms of the original and the renormalized textures.

Note that the renormalization of **8b/ch source texture** would produce a quantization in the renormalized histogram, thus leading to a larger color distortion during 5:6:5 quantization stage of DXT compression.

Histogram renormalization example

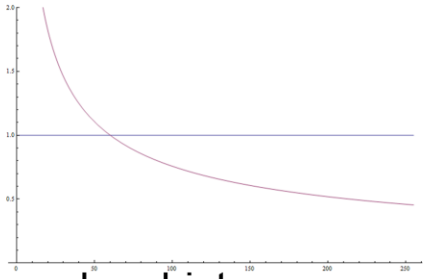


The left most image shows the results for simple DXT-compressed texture under strong lighting conditions. On the right most one there are results for the renormalized and then DXT-compressed texture. Note that the blocky color artifacts are significantly reduced.

The texture appearance is not changed due to reconstruction of original range in the shader.

Gamma vs linear space for color textures

- Two h/w color spaces for free: linear/gamma
  - Solution to the equation
    $$x_\gamma{}' = x'$$
  - Median (**linear space**):
    $$x = \frac{5}{11} = 0.(45) \approx \frac{\mathbf{116}}{255}$$
  - Choose the right color space based on histogram:
  - Rule of thumb: use linear if >75% of pixels are above the median



Advances in Real-Time Rendering Course Siggraph 2010, Los Angeles, CA

The most of texture authoring software work in sRGB space. This is the native space of the display and it is perfectly represented in 8 bits.

Also there is generally accepted opinion that the color textures should be stored in sRGB color space too.

The interesting fact that there are two color spaces supported natively for free in the modern GPU. That means we can easily jump from one to another with no performance impact.

That fact arises a very interesting question about the color density in each color space. The density of the color space shows the quantization interval for the considered color space for a particular color range. In order to answer this question we can solve the equation of densities deduced from comparing the derivatives of each color space transformation in linear space.

The color transformation for linear space is obviously an identity transformation. The transformation from gamma space is a power function.

By solving this equation we can acquire the resulting density median for these color spaces. It shows the point of equal density of both linear and gamma space. This point is in the **linear space**. This means that the linear space has better precision above the value 116/255. And the gamma space is more precise below it.

So, we use the more appropriate color space for each particular texture judging by the texture's histogram.

The **rule of thumb** that works for us is to switch to linear color space each time the number of texels above the median point in the texture is more than **75%**.

Gamma vs linear space on Xbox 360

- Xbox 360 has a **picewise** gamma curve
  - Median (**linear space**):

$$x \approx \frac{90}{255}$$

  - sRGB textures stored as **8 b/ch** in **linear space** in texture cache
    - Otherwise have to sample it _AS16 => cache pollution
- It is highly recommended to use as many as possible linear textures on Xbox 360

Advances in Real-Time Rendering Course Siggraph 2010, Los Angeles, CA

It is also generally correct for Xbox 360. However we need to take into account a different gamma curve used by Xbox's GPU. It is a piecewise linear approximation to the power function of gamma transformation.

This shifts the choice of color space even more towards the **linear space**. The median is lower for piecewise linear gamma curve: 90/255 (in linear space). That means it is **25/255** in gamma space! So, keeping the rule of thumb the same (if >75% texels are above the median) leads us to the conclusion that the majority of textures could be beneficially stored in linear space for this platform.

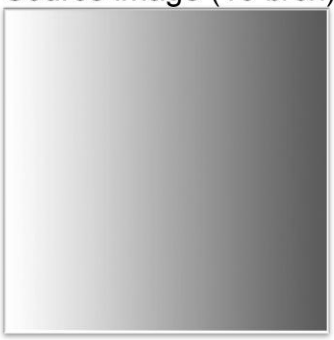
On the other hand it solves another very important issue of the Xbox 360's GPU related to gamma-space textures automatically. Every time the texture stored in gamma space is fetched in the shader, the GPU does the digamma and then stores the result back into 8 bits into the texture cache. This is a default behavior and it definitely introduces a great precision loss. In order to avoid the unwanted quantization, one could customize the texture format and specify the desired filtering precision by adding _AS16 suffix to the format (see the Xbox 360 documentation for more information). But that would definitely lead to much higher texture cache pollution and a consequent performance degradation.

However storing the majority of textures in linear space for this platform definitely solves this issue and further improves the color precision.

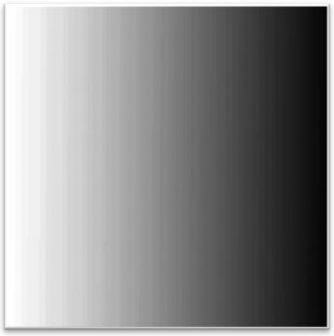
So as a conclusion we'd like to highly recommend using linear space textures for Xbox 360.

Gamma / linear space example

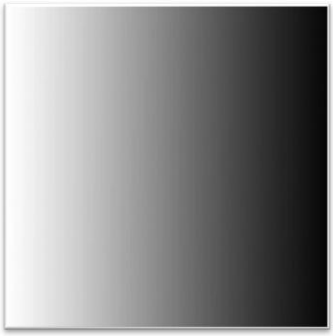
Source image (16 b/ch)



Gamma (contrasted)



Linear (contrasted)



Advances in Real-Time Rendering Course Siggraph
2010, Los Angeles, CA

This is an example bright gradient texture (all colors are above the median) stored in 16 b/ch shown on the left most image. The middle image shows the same texture stored in gamma space and exaggerated in order to show the color banding on the usual display. The right most image shows the original texture stored in linear space with the same exaggeration.

It clearly shows the benefit of storing the bright textures in linear space. Note that this exaggeration could easily happen under some strong and/or contrast lighting conditions and using contrast tone mapping operator, like a filmic tone mapping.

Normal maps precision

- Artists used to store normal maps into 8b/ch texture
  - Normals are quantized from the very beginning!
- Changed the pipeline to ALWAYS export **16b/channel normal maps**!
  - Modify your tools to export that by default
  - Transparent for artists

The rest slides of the texture compression part of the talk are devoted to the compression of normal maps.

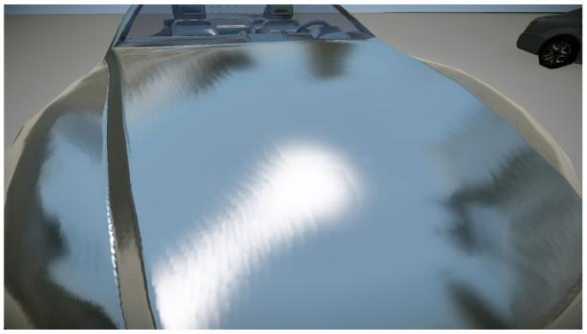
Historically, normal maps are being represented as a color maps. However the recent research in this area showed that the precision required to represent normals is much higher and heavily depends on the lighting conditions and the material properties.

As normal maps are usually a result of the matching process of low-poly and high-poly model, it is always possible to extract the normals of a very high precision.

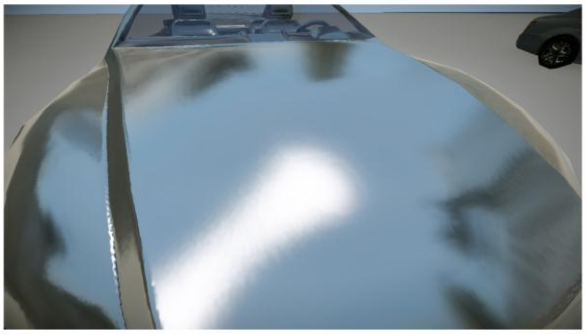
Thus we recommend to modify the exporting tool you use in order to produce a 16bits/channel uncompressed normal map in order to be compressed further. This is a very important step, as the 8bits/channel normal maps introduce a huge error from the very beginning.

16-bits normal maps example

3Dc from 8-bits/channel source



3Dc from 16-bits/channel source



Advances in Real-Time Rendering Course Siggraph
2010, Los Angeles, CA

This is an example of quantization artifacts introduced by storing source normal maps in 8b/ch format. As you can see, even the 8b/texel 3Dc compression gives much better results with 16b/ch, as it introduces artifacts only at the compression stage. However with 8b/ch normal maps the quantization error is accumulated from both 8b/ch quantization and the quantization introduced by the 3Dc compression.

3Dc encoder improvements

- 3Dc is **much better than ARGB8**
  - Interpolants are produced at 16-bits precision on majority of GPUs!
- Usual 3Dc encoder: compress x and y **independently** as two **alpha** channels – **doesn't** treat x-y as a normal!
- We propose to improve 3Dc encoder:
  - Treat two alpha blocks as a whole x-y normal instead
  - Compute the error for normal instead of "color difference":

$$\Delta N = ArcCos\left(\frac{(N_c \cdot N)}{\|N_c\|\|N\|}\right)$$

Also we propose an improvement to the 3Dc encoder. The usual 3Dc encoder encodes the normal map into a block-compressed two-channeled format. Each block consists of two alpha channel blocks of DXT5 format leading to 8bit/texel compression.

However the precision of data represented with 3Dc compressed block can exceed the precision of the ARGB8 format. The reason is that the majority of GPUs compute the interpolated values between two anchors in higher precision rather than 8 bits. E.g. The precision of interpolated values on Xbox 360 is 16 bits. This leads us to the point that the source uncompressed normal map should be stored definitely with the precision higher than 8b/ch! However this change should be also reflected into the 3Dc encoder.

Besides that, all existing 3Dc encoders perform the block compression of each of two channels separately, treating each block as a usual alpha block of DXT5 texture.

However while compressing normals, the error should be computed differently from alpha compression. Besides that the correlation between x and y channel of 3Dc block should be reflected in that error.

We propose to use a common error to measure directional deviations: $\Delta N = ArcCos\left(\frac{(N_c \cdot N)}{\|N_c\|\|N\|}\right)$. So, after two candidate pairs of anchors for a 3Dc block are chosen, the normals are decompressed, the z component is reconstructed and the error between the source and the compressed normals is measured.

3Dc encoder improvements, cont'd

- One 1024x1024 texture is compressed in **~3 hours** with CUDA on Fermi!
  - Brute-force exhaustive search
  - Too slow for production
- Notice: solution is close to common 3Dc encoder results
- Adaptive approach: compress as 2 alpha blocks, measure error for normals. If the error is higher than threshold, run high-quality encoder

Advances in Real-Time Rendering Course Siggraph 2010, Los Angeles, CA

However using this error we need to excess all the possible pairs of (x,y) anchors. That correlation complicates and slows down the compression process. The performance of 3Dc compression becomes production-unfriendly.

But as it was noticed during the development, the solution found by a common 3Dc encoder during the compression of separate alpha-blocks, is very close to the solution with the best error. Besides that, the "common solution" is the best solution in many situations.

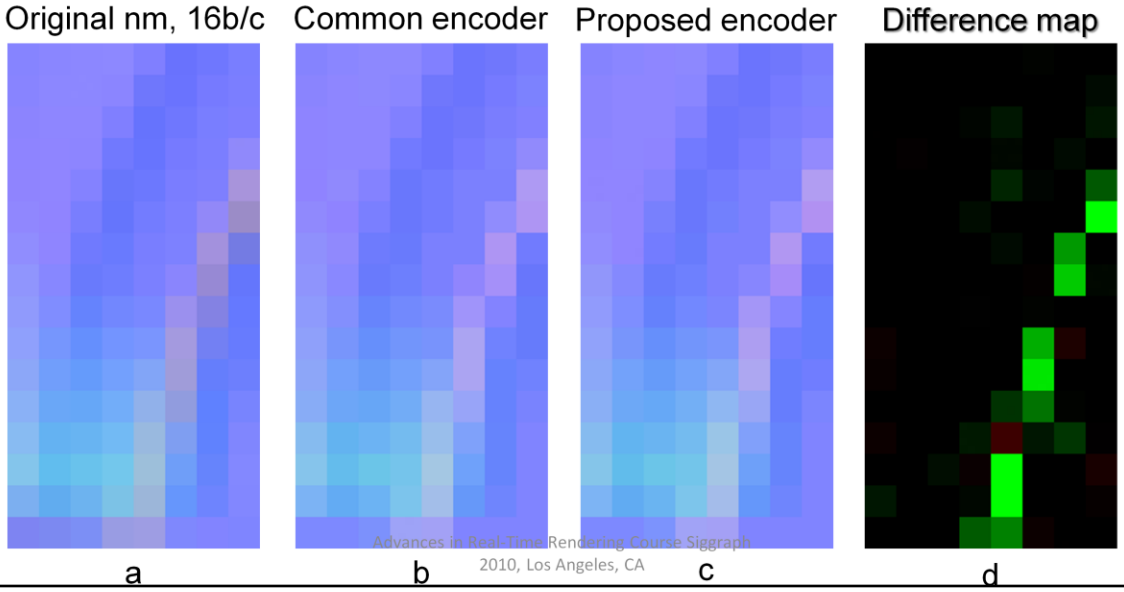
So, we propose an adaptive approach for the 3Dc encoding:

1. Compress the block with a usual 3Dc compression approach
2. Measure the directional deviation with the proposed error
3. If the error is higher than some threshold, initiate the improved compression stage.

Note that we use an exhaustive search only around the "common solution". We use [v-4/255;v+4/255] interval for the exhaustive search, which is proved to be enough.

Thus the performance overhead introduced is minimal.

