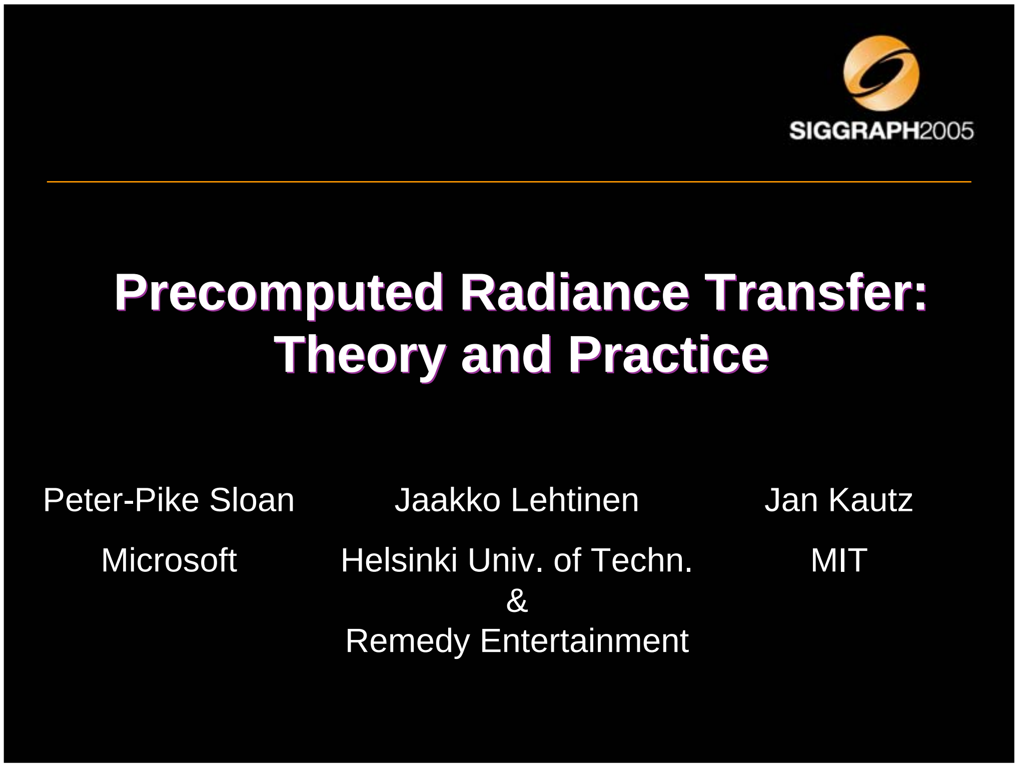
SIGGRAPH2005



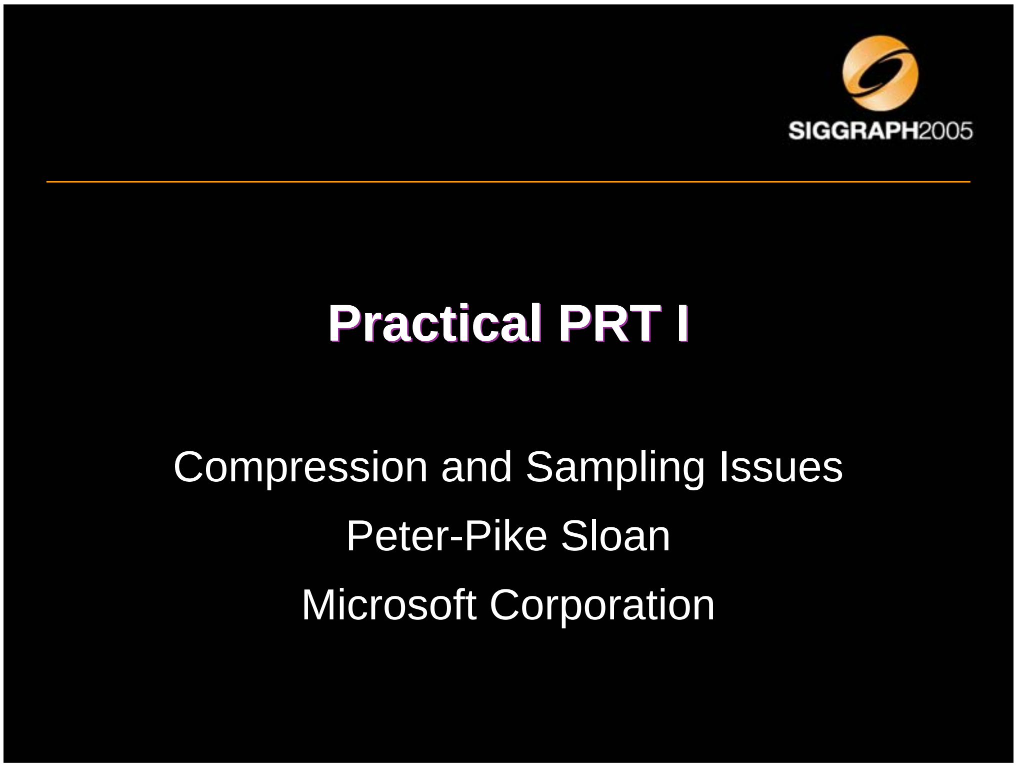
SIGGRAPH2005

# Precomputed Radiance Transfer: Theory and Practice

Peter-Pike Sloan
Microsoft

Jaakko Lehtinen
Helsinki Univ. of Techn.
&
Remedy Entertainment

Jan Kautz
MIT



SIGGRAPH2005

# Practical PRT I

Compression and Sampling Issues

Peter-Pike Sloan