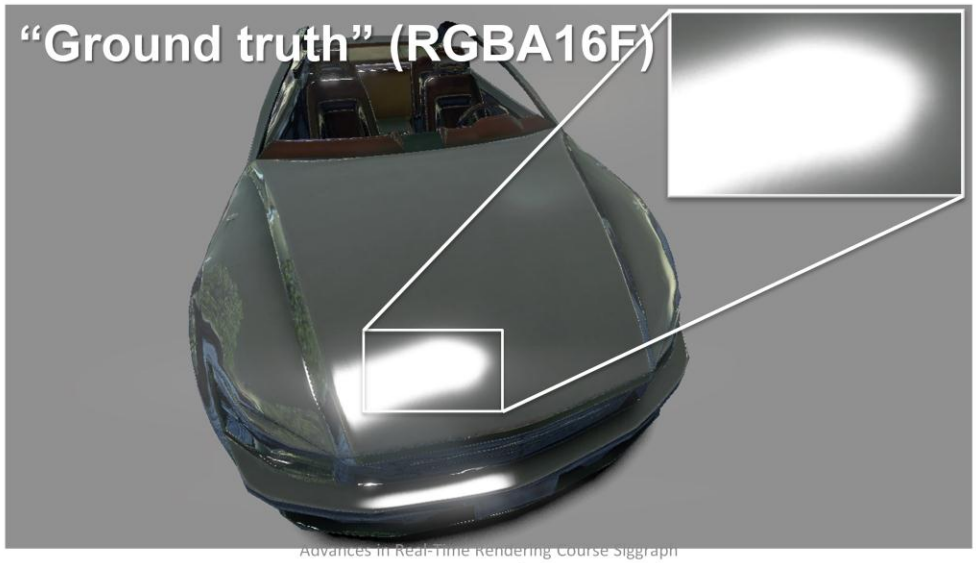
3Dc improvement example



Here is a comparison of the 3Dc encoder (image b) and the improved 3Dc encoder (image c). As the differences are hardly visible in a narrow color space of display, we also provide a difference map (d). This map shows a pixel in green if our method has smaller error compared to usual 3Dc encoder. The intensity is the difference in errors, amplified by 5. Also it shows a pixel in red if our method provides a worse error for it. As you can see our method is mostly surpasses the usual 3Dc encoder.

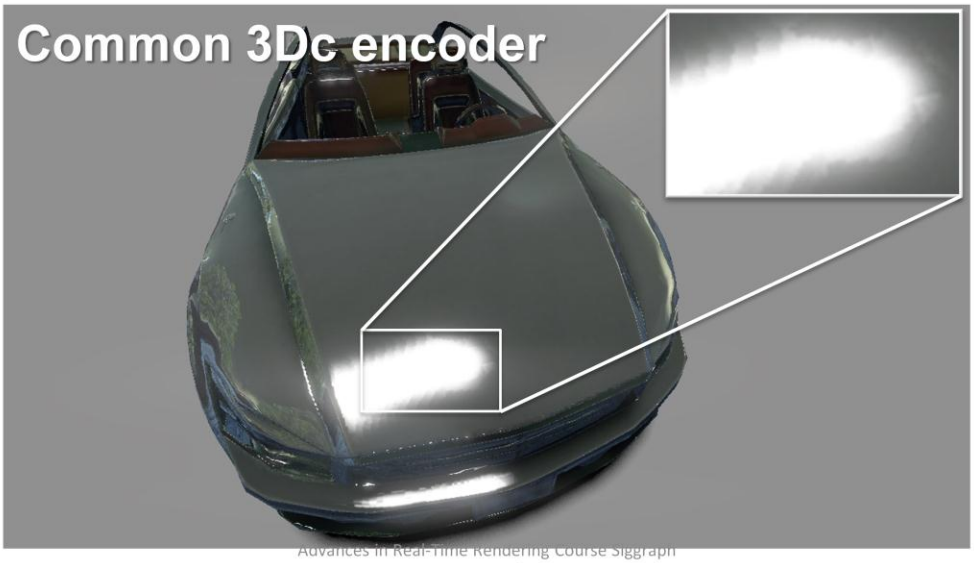
However this is a pure theoretical comparison. Let's see how does it look in practice.

3Dc improvement example



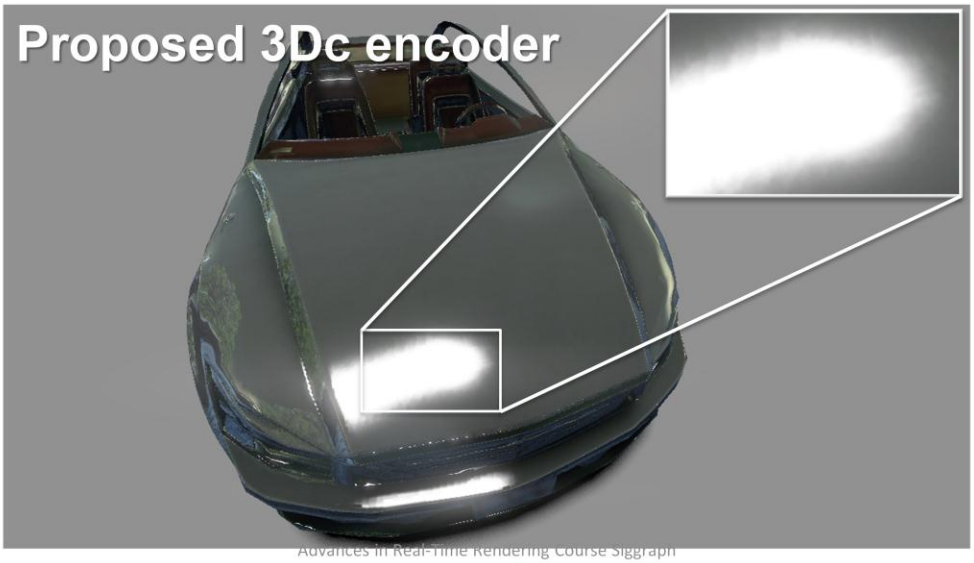
This is an image of the car rendered with the uncompressed normal map, stored in RGBA16F format. As we mentioned before, the ARGB8 format is not sufficient for the comparison with 3Dc format, as it introduces more quantization than 3Dc.

3Dc improvement example



This is the image produced using a usual 3Dc encoder. You see the quantization on the hood leading to the banding of specular reflections.

3Dc improvement example



And this image is rendered with the **improved 3Dc encoder**. As you can see, the specular reflections are much **less banded**.

Note that the format remains the same (3Dc)! The improvements are done only on the **encoder's side**.

DIFFERENT IMPROVEMENTS

Occlusion culling

- Use software z-buffer (aka coverage buffer)
  - Downscale previous frame's z buffer on consoles
    - Use conservative occlusion to avoid false culling
  - Create mips and use hierarchical occlusion culling
    - Similar to Zcull and Hi-Z techniques
    - Use AABBs and OOBBs to test for occlusion
  - On PC: place occluders manually and rasterize on CPU
    - CPU↔GPU latency makes z buffer useless for culling

Advances in Real-Time Rendering Course Siggraph 2010, Los Angeles, CA

We have a low-resolution software depth buffer since CryENGINE 2 and Crysis 1.

This significantly reduces both CPU and GPU workload, as we skip the complete processing and preparation for rendering for culled objects.

However a complete software culling is less efficient, as we are able to render only some very small and sparse part of the real scene on the CPU.

However we improved this approach on consoles.

Because of a thin GPU-CPU intercommunication layer (fast memory busses, no API limitations, no virtualization), we are able to retrieve the z buffer from GPU with only one frame delay.

We downscale the z buffer on GPU using a max() filter in order to preserve a conservative visibility detection. On Xbox 360 we untile the downscaled depth buffer on CPU.

Afterwards we construct mip levels with a minimum and maximum filter in order to construct a hierarchical representation of the scene depth. This is a rather important acceleration structure for culling. A similar algorithm is used in a vast majority of modern GPUs.

Then during the visibility detection we project an AABB or OOBB of the object into the screen space and detect the necessary mip level.

- If the object is completely occluded by the minimum level of this mip, we cull this

object out.

- If the object is in front of the maximum bound of the mip, it's definitely not occluded.
- If the object's depth bounds intersect the minimum-maximum interval of this mip, we intersect it with the depth information from the higher mip in order to detect a precise visibility.

However on PC we use a complete software solution with some hand-placed low-poly proxies made by level designers. This is due to the high latency in the GPU-CPU intercommunication, which leads us to a 3-frames delay. This is unacceptable for culling.

SSAO improvements

- Encode depth as 2 channel 16-bits value [0;1]
  - Linear detph as a rational: depth=x+y/255
- Compute SSAO in half screen resolution
  - Render SSAO into the same RT (another channel)
  - Bilateral blur fetches SSAO and depth at once
- Volumetric Obscurrance [LS10] with **4**(!) samples
- Temporal accumulation with simple reprojection
- Total performance: **1**ms on X360, **1.2**ms on PS3

We use downscaled z-buffer.

During downscaling, we encode the linear depth value into two channels of the ARGB8 texture.

The rest channels we use for SSAO computation of current frame and the temporally accumulated results.

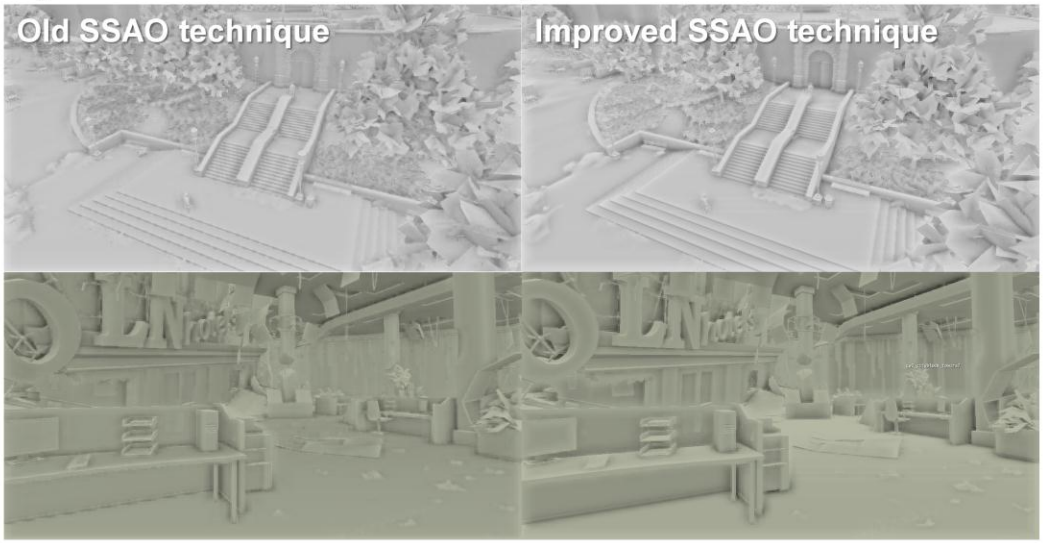
We compute the SSAO in half resolution of the screen. Then we do a bilateral upscaling onto the screen.

We use Volumetric Obscurrance[LS10], which allowed us to lower down the number of samples to 4!

Of course this approach is supplemented by temporal accumulation in order to provide more samples over time. However we don't use sophisticated cache rejection schemes, rather a simple reprojection from previous frame.

Thus the performance achieved is around 1 ms on both consoles.

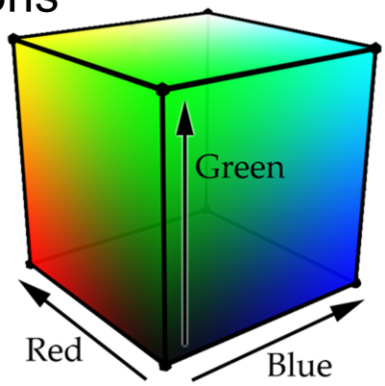
Improvements examples on consoles



Advances in Real-Time Rendering Course Siggraph
2010, Los Angeles, CA

Color grading

- Bake all global color transformations into 3D LUT [SELAN07]
  - 16x16x16 LUT proved to be enough
- Consoles: use h/w 3D texture
  - Color correction pass is one lookup
    - newColor = tex3D(LUT, oldColor)



The color transformations are a very important tools for artists.

However creating the whole infrastructure of sophisticated professional filters and other tools is usually an unaffordable task for the game engine.

The vast majority of image-wide filters (like color correction, contrast, brightness, levels, selective colors etc.) can be represented as a **mapping** of an rgb cube into another rgb cube.

This mapping can be represented as a **3D Look-Up Table** [Selan07].

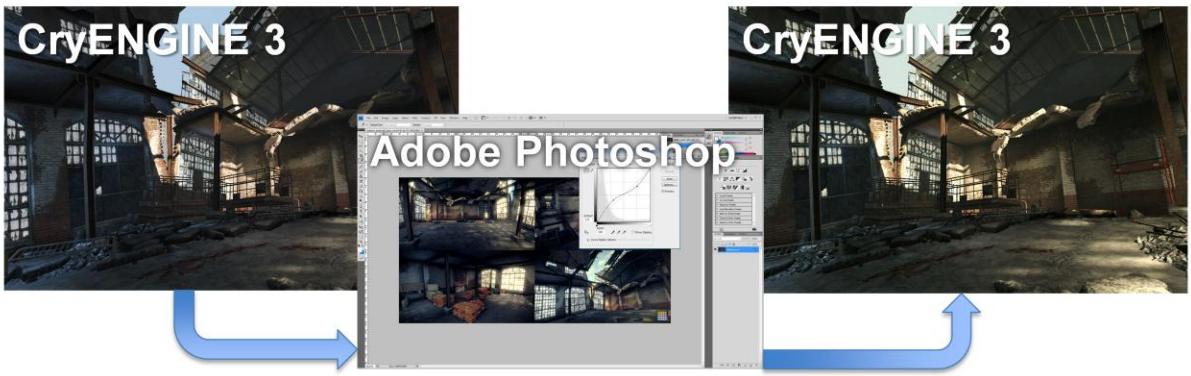
Taking into account the color-space consistency of all filters, the size of this LUT can be very small.

After doing some experiments, we figured out that the size of **16x16x16** is more than enough for the majority of color transformations.

Also it is possible to use the GPU-aided 3D textures, which speeds up the color correction on consoles, making it only **one texture look up**.

Color grading

- Use Adobe Photoshop as a color correction tool
- Read transformed color LUT from Photoshop



Another very important part of the color grading is to provide a convenient **workflow**.

For that purposes we unwrap the **16x16x16 identity LUT** into 16 slices and bake it into the **source image** (which is usually a game screenshot).

This is a small **chart**, that consists of 16 slices, 16x16 each. This chart initially maps each rgb color to itself, which is an identity transformation for the source screenshot.

After artist has finished working on the color correction of the image/screenshot, we **read back** the chart and **save it as a texture LUT**.

Afterwards we **load** this LUT into the engine (**2D** or **3D** texture, depending on platform and API limitations) and perform the **full-screen color transformation** with it.

This was an artist has completely **identical results** to what he/she wanted to achieve in the image authoring tool.

Color chart example for Photoshop

This is an example of the image, which consists of several screenshots of the same location and one small color chart in the corner.

This color chart serves as a probe for all color transformation and post processes an artist applies onto the image.

DEFERRED PIPELINE

Advances in Real-Time Rendering Course Siggraph
2010, Los Angeles, CA

This part of the talk is devoted to the deferred pipeline in CryENGINE 3.

Performance challenges will be discussed as well as some new solutions and techniques.

Why deferred lighting?

Advances in Real-Time Rendering Course Siggraph
2010, Los Angeles, CA

This is an example screenshot of a real level in Crysis 2.

Why deferred lighting?

Advances in Real-Time Rendering Course Siggraph
2010, Los Angeles, CA

This is the **diffuse** lighting buffer corresponding to the screenshot.

Why deferred lighting?

Advances in Real-Time Rendering Course Siggraph
2010, Los Angeles, CA

This is the **specular** lighting buffer corresponding to the screenshot.

Introduction

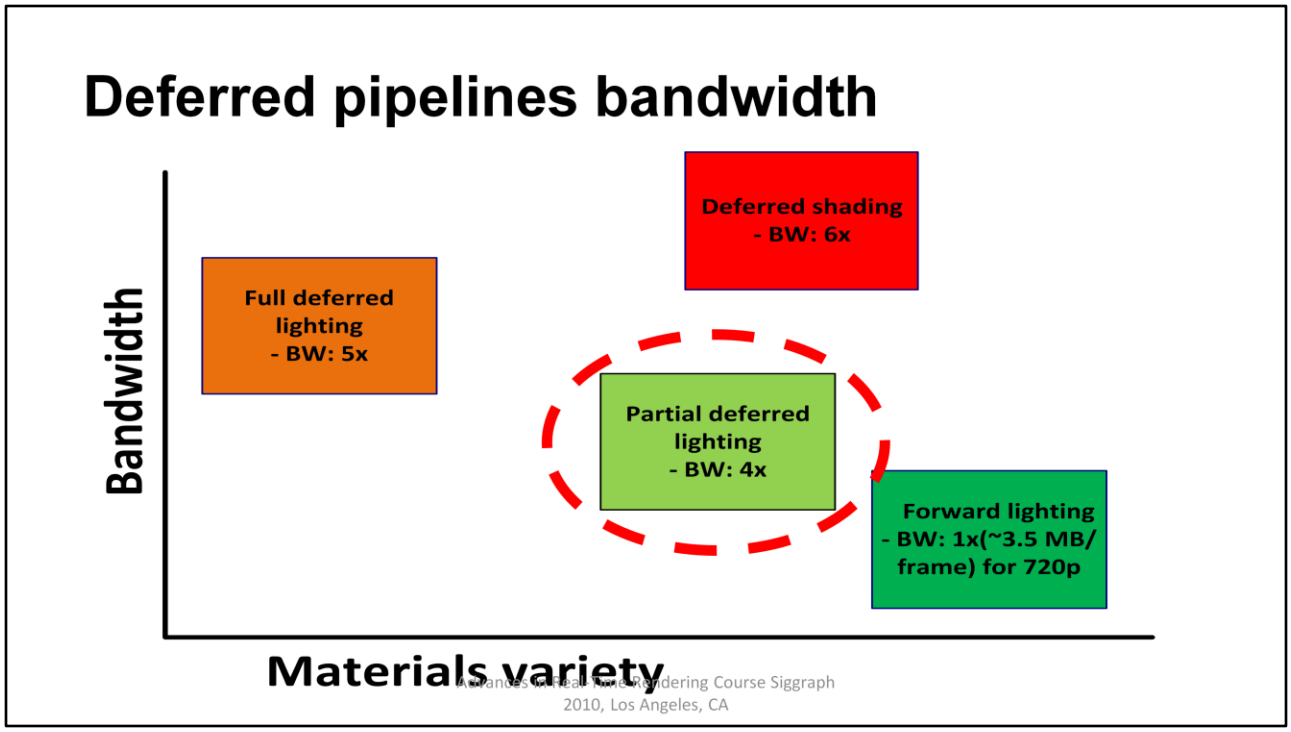
- Good decomposition of lighting
  - No lighting-geometry interdependency
- Cons:
  - **Higher memory and bandwidth requirements**

Advances in Real-Time Rendering Course Siggraph
2010, Los Angeles, CA

The deferred approach is widely known for its good property of decomposing the lighting depth from the geometry granularity.

That means that the geometry is rendered before the actual lighting is done.

However, we have to store screen-space “fragments” for the deferred lighting into a Geometry buffer (G-Buffer). This leads to much **higher bandwidth**.



This is a schematic chart showing the trade-off between the number of attributes stored per fragment versus the shading flexibility and materials variety.

The forward shading is here for reference, as it looks like a perfect solution (the highest materials variety vs the lowest bandwidth), however the forward lighting does not decouple the complexity. That is inappropriate for modern games with very complicated layered lighting, as shown before.

Thus there are three widely-known types of deferred rendering pipeline:

- Classic deferred shading (S.T.A.L.K.E.R. etc) [GPU Gems 2]. G-Buffer layout: depth, normal, glossiness, albedo, occlusion and material index – 112-144 **bits/pixel**
- Full deferred lighting (aka Light Prepass) [Engel09]. G-Buffer layout: depth, normal, glossiness and albedo – **96 bits/pixel** (best case)
- Partial deferred lighting [Mittring09]. G-Buffer layout: depth, normal and glossiness – **64 bits/pixel**.

However, shortening the layout comes with cost: material variety becomes more and more limited.

Major issues of deferred pipeline

- No anti-aliasing
  - Existing multi-sampling techniques are too heavy for deferred pipeline
  - Post-process antialiasing doesn't remove aliasing completely
    - Need to super-sample in most cases
- Limited materials variations
  - No anisotropic materials
- Transparent objects are not supported

Advances in Real-Time Rendering Course Siggraph 2010, Los Angeles, CA

There are also some **fundamental limitations** of the deferred pipeline on GPUs:

1. The **anti-aliasing** becomes a super-sampling: need to store the G-Buffer information **per sample**. **Dramatically** increases the bandwidth.
2. **Arbitrary materials** increase the amount of stored information **per pixel** significantly. That's why the materials variety is usually very **limited**.
3. The **transparent objects** should be rendered into a "**deep G-Buffer**" in order to store **all transparent layers** for deferred shading/lighting. This is a similar problem to anti-aliasing.

Lighting layers of CryENGINE 3

- Indirect lighting
  - Ambient term
  - Tagged ambient areas
  - Local cubemaps
  - Local deferred lights
  - Diffuse Indirect Lighting from LPVs
  - SSAO
- Direct lighting
  - All direct light sources, with and without shadows

Advances in Real-Time Rendering Course Siggraph 2010, Los Angeles, CA

In CryENGINE 3 we have the following layers of lighting:

- Indirect lighting
  - Ambient term
  - Tagged ambient areas
  - Local cubemaps
  - Local deferred lights
  - Diffuse Indirect Lighting from LPVs
  - SSAO
- Direct lighting

- All direct light sources, with and without shadows

These layers are applied in the order enlisted above. However some of these layers are optional depending on the area and lighting artist’s setup.

G-Buffer. The smaller the better!

- Minimal G-Buffer layout: 64 bits / pixel
  - **RT0**: Depth 24bpp + Stencil 8bpp
  - **RT1**: Normals 24 bpp + Glossiness 8bpp
- Stencil to mark objects in lighting groups
  - Portals / indoors
  - Custom environment reflections
  - Different ambient and indirect lighting

Advances in Real-Time Rendering Course Siggraph
2010, Los Angeles, CA

As it was described before, we use a **minimal G-Buffer layout** [Mittring09].

The layout consists of the **depth** (24bits), **stencil** (8bits), **normal** (24bits) and the **glossiness** (8bits).

We use stencil to mark objects that lay in **different areas**.

E.g. we mark objects that should receive different ambient in different indoor rooms with different stencil masks.

G-Buffer. The smaller the better, Cont'd

- Glossiness is non-deferrable
  - Required at lighting accumulation pass
  - Specular is non-accumulative otherwise
- Problems of this G-Buffer layout:
  - Only Phong BRDF (normal + glossiness)
    - **No aniso materials**
  - Normals at 24bpp are too quantized
    - Lighting is banded / of low quality

Advances in Real-Time Rendering Course Siggraph 2010, Los Angeles, CA

The very important lighting parameter is the **glossiness**.

On one hand, it represents the specular power. Without this value the specular contribution from different lights **cannot be accumulated** together.

But on the other hand the glossiness together with the normal define a **complete lobe of the Phong BRDF**. Note that without the glossiness, the **BRDF is undefined**, thus the lighting **cannot be performed**.

However the consequent problem with this G-Buffer layout is the small variations of materials, **limited by the Phong shading**.

Another very important problem is the representation of normals in the G-Buffer. We store normalized world-space normals into RGB channels of the RGBA8 render target.

With this representation, the normals are stored with an **insufficient precision**.

STORING NORMALS IN G-BUFFER

Advances in Real-Time Rendering Course Siggraph
2010, Los Angeles, CA

This part it devoted to the new technique for storing normals in **24-bits** G-Buffer efficiently.

Normals precision for shading

- Normals at 24bpp are too quantized, lighting is of a low quality
- 24 bpp should be enough. What do we do wrong? We store normalized normals!
- Cube is 256x256x256 cells = 16777216 values
- We use only cells on unit sphere in this cube:
  - **~289880** cells out of **16777216**, which is ~ **1.73 %** ! !

Advances in Real-Time Rendering Course Siggraph
2010, Los Angeles, CA

As it described before, with a current representation, the normals are stored with an **insufficient precision**.

That leads to banding artifacts with **all sorts of normals-related shading**, like diffuse lighting, environment reflection, specular lighting etc.

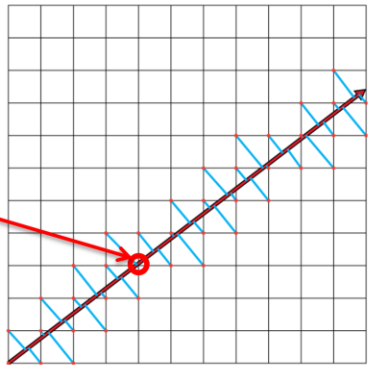
However talking from the information theory perspective, 24bpp should be enough to represent a direction (essentially a point on the sphere) very precisely.

Seems like we're doing something wrong.

If we count how many values we use out of these 24 bits, it turned out that we use only values on the unit sphere out of a 3d grid 256x256x256. That leads us to the usage of around **2%**!

Normals precision for shading, part III

- We have a cube of 256^3 values!
- Best fit: find the quantized value with the minimal error for a ray
  - Not a real-time task!
    - Constrained optimization in 3DDDA
- Bake it into a cubemap of results
  - Cubemap should be huge enough (obviously > 256x256)



Advances in Real-Time Rendering Course Siggraph 2010, Los Angeles, CA

We propose the new method to utilize **all values** inside this unit cube.

Having the original normal (=direction), we take **each cell** of the cube the normal ray **intersects**.

We compute the **quantization error** for every such a cell.

The quantization error tells us what would be the **deviation** of the center of the cell compared to the original normal direction.

That shows the error if we store this direction **into that cell**.

Using that, we find the cell with **minimal deviation** from the normal's ray on the ray's way.

Thus we find the **best cell** for some particular direction.

However finding this minimum is **non-trivial brute-force search task**, which **cannot be done in real-time** for each normal we want to store into G-Buffer.

Thus we decided to **prebake** the result of the search into a **huge cubemap** of directions. Each texel of the cubemap stores **the distance to the best cell** for the normal with this direction.

The cubemap's face should be **larger** than 256x256 in order to provide fine-grained solutions for directions of normals.

The **larger** the cubemap the **more precise** the normals representation. However even a cubemap of **2k x 2k** is already **problematic** for performance and memory on

consoles.