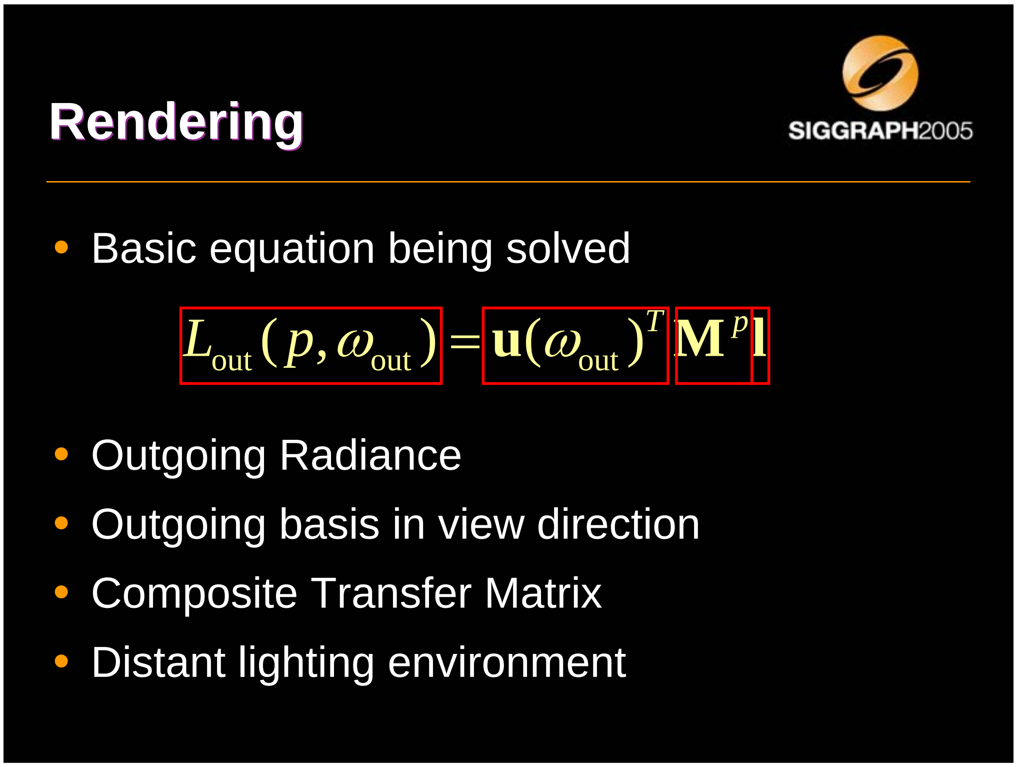
Microsoft Corporation

SIGGRAPH2005

demo

## Rendering Demos

[show basic PRT demos – don’t visualize any techniques, just show rendering in general]



## Rendering

SIGGRAPH2005

- Basic equation being solved

$$\boxed{L_{\text{out}}(p,\omega_{\text{out}})} = \boxed{\mathbf{u}(\omega_{\text{out}})^T}\boxed{\mathbf{M}^p}\boxed{\mathbf{l}}$$

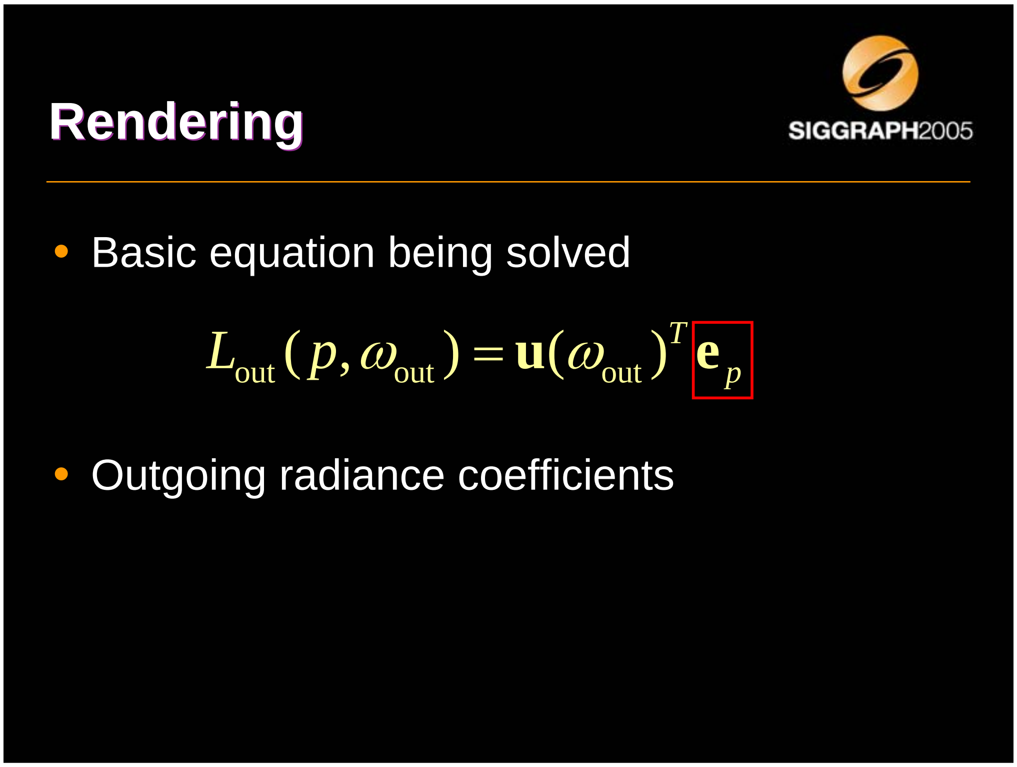
- Outgoing Radiance
- Outgoing basis in view direction
- Composite Transfer Matrix
- Distant lighting environment

We will first revisit the basic equation that is used to generate outgoing radiance at a surface point (p) in direction (omega out).

You evaluate the outgoing basis functions in the direction omega out (this is a row vector of coefficients)

You then multiply that times the transfer matrix at the point (p).

And final dot the result with the coefficients of the environment lighting vector l_env