Normals precision for shading, part III

- Extract the most meaningful and unique part of this symmetric cubemap
- Save into 2D texture
- Look it up during G-Buffer generation
- Scale the normal
- Output the adjusted normal into G-Buffer
- See appendix A for more implementation details

Advances in Real-Time Rendering Course Siggraph
2010, Los Angeles, CA

However this cubemap has a lot of **symmetry** inside. The cubemap's symmetry comes from the inherited symmetry of the task itself. Indeed, the best cell solution should be the same in **all 8 octants**. Also inside the octant there is a **diagonal symmetry** for the same reason.

Using that fact, we extracted the unique part of the cubemap and saved it into a small **512x512** 2D texture. The emulation of cubemap lookup is done in the pixel shader. The detailed shader code and the texture itself you can find in the **appendix** of these slides.

The algorithm of outputting the normal into the G-Buffer is as follows:

- Prepare the texture coordinates for 2D lookup and look up the distance to the best cell from the precomputed 2D texture
- Scale the normalized normal by this value in order to fit it into the precomputed best cell
- Output the scaled normal into the G-Buffer.

Best fit for normals

- Supports alpha blending
  - Best fit gets broken though. Usually not an issue
- Reconstruction is just a normalization!
  - Which is usually done anyway
- Can be applied to some selective smooth objects
  - E.g. disable for objects with detail bump
- Don't forget to create mip-maps for results texture!

Advances in Real-Time Rendering Course Siggraph 2010, Los Angeles, CA

This method has a lot of advantages.

- It **supports alpha-blending**, as it becomes a blending between two unnormalized vectors, which is still linear and correct. However in case of alpha blending, we lose the best fit solution in the end. But it is not that important for the production, as the blending is mostly used for decals, smooth transitions and detail normal mapping, where the smoothness of normals is not an issue.
- The reconstruction is extremely cheap (even for free): it is only **one normalization**, which is **done anyway** in most of engines because of unnormalized results provided by alpha blending.
- It is **backwards-compatible** with storing normalized normals solution. That means that the best fit can be applied **selectively per object**, based on the reflectance, smoothness, presence of detail normal maps etc.

Also a small note on the performance: the **mip maps** for the 2D texture **are essential**, otherwise the texture cache thrashing becomes quickly apparent.

Storage techniques breakdown

1. **Normalized normals:**
   - **~289880** cells out of **16777216**, which is ~ **1.73 %**
2. **Divided by maximum component:**
   - **~390152** cells out of **16777216**, which is ~ **2.33 %**
3. **Proposed method (best fit):**
   - **~16482364** cells out of **16777216**, which is ~ **98.2 %**
     - Two orders of magnitude more

Advances in Real-Time Rendering Course Siggraph
2010, Los Angeles, CA

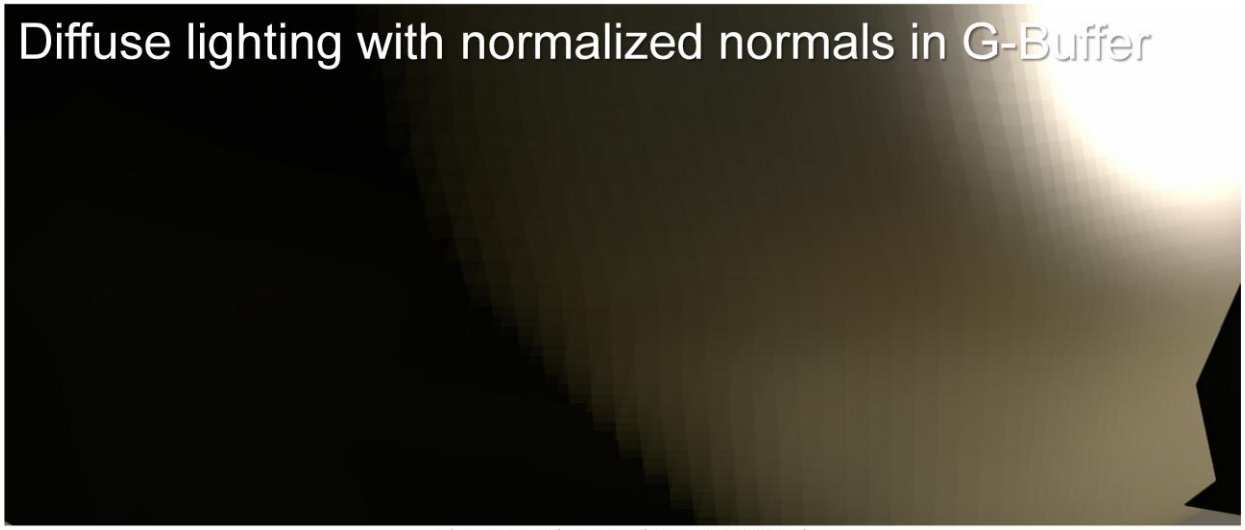
This is a small **breakdown** of the existing storage methods.

It shows how many bits it uses out of **total number of 24 bits**.

Both old techniques uses ~**17 bits**, which explains the low normals quality.

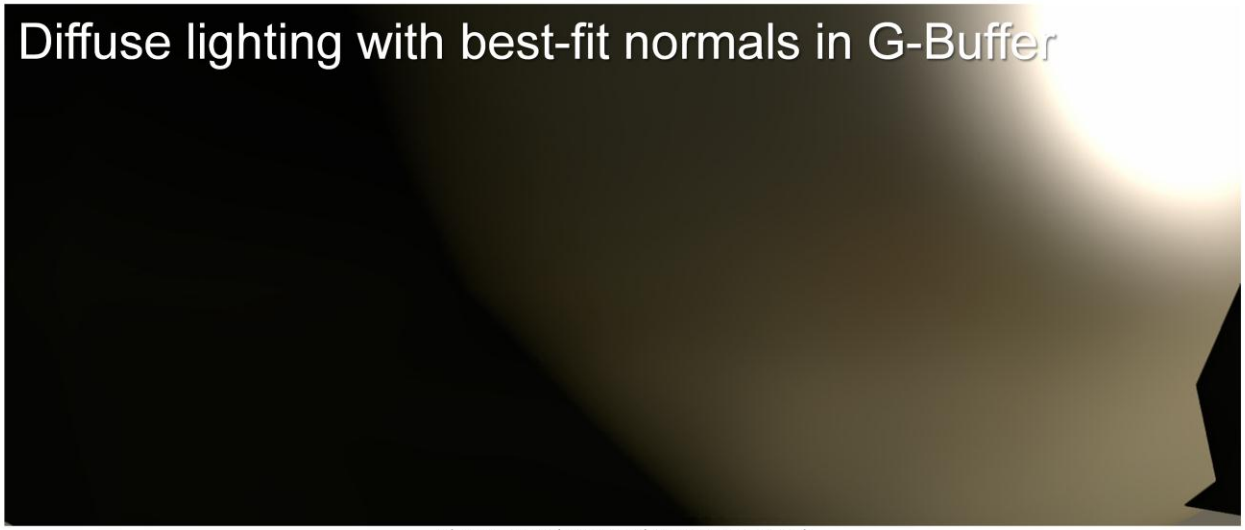
However the proposed technique uses almost the **whole range of 24 bits**, beating the old ones with almost **two orders of magnitude**.

Normals precision in G-Buffer, example



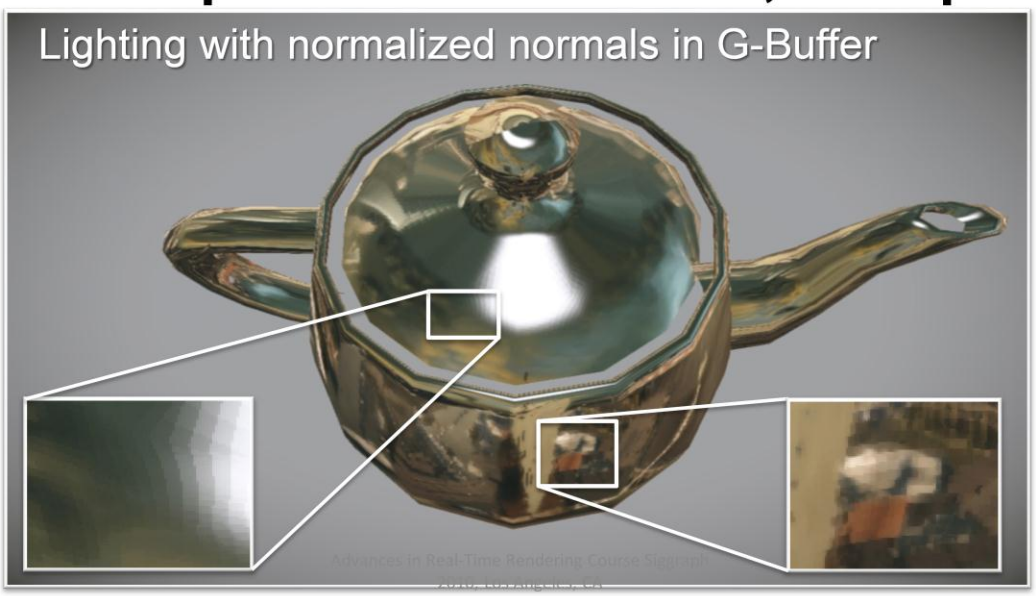
Here is a series of screenshots demonstrating the results of the new technique compared to old ones.

Normals precision in G-Buffer, example



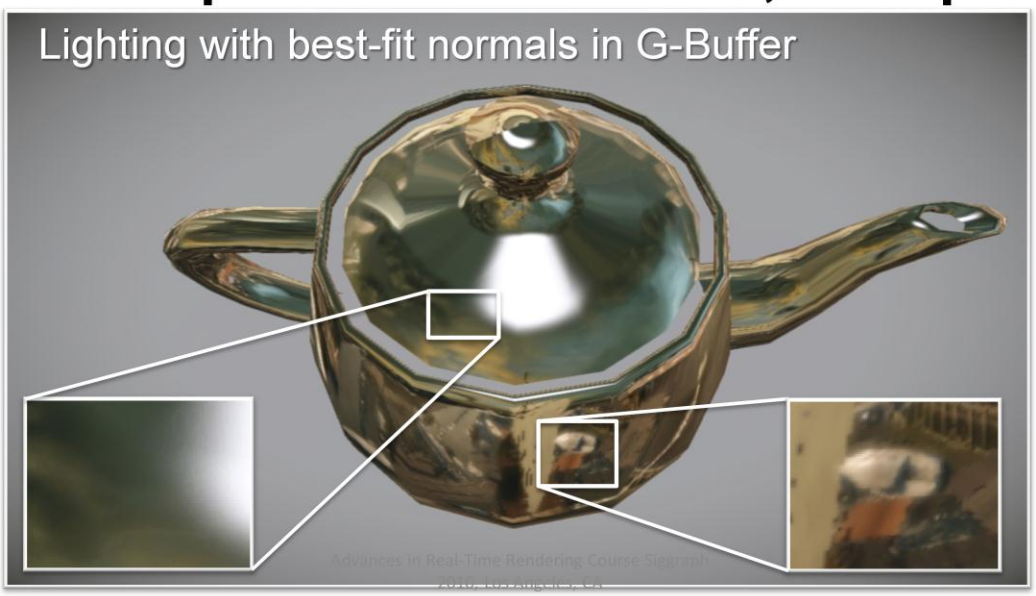
Here is a series of screenshots demonstrating the results of the new technique compared to old ones.

Normals precision in G-Buffer, example



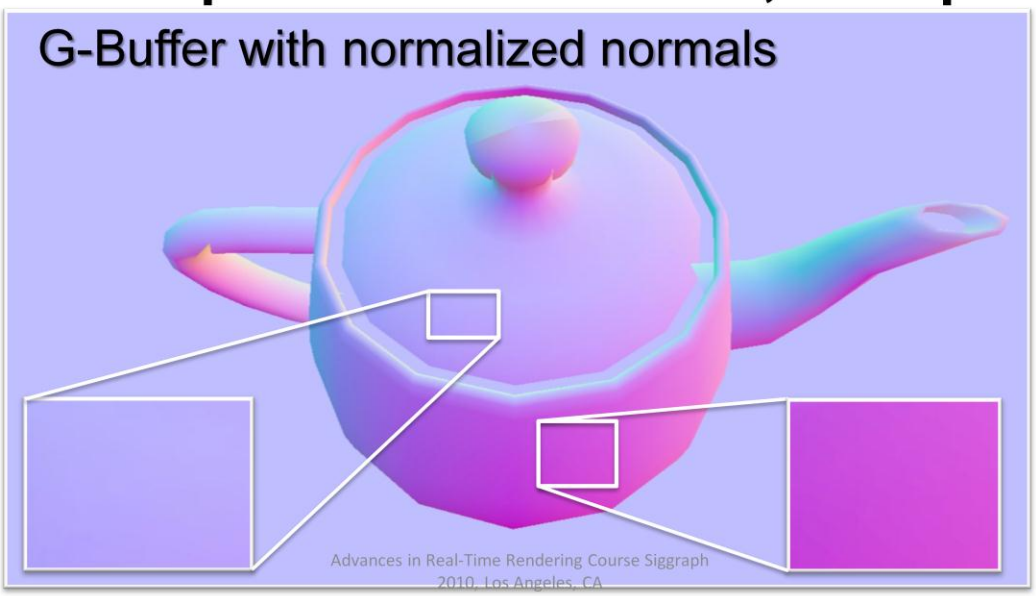
Here is a series of screenshots demonstrating the results of the new technique compared to old ones.