## Rendering

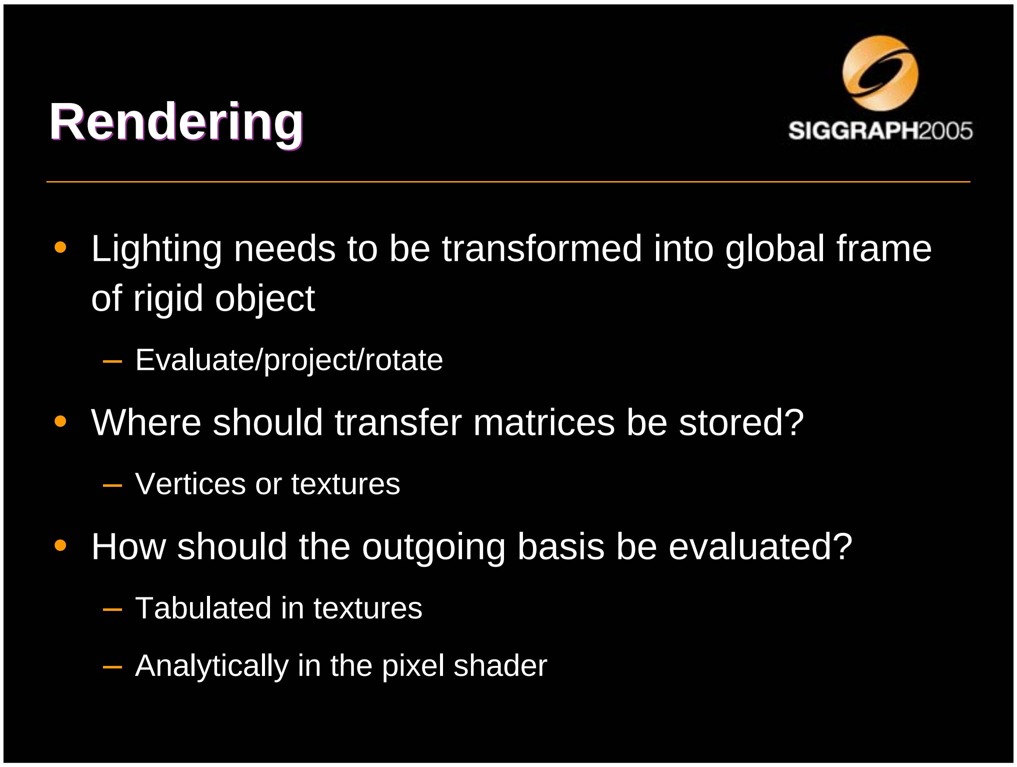
SIGGRAPH2005

- Basic equation being solved

$$L_{\text{out}}(p,\omega_{\text{out}}) = \mathbf{u}(\omega_{\text{out}})^T \boxed{\mathbf{e}_p}$$

- Outgoing radiance coefficients

Another way to look at the above equation is that we evaluate the outgoing radiance function at the point p (e_p) in the direction omega_out.



## Rendering

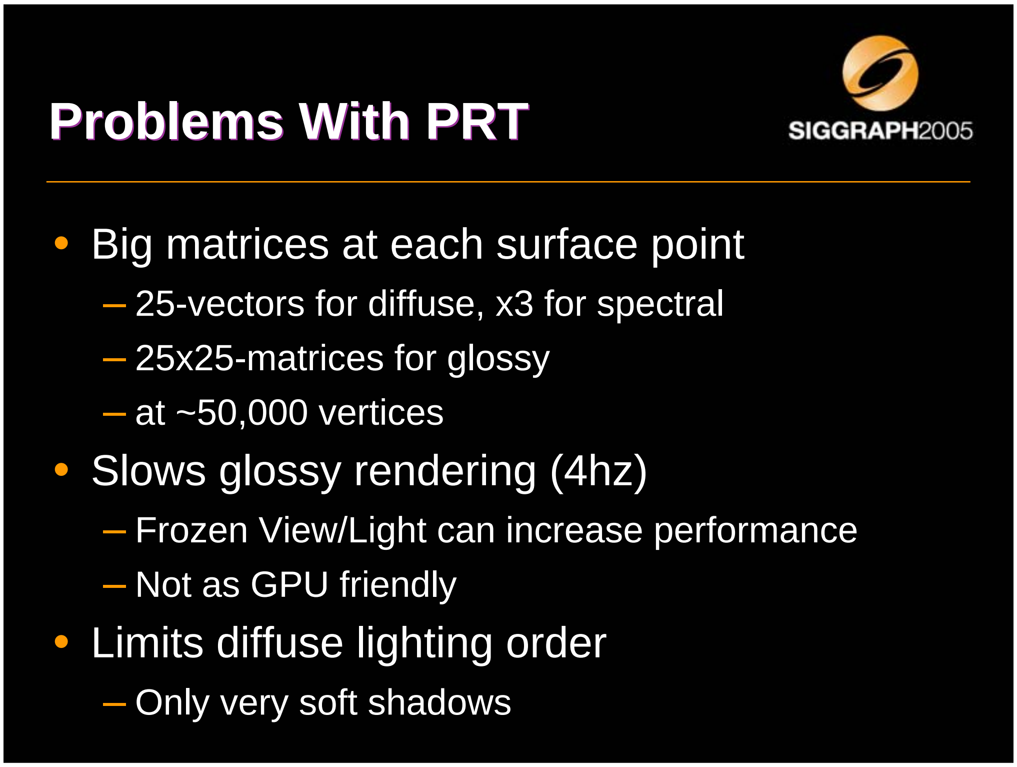
SIGGRAPH2005

- Lighting needs to be transformed into global frame of rigid object
  - Evaluate/project/rotate
- Where should transfer matrices be stored?
  - Vertices or textures
- How should the outgoing basis be evaluated?
  - Tabulated in textures
  - Analytically in the pixel shader

The light on the right hand side has to be represented in the global frame of the object. This can be achieved in several ways, for example if the light basis functions have efficient rotation formula you can just apply them, in general you can project the lighting into the basis, or you can evaluate simple analytic models as Jan talked about earlier.