Normals precision in G-Buffer, example



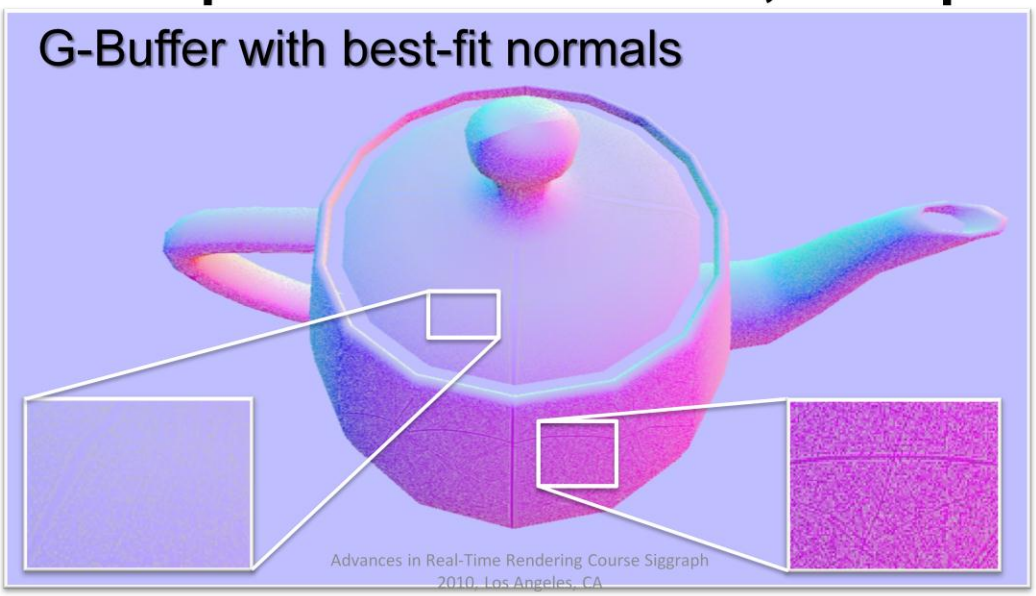
Here is a series of screenshots demonstrating the results of the new technique compared to old ones.

Normals precision in G-Buffer, example



Here is a series of screenshots demonstrating the results of the new technique compared to old ones.

Normals precision in G-Buffer, example



Here is a series of screenshots demonstrating the results of the new technique compared to old ones.

PHYSICALLY-BASED BRDFS

Advances in Real-Time Rendering Course Siggraph
2010, Los Angeles, CA

This part is devoted to the importance of BRDF normalization and energy conservation.

Lighting consistency: Phong BRDF

- What's your Phong BRDF? $\boldsymbol{s = (r \cdot l)_+^{\,n}}$?
  - Make sure you normalize it: $s = \frac{\boldsymbol{n+2}}{\boldsymbol{2\pi}} (r \cdot l)_+^{\,n}$
    - Energy preserving: very important for HDR post-processing
    - Consistent with non-analytical lighting, such as environment mapping
    - Simpler artistic control over specular
- Learn more details at „Physically Based Shading Models" course from Nauty Hoffman

Advances in Real-Time Rendering Course Siggraph 2010, Los Angeles, CA

It turned out that the section of this talk highly overlaps with the Physically based shading model" course presented at the same time at the SIGGRAPH:

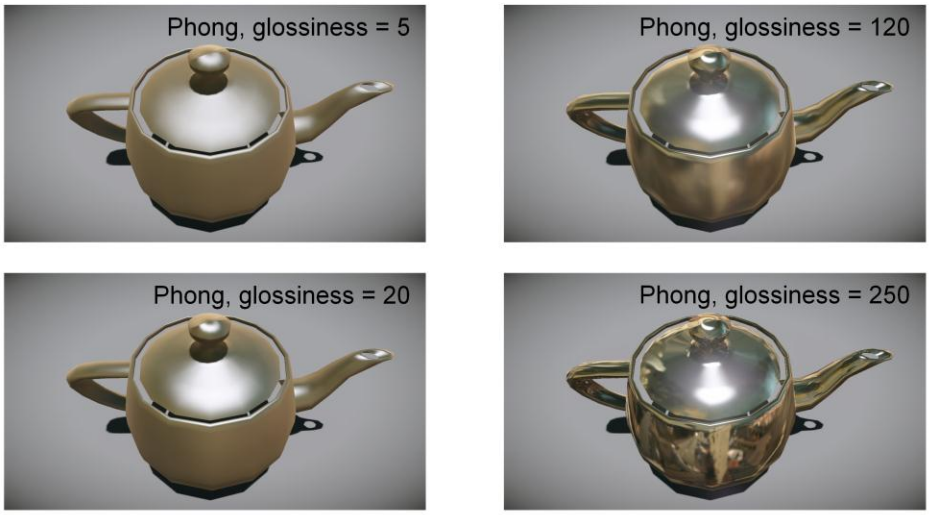
http://renderwonk.com/publications/s2010-shading-course/hoffman/s2010_physically_based_shading_hoffman_b.pdf

I'd highly recommend to refer to this course in order to understand the advantages of BRDF normalization in more details and examples.

Briefly speaking, here are the most important ones:

- Energy preserving: very important for HDR post-processing
- Consistent with non-analytical lighting, such as environment mapping
- Simpler artistic control over specular

Consistent lighting example



Advances in Real-Time Rendering Course Siggraph
2010, Los Angeles, CA

Here are the results for different glossiness values for Phong BRDF. The reflectance coefficient of the material remains the same in all images.

However please notice the intensity of analytic reflection coming from the sun and how **consistent** it is with the precomputed environment lighting.

Consistent lighting example

Here are some in-game examples of the consistency across the precomputed and analytical lighting.

Consistent lighting example

Here are some in-game examples of the consistency across the precomputed and analytical lighting.

HDR…
VS BANDWIDTH VS PRECISION

Advances in Real-Time Rendering Course Siggraph
2010, Los Angeles, CA

This part of the talk is devoted to the **low-bandwidth, high-quality HDR pipeline** on **consoles**.

HDR on consoles

- Can we achieve bandwidth the same as for LDR?
- **PS3**: RGBK (aka RGBM) compression
  - RGBA8 texture – the same bandwidth
  - RT read-backs solves blending problem
- **Xbox360**: Use R11G11B10 texture for HDR
  - Same bandwidth as for LDR
    - Remove _AS16 suffix for this format for better cache utilization
  - Not enough precision for linear HDR lighting!

Advances in Real-Time Rendering Course Siggraph 2010, Los Angeles, CA

The ideal goal for us is to achieve the complete HDR rendering pipeline with no additional bandwidth or performance overhead.

Thus we came up with **RGBK compression** for all our HDR buffers on **PS3**. We use RGBA8 render targets for light accumulation buffers (both diffuse and specular) and for the final HDR shaded scene render target.

We use custom blending with read-backs on PS3 in order to achieve proper blending operations with RGBK encoding.

However it works unless you have some overlapped geometry to draw. This is the only case with transparent objects for us. So, as we draw transparent objects in the end, we decode the RGBK buffer into a fat ARGB16F render target before rendering any transparency. This provided a small overhead, as the decoding is conjoined with the full-screen global fog pass.

On the other hand there is **no** read-back capability on Xbox 360's GPU. Thus we render the whole scene into an ARGB16 render target **in the EDRAM** and **resolve** it into an **R11G11B10** texture in the system memory.

Thus the bandwidth outside EDRAM remains the **same as for LDR** rendering. The R11G11B10 format gets **expanded** into ARGB16 in the texture cache **by default**. To

avoid that, we would recommend to **remove _AS16** suffix from this format.

However even 11 bits per channel provide way too **low storage precision** for HDR lighting in linear space.

HDR on consoles: dynamic range

- Use dynamic range scaling to improve precision
- Use average luminance to detect the efficient range
  - Already computed from previous frame
- Detect lower bound for HDR image intensity
  - The final picture is LDR after tone mapping
  - The LDR threshold is 0.5/255=1/510
  - Use inverse tone mapping as estimator

Advances in Real-Time Rendering Course Siggraph
2010, Los Angeles, CA

Thus we decided to use a **floating range window** for HDR.

We define **minimum and maximum** values for HDR buffers and **rescale all input light parameters** on CPU before submitting it to the GPU.

The range window is based on the **average frame luminance**. The maximum value is **adjusted** empirically for each level.

However the **lower bound** can be computed **analytically**, based on the **tone mapping operator** in use.

The idea behind that is that the HDR image becomes **LDR** in order to be presented on the **LDR display**.

However the **lower threshold** for the majority of displays does not exceed **0.5/255**.

Using this lower bound of the **output color**, we can apply an **inverse tone mapping operator** to it and find a minimum **HDR** value **analytically**.

HDR on consoles: lower bound estimator

- Some examples (l – average frame luminance):
  - Exponential tone mapping $T(c) = 1 - e^{-\frac{c}{l}}$
    - **Estimator** $\boldsymbol{c_{min}} = -l \ln(\frac{510}{509}) \approx 0.00196271l$
  - Filmic tone mapping $T(c' \equiv c/l - 0.004) \approx (\frac{c'(6.2\,c'+0.5)}{c'(6.2\,(c'+1.7)+0.06}$)$^{2.2}$
    - **Estimator** $\boldsymbol{c_{min}} \approx 0.0119052l$
  - Filmic tonemapping is much less sensitive to dark colors

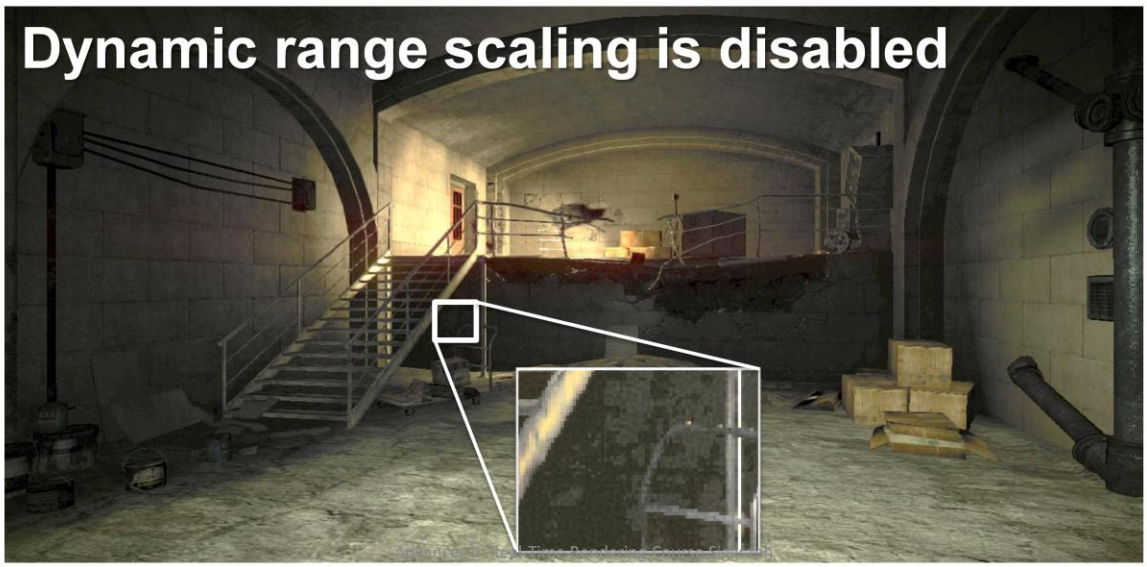
On this slide there are **two tone mapping operators** we use. The *l* is an **average luminance** in these equations. The c is an **input HDR color**.

We use value for *l* from **previous frame**, as it changes smoothly due to **eye adaption**.

The **exponential tone mapping** operator is very sensitive to dark colors.

However the **filmic tone mapping** operator is rather **moderately tolerable** to dark colors.

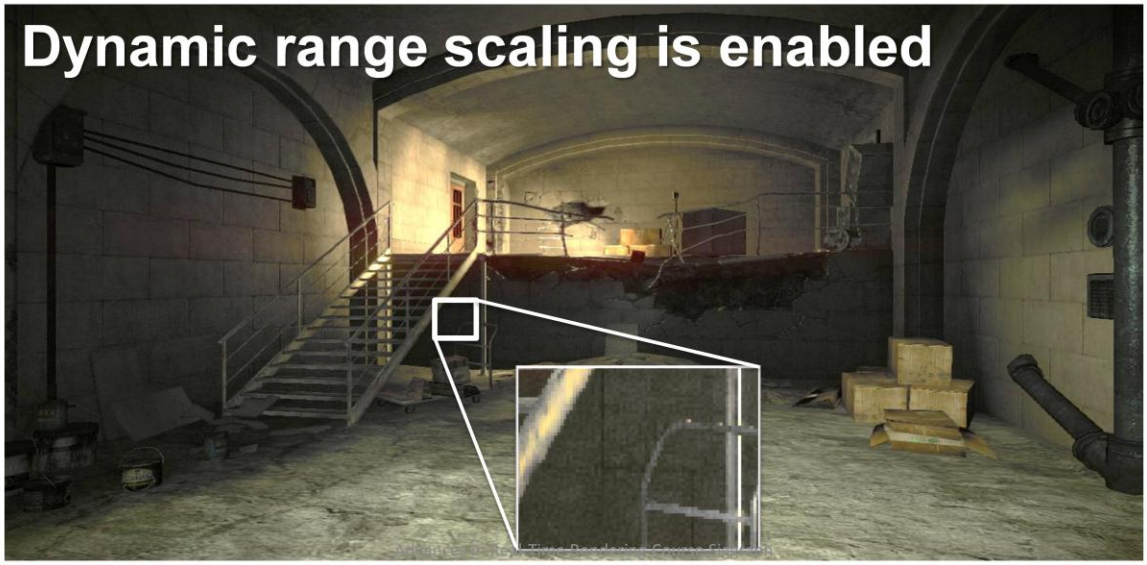
HDR dynamic range example



2010, Los Angeles, CA

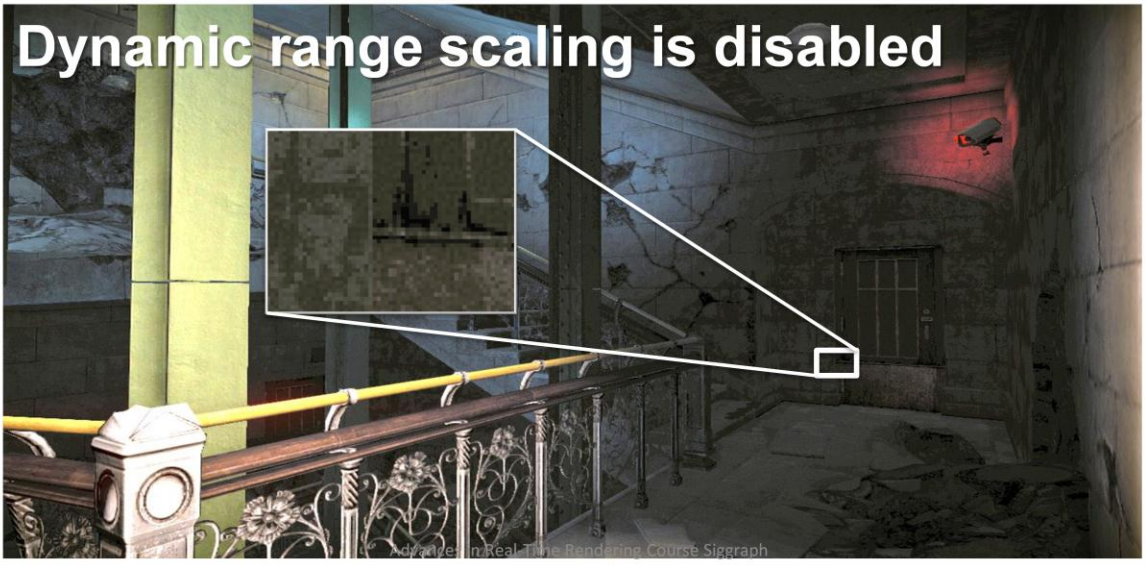
Here are the resulting screenshots made on Xbox 360 with **R11G11B10** HDR buffers.

HDR dynamic range example



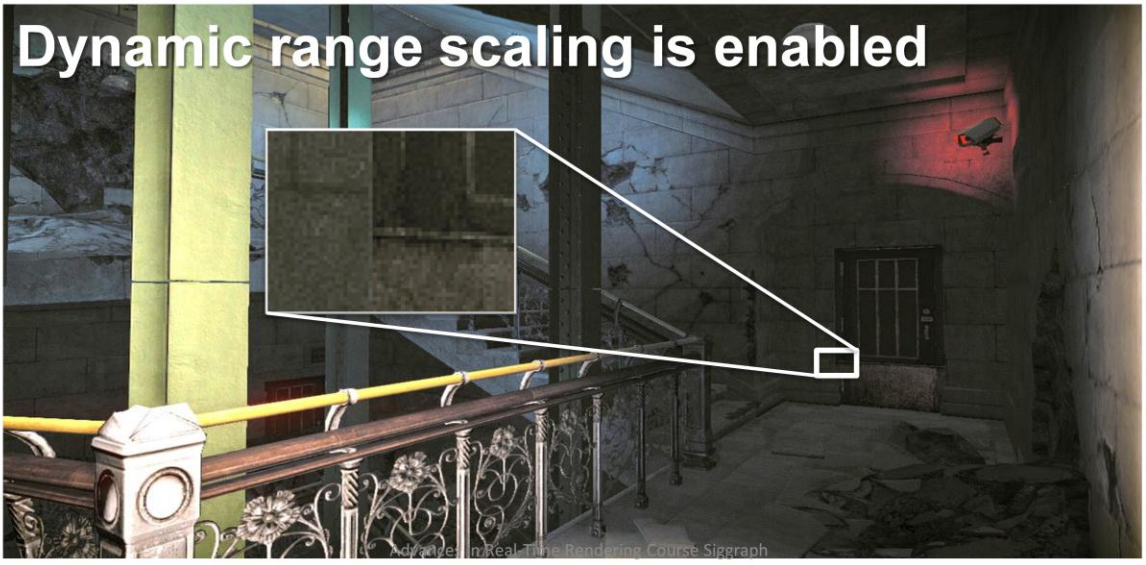
Here are the resulting screenshots made on Xbox 360 with **R11G11B10** HDR buffers.

HDR dynamic range example



Here are the resulting screenshots made on Xbox 360 with **R11G11B10** HDR buffers.

HDR dynamic range example



Here are the resulting screenshots made on Xbox 360 with **R11G11B10** HDR buffers.

LIGHTING TOOLS: CLIP VOLUMES

Advances in Real-Time Rendering Course Siggraph 2010, Los Angeles, CA

This small section is devoted to rather small, but extremely **powerful** concept of **tagging areas and volumes** with **deferred pipeline**.

Clip Volumes for Deferred Lighting

- Deferred light source w/o shadows tend to bleed:
  - Shadows are expensive
- Solution: use artist-defined clipping geometry: **clip volumes**
  - Mask the stencil in addition to light volume masking
  - Very cheap providing fourfold stencil tagging speed

Advances in Real-Time Rendering Course Siggraph
2010, Los Angeles, CA

There is a common problem of light bleeding become apparent with the deferred approach: the boundaries of lighting are not controllable.

E.g. the deferred light placed in one room can bleed through the wall into another room.

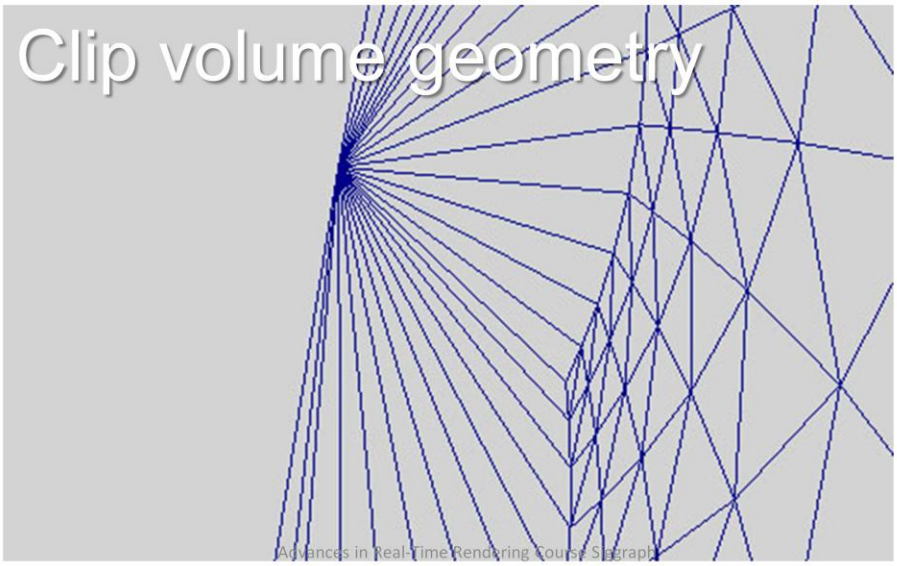
Thus we decided to provide a tool for artists that they can specify a custom stencil culling geometry for each light source in the scene.

This approach is very cheap provided that the clipping geometry is rather coarse and the stencil tagging is very fast on consoles.

Clip Volumes example

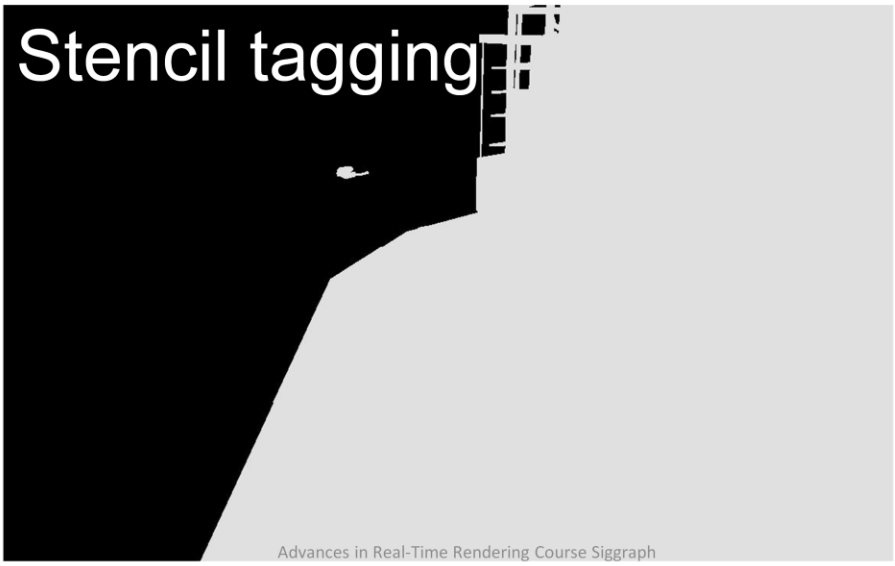
This is an example of how the clip volume works for some particular light.

Clip Volumes example



This is an example of how the clip volume works for some particular light.

Clip Volumes example



This is an example of how the clip volume works for some particular light.

Clip Volumes example

This is an example of how the clip volume works for some particular light.

Clip Volumes example

This is an example of how the clip volume works for some particular light.

DEFERRED LIGHTING AND ANISOTROPIC MATERIALS

Advances in Real-Time Rendering Course Siggraph
2010, Los Angeles, CA

This part of the talk is devoted to the problem of materials variety with deferred pipeline.

Anisotropic deferred materials

- G-Buffer stores only **normal** and **glossiness**
  - That defines a BRDF with a **single Phong lobe**
- We need **more lobes** to represent anisotropic BRDF
  - Could be extended with **fat G-Buffer** (**too heavy** for production)
- Consider one **screen pixel**
  - We have **normal** and **view vector**, thus BRDF is defined on sphere
  - Do we need all these lobes to illuminate this pixel?
  - Lighting distribution is **unknown** though

Advances in Real-Time Rendering Course Siggraph
2010, Los Angeles, CA

The core idea is **preserve the current bandwidth** and interfere with the deferred pipeline as less as possible.

The current G-Buffer layout can store **only a Phong BRDF** as it was mentioned before.

However in order to represent a complex BRDF, such as a highly anisotropic Ward BRDF, we need much **more lobes**. This can be implemented by **expanding the G-Buffer layout** and adding **more information** about BRDF.

However the whole lighting pipeline would **suffer** from that. Moreover, taking into account the complex **layered lighting** and increasing data size stored in G-Buffer per pixel, the **bandwidth grows too quickly**. This fact makes this approach **unaffordable** for the actual **game production**.

However if we consider **one pixel** of the G-Buffer, we know a lot of **fixed** parameters, such as normal, view direction etc. (note that for a given particular pixel it is **fixed and known** at the G-Buffer generation time). Thus the BRDF can be defined as a **function on the sphere**.

But something that is unknown at that time for us is **lighting conditions**.

Anisotropic deferred materials, part I

- Idea: Extract the major Phong lobe from NDF
  - Use microfacet BRDF model [CT82]: $\rho(x,o,i) = \frac{D(x,h)S(x,o)S(x,i)F(x,o,i)}{4(i\cdot n)(o\cdot n)}$
  - Fresnel and geometry terms can be deferred
  - Lighting-implied BRDF is proportional to the NDF: $\boldsymbol{\rho(x,o,i) \sim D(x,h)}$
- Approximate NDF with **Spherical Gaussians** [WRGSG09]
  - Need only ~7 lobes for Anisotropic Ward NDF

Advances in Real-Time Rendering Course Siggraph 2010, Los Angeles, CA

So the general idea behind the approach is to do a **Phong lobe extraction** at the G-Buffer generation time.

There are several advantages:

- G-Buffer layout, memory footprint and bandwidth **remains the same**!
- Support for **arbitrary** type and complexity of BRDF
- Completely **orthogonal and transparent** for the subsequent lighting pass

First, consider a microfacet BRDF model.

The Fresnel and geometry terms of this model can be completely **decoupled** from lighting and applied in a subsequent shading pass per object.

Thus the part of BRDF that affects the lighting directly and cannot be decoupled from the lighting pass, is the **Normal Distribution Function**.

We approximate the NDF of the BRDF with **Spherical Gaussians** in spirit of [WRGSG09]. Note that most of common BRDFs have **analytical formulas** for this basis. Another important fact is the compactness of the representation. E.g. anisotropic Ward BRDF can be represented by only **7 basis functions** in a vast majority of cases.

Anisotropic deferred materials, part II

- Approximate lighting distribution with **SG** per object
  - Merge SG functions if appropriate
  - Prepare several approximations for huge objects
- Extract the principal Phong lobe into G-Buffer
  - Convolve lobes and extract the mean normal (next slide)
- Do a usual deferred Phong lighting
- Do shading, apply Fresnel and geometry term

Advances in Real-Time Rendering Course Siggraph 2010, Los Angeles, CA

After we've got a compact representation of the BRDF for **current screen pixel**, we can extract the **principal Phong lobe** out of it.

In order to do that, we need to know **lighting conditions** for **this pixel**.

We approximate the lighting conditions with **Spherical Gaussians per object**.

The approximation is done on **CPU**. Similar lobes can be **merged together** at this time in order to minimize the representation.

If an object is huge or long, we can prepare **several lighting representations** in different places of the object and interpolate between them in the vertex shader.

So in order to extract the Phong lobe, we **convolve** each SG function of the **BRDF** representation with each SG function of the **lighting** representation. We use the result to **weight** the normal and **fit** the glossiness factor after each convolution.

Note that the SG basis is **not orthogonal**, so the cost is **polynomial O(n*m)**, where **n** is number of basis function representing lighting and **m** is the number of function of BRDF representation.

After the normal and the glossiness are extracted, we output it as a **usual Phong lobe** into G-Buffer.

Then we do a **usual** deferred Phong lighting with many lights. Note that normals and glossiness are already adjusted to best approximate the BRDF by a Phong BRDF for each **particular pixel**.

Thus the variety of BRDFs becomes completely **invisible** at the **deferred lighting**

stage. That fact itself **unifies** the pipeline and has a very **positive performance implications**.

And we apply the Fresnel and geometry term in the end **after the lighting is done**.

Extracting the principal Phong lobe

- CPU: prepare SG lighting representation per object
- Vertex shader:
  - Rotate SG representation of BRDF to local frame
  - Cut down number of lighting SG lobes to ~7 by hemisphere
- Pixel shader:
  - Rotate SG-represented BRDF wrt tangent space
  - Convolve the SG BRDF with SG lighting
  - Compute the principal Phong lobe and output it

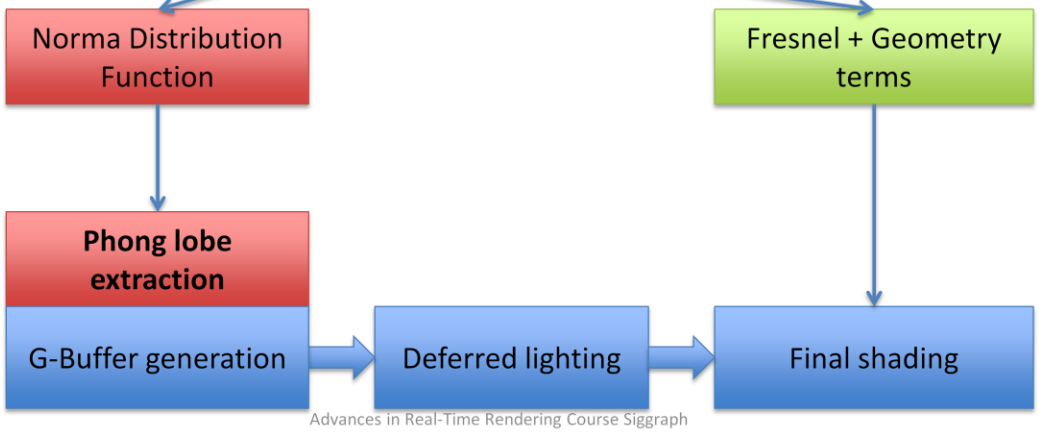
Advances in Real-Time Rendering Course Siggraph
2010, Los Angeles, CA

So, the extraction pipeline is as follows:

- CPU: Approximate lighting conditions with SG for each object (e.g. in object's center or multiple points)
- Vertex shader:
  - generate SG coefficients for BRDF representation
  - Cull invisible SG lobes of lighting representation (based on hemisphere of visibility)
- Pixel shader:
  - Do a local rotation of BRDF based on normal maps (if any)
  - Convolve each SG of BRDF with each SG of lighting representation
  - Accumulate principal Phong lobe and output it

Anisotropic deferred materials

$$\rho(x,o,i) = \frac{D(x)S(x,o)S(x,i)F(x,o,i)}{4(i\cdot n)(o\cdot n)}$$

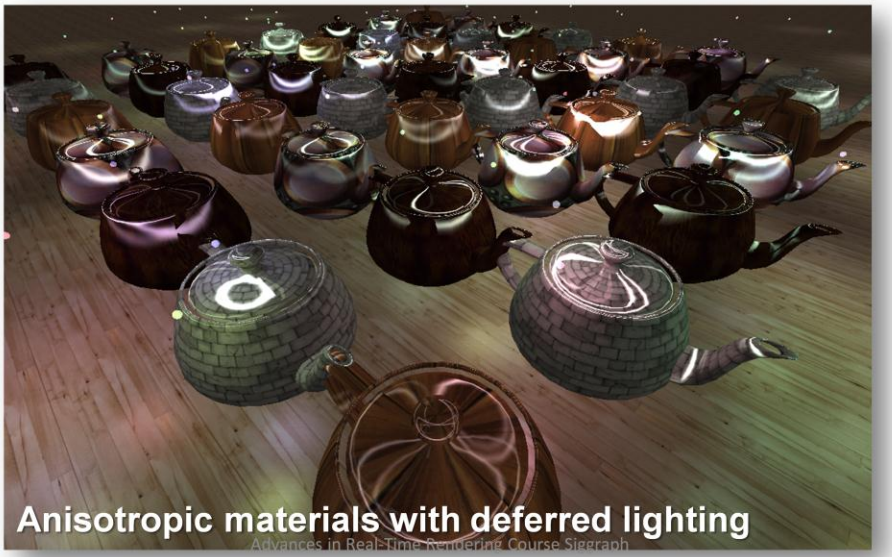


Advances in Real-Time Rendering Course Siggraph 2010, Los Angeles, CA

This is a diagram of **decoupling** the BRDF complexity from the lighting complexity.

As you can see, the BRDF complexity is completely **eliminated** from the lighting pass.

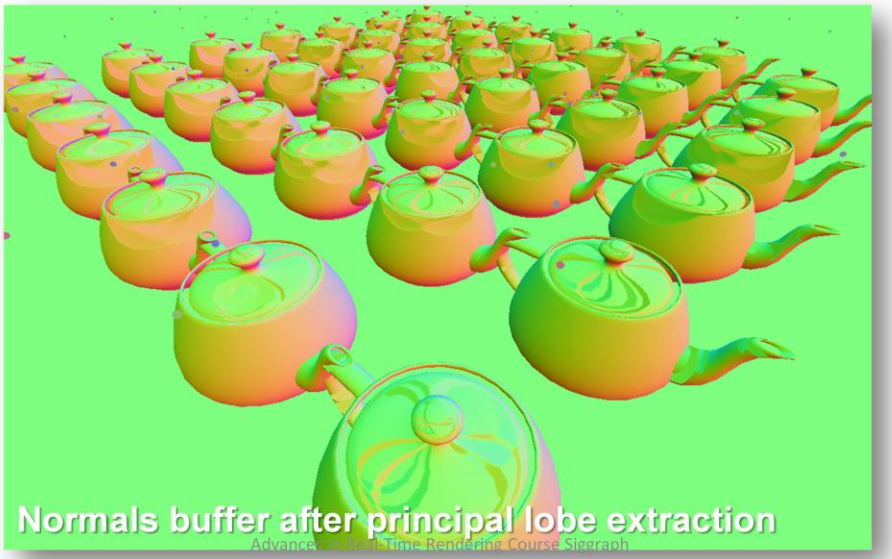
Anisotropic deferred materials



Advances in Real-Time Rendering Course Siggraph 2010, Los Angeles, CA

This is an example of the technology at work

Anisotropic deferred materials



This is an example of the technology at work

Anisotropic deferred materials: why?

- Cons:
  - Imprecise lobe extraction and specular reflections
    - But: see [RTDKS10] for more details about perceived reflections
  - Two lighting passes per pixel?
    - But: hierarchical culling for prelighting: Object → Vertex → Pixel
- Pros:
  - No additional information in G-Buffer: **bandwidth preserved**
  - Transparent for subsequent lighting pass
  - **Pipeline unification**: shadows, materials, shader combinations

Advances in Real-Time Rendering Course Siggraph 2010, Los Angeles, CA

There are several obvious **disadvantages** of this technique:

- Lobes are extracted **not precisely**. Specular highlight appears **slightly shifted** due to **per-object lighting approximation**
  - However this turns out to be a rather small issue due to the **human perception limitations**. Please see **[RTDKS10]** for more details.
- Essentially we do the lighting **twice**: the **first time** during the G-Buffer generation pass, and **the second time** is the actual deferred lighting.
  - However the first pass is **hierarchical** with **merging** and **culling**. And it is executed only for pixels **covered by objects** with complex BRDF, which is a reasonable cost.

Besides that, there are several obvious **advantages** of the technique:

- The G-Buffer **remains the same**!
- The lighting pipeline is **unchanged**.
- The materials pipeline stays **unified**, as deferred lights can be **applied in-place** for objects with arbitrary BRDFs. That means we don't need to shade it **separately** in forward pass; we don't need to **store all shadow maps** and compute many **shader permutations** for forward shading with different number and types of light sources etc.

DEFERRED LIGHTING AND ANTI-ALIASING

Advances in Real-Time Rendering Course Siggraph
2010, Los Angeles, CA

This part of the talk is devoted to the aliasing problem in deferred pipeline and the proposed solution to it.

Aliasing sources

- Coarse surface sampling (rasterization)
  - Saw-like jaggy edges
  - Flickering of highly detailed geometry (foliage, gratings, ropes etc.) because of sparse sampling
    - Any post MSAA (including MLAA) won't help with that
- More aliasing sources
  - Sparse shading
    - Sudden spatial/temporal shading change
  - Sparse lighting etc etc.

There are **multiple** sources of aliasing in rendering pipeline.

Some aliasing artifacts such as **jaggy edges** can be addressed with some **postprocessing techniques**, such as Morphological Anti-Aliasing.

However there are some **fundamentally different sources of aliasing** that cannot be solved by postprocessing, as it require **supersampling**.

E.g. flickering of highly discontinuous geometry such as foliage or thin ropes are caused by **per-screen-pixel visibility** resolution, which is considered too sparse for such kind of geometry. So in this case the **visibility** should be resolved at **higher resolution** with some **supersampling** technique.

Another source of aliasing can be **discontinuities in shading**, e.g. **reflections** of high-frequency or discontinuous **shadows** etc. This means we need to supersample the **shading** per pixel for.

Hybrid anti-aliasing solution

- Post-process AA for near objects
  - Doesn‘t supersample
  - Works on edges
- Temporal AA for distant objects
  - Does temporal supersampling
  - Doesn‘t distinguish surface-space shading changes
- Separate it with stencil and non-jitterred camera

Advances in Real-Time Rendering Course Siggraph
2010, Los Angeles, CA

However we noticed that in the majority of scenes, the supersampling is required mostly only for **distant objects**, where the GPU **fails** to approximate shading/mip-mapping/visibility per pixel.

Thus we propose a **hybrid solution** that solves **different** aliasing problems based on the **distance** to camera (some other parameters could be taken into account as well though).

We apply a **post-processing edge-detection** algorithm on **close-up objects**. That solves the problem with **edges**. Note that there is almost **no supersampling required** for near objects, as they are sampled and shaded at **sufficient rate in screen space**.

We **supplement** it with **temporal supersampling** based on **reprojection** of previous results. This temporal supersampling is done only for **distant objects**.

We render different parts of the scene with different techniques and separate it with **stencil masking**.

Post-process Anti-Aliasing

- We decided not to invest compute power to MLAA
  - Doesn‘t remove aliasing completely anyway
  - There are cheaper ways to blur edges
- We use very simple yet efficient filter
  - Compute color deviation: $d = \sum_{adj} |s_i - s_0|$
  - Blur current pixel based on deviation from current color
  - Use only 4 adjacent samples
    - Take into accound all 8 adjacent pixels with bilinear interpolation

As the edge-detection algorithm needs to smooth the shapes of close-up objects only. Thus we decided to make it **as simple as possible**.

So we compute the **color deviation** from **8 surrounding screen pixels** in 4 lookups with bilinear filtering and then decide how strong the current pixel should be **blurred**.

Temporal Anti-Aliasing

- Use temporal reprojection with cache miss approach
  - Store previous frame and depth buffer
  - Reproject the texel to the previous frame
  - Assess depth changes
  - Do an accumulation in case of small depth change
- Use sub-pixel temporal jittering for camera position
  - Take into account edge discontinuities for accumulation
- See [NVLTI07] and [HEMS10] for more details

We use screen-space sub-pixel temporal camera jittering. That means we shift the camera position with a subsample pattern in screen space for each frame. So for still camera we can accumulate sub-pixel supersampling over time.

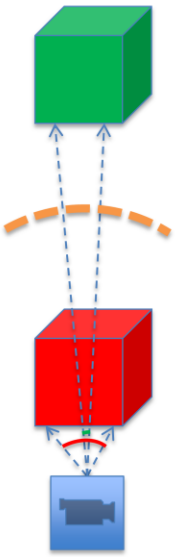
In case of dynamic objects and/or camera movements, we use cache miss philosophy based on the depth changes across frames.

In order to accumulate super-sampled visibility on the edges, we check for depth discontinuities in the frame's depth buffer.

Please see [NVLTI07] and [HEMS10] for more details.

Hybrid anti-aliasing solution

- Separation by distance guarantees small changes of view vector for distant objects
  - Reduces the fundamental problem of reverse temporal reprojection:
    view-dependent changes in shading domain
  - Separate on per-object base
    - Consistent object-space shading behavior
    - Use stencil to tag an object for temporal jittering



Advances in Real-Time Rendering Course Siggraph
2010, Los Angeles, CA

The reason is that the reprojection in it is based on the **depth buffer**. Thus it is impossible to take into account **shading-space local changes** on objects. That might lead to **ghosting** shadows, reflections and so on, if we apply it onto close-up objects.

Hybrid anti-aliasing example

This image shows two parts of the scene (“close-up” case and “distant” case) and demonstrates how different techniques are applied onto different parts of the scene.

Please watch the **supplementary video** for more details.

Hybrid anti-aliasing example

This image shows two parts of the scene (“close-up” case and “distant” case) and demonstrates how different techniques are applied onto different parts of the scene.

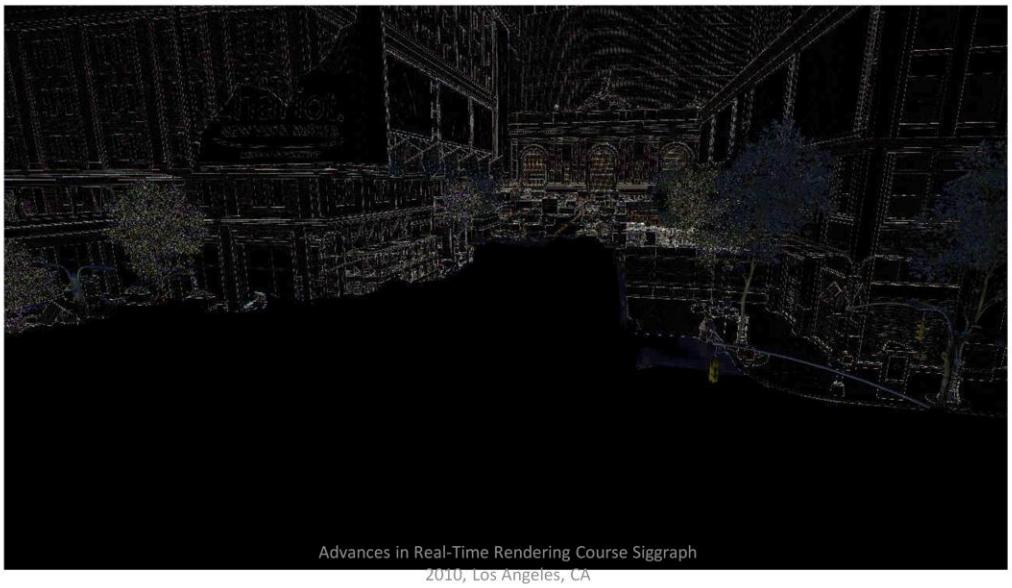
Please watch the **supplementary video** for more details.

Hybrid anti-aliasing example

This image shows two parts of the scene (“close-up” case and “distant” case) and demonstrates how different techniques are applied onto different parts of the scene.

Please watch the **supplementary video** for more details.

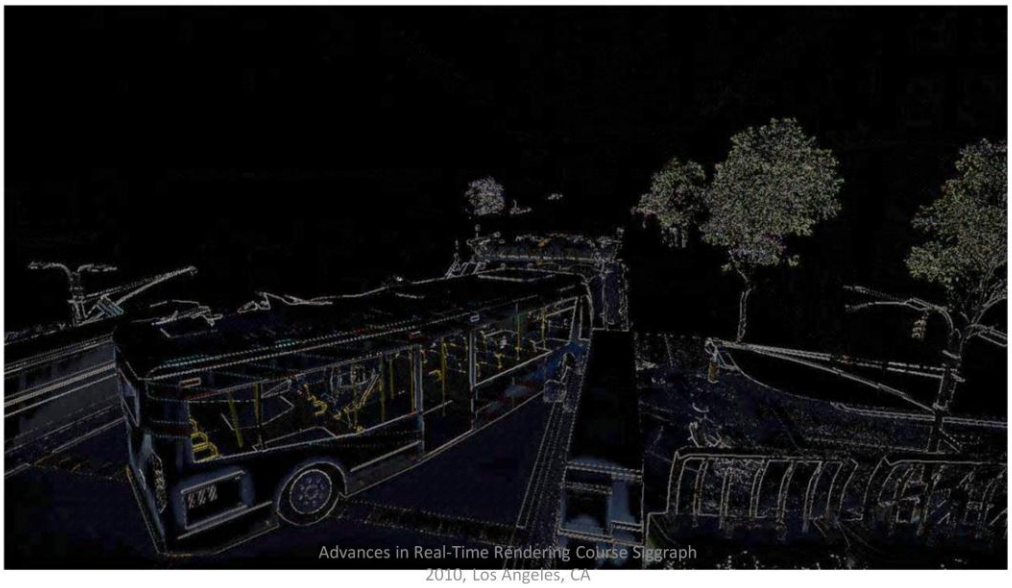
Temporal AA contribution



This image shows two parts of the scene (“close-up” case and “distant” case) and demonstrates how different techniques are applied onto different parts of the scene.

Please watch the **supplementary video** for more details.

Edge AA contribution



This image shows two parts of the scene (“close-up” case and “distant” case) and demonstrates how different techniques are applied onto different parts of the scene.

Please watch the **supplementary video** for more details.

Hybrid anti-aliasing video

Please watch the **supplementary video** for more details.

Conclusion

- Texture compression improvements for consoles
- Deferred pipeline: some major issues successfully resolved
  - √ Bandwidth and precision
  - √ Anisotropic materials
  - √ Anti-aliasing
- Please look at the full version of slides (including texture compression) at: http://advances.realtimerendering.com/

Improvements to **textures quality** on **consoles** was proposed.

We demonstrated a few **solutions** to several **fundamental problems** of **deferred lighting**.

Acknowledgements

- **Vaclav Kyba** from R&D for implementation of temporal AA
- **Tiago Sousa**, **Sergey Sokov** and the **whole Crytek R&D** department
- **Carsten Dachsbacher** for suggestions on the talk
- **Holger Gruen** for invaluable help on effects
- **Yury Uralsky** and **Miguel Sainz** for consulting
- **David Cook** and **Ivan Nevraev** for consulting on Xbox 360 GPU
- **Phil Scott, Sebastien Domine, Kumar Iyer** and the **whole Parallel Nsight team**

I'd like to thank all these people as well as Crytek GmbH for providing me the information and time do prepare this talk!

Thank you for your attention

QUESTIONS?

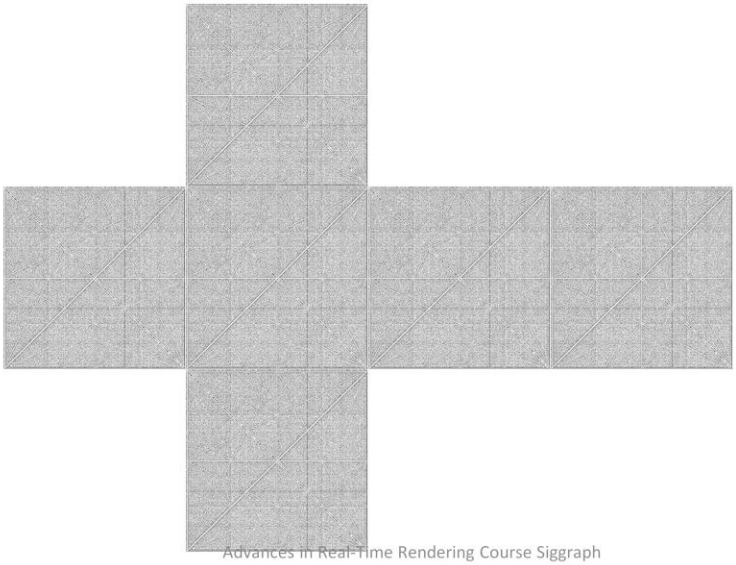
??

APPENDIX A:
BEST FIT FOR NORMALS

Function to find minimum error:

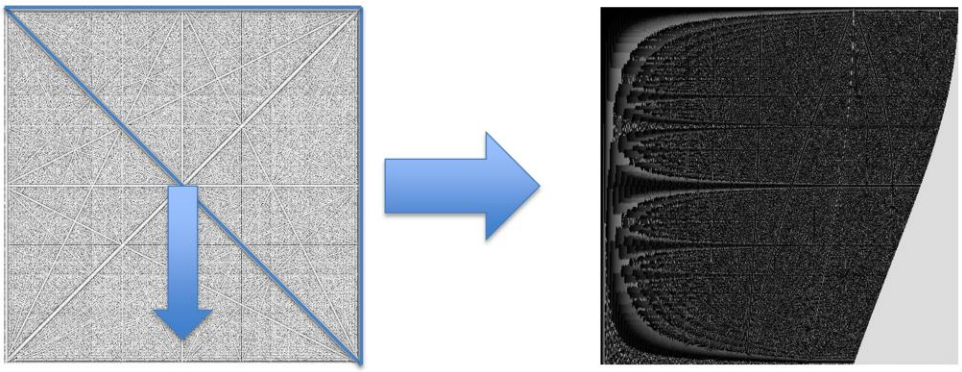
```
float quantize255(float c)
{
        float w = saturate(c * .5f + .5f);
        float r = round(w * 255.f);
        float v = r / 255.f * 2.f - 1.f;
        return v;
}

float3 FindMinimumQuantizationError(in half3 normal)
{
        normal /= max(abs(normal.x), max(abs(normal.y), abs(normal.z)));
        float fMinError = 100000.f;
        float3 fOut = normal;
        for(float nStep = 1.5f;nStep <= 127.5f;++nStep)
        {
                float t = nStep / 127.5f;

                // compute the probe
                float3 vP = normal * t;

                // quantize the probe
                float3 vQuantizedP = float3(quantize255(vP.x), quantize255(vP.y), quantize255(vP.z));

                // error computation for the probe
                float3 vDiff = (vQuantizedP - vP) / t;
                float fError = max(abs(vDiff.x), max(abs(vDiff.y), abs(vDiff.z)));

                // find the minimum
                if(fError < fMinError)
                {
                        fMinError = fError;
                        fOut = vQuantizedP;
                }
        }
        return fOut;
}
```

Cubemap produced with this function



Extract unique part

- Consider one face, extract non-symmetric part into 2D texture
  - Also divide y coordinate by x coordinate to expand the triangle to quad
  - To download this texture look at: http://advances.realtimerendering.com/



Advances in Real-Time Rendering Course Siggraph 2010, Los Angeles, CA

The texture can be copied over from this slide or downloaded from http://advances.realtimerendering.com/

Note that the 8-bits precision is sufficient if the best fit length is stored for the normal divided by maximum component.

Function to fetch 2D texture at G-Buffer pass:

```
void CompressNormalToUnsignedGBuffer(inout half4 vNormal)
{
  // renormalize (needed only if any blending or interpolation happened before)
  vNormal.rgb = normalize(vNormal.rgb);
  // get unsigned normal for the cubemap lookup
   half3 vNormalUns = abs(vNormal.rgb);
  // get the main axis for cubemap lookup
  half maxNAbs = max(vNormalUns.z, max(vNormalUns.x, vNormalUns.y));
  // get texture coordinates in a collapsed cubemap
  float2 vTexCoord = vNormalUns.z<maxNAbs?(vNormalUns.y<maxNAbs?vNormalUns.yz:vNormalUns.xz):vNormalUns.xy;
  vTexCoord = vTexCoord.x < vTexCoord.y ? vTexCoord.yx : vTexCoord.xy;
  vTexCoord.y /= vTexCoord.x;
  // fit normal into the edge of unit cube
  vNormal.rgb /= maxNAbs;
  // look-up fitting length and scale the normal to get the best fit
  half fFittingScale = tex2D(normalsSampler2D, vTexCoord).a;
  // scale the normal to get the best fit
  vNormal.rgb *= fFittingScale;
  // wrap to [0;1] unsigned form
  vNormal.rgb = vNormal.rgb * .5h + .5h;
}
```

Advances in Real-Time Rendering Course Siggraph
2010, Los Angeles, CA

Note that the G-Buffer pass is usually bound by vertex processing, so

References

- [CT81] *Cook, R. L., and Torrance, K. E.* 1981. “A reflectance model for computer graphics”, SIGGRAPH 1981
- [HEMS10] Herzog, R., Eisemann, E., Myszkowski, K., Seidel, H.-P. 2010. “Spatio-Temporal Upsampling on the GPU” I3D 2010.
- [LS10] *Loos, B.J. and Sloan, P.-P. 2010 “*Volumetric Obscurance”, I3D symposium on interactive graphics, 2010
- [NVLTI07] Nehab, D., Sander, P., Lawrence, J., Tatarchuk, N., Isidoro, J. 2007. “Accelerating Real-Time Shading with Reverse Reprojection Caching”, Conference On Graphics Hardware, 2007
- *[RTDKS10] T. Ritschel, T. Thormählen, C. Dachsbacher, J. Kautz, H.-P. Seidel,* 2010. *“*Interactive On-surface Signal Deformation”, SIGGRAPH 2010
- [SELAN07] *Selan, J. 2007.* “Using Lookup Tables to Accelerate Color Transformations”, GPU Gems 3, Chapter 24.
- [WRGSG09] *Wang., J., Ren, P., Gong, M., Snyder, J., Guo, B. 2009.* “All-Frequency Rendering of Dynamic, Spatially-Varying Reflectance”, SIGGRAPH Asia 2009