The transfer matrices can be stored either at every vertex or at every texel.

The outgoing basis is generally evaluated per-pixel (based on interpolated or computed omega out.) This is generally done in a pixel shader, and can either be tabulated (for example in cube maps) or evaluated analytically.



## Problems With PRT

SIGGRAPH2005

- Big matrices at each surface point
  - 25-vectors for diffuse, x3 for spectral
  - 25x25-matrices for glossy
  - at ~50,000 vertices
- Slows glossy rendering (4hz)
  - Frozen View/Light can increase performance
  - Not as GPU friendly
- Limits diffuse lighting order
  - Only very soft shadows

There are a couple of limitations of PRT that makes it difficult to use it as discussed so far from a practical stand point.

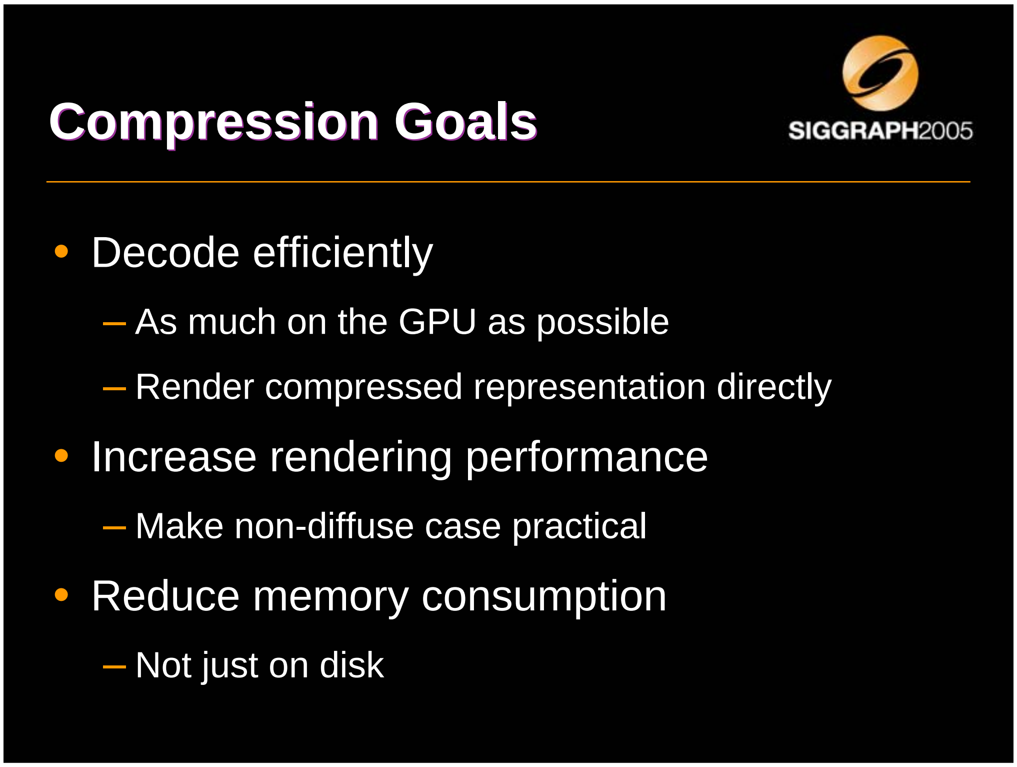
The matrices can be very large, even in the diffuse case 25d vectors are required, 3 time as many if you want to model color bleeding.

The glossy case requires 25^2 matrices, at roughly 50k vertices like most of our examples this requires over 100mb.

Glossy rendering is also slow – you can freeze the view/light to gain some performance, but this is a serious constraint.

Also glossy rendering is not as GPU friendly as the diffuse case – with much of the work being done on the CPU.

Finally, the light order for the diffuse case is limited because of the number of coefficients that can be stored per vertex on current hardware.



## Compression Goals

SIGGRAPH2005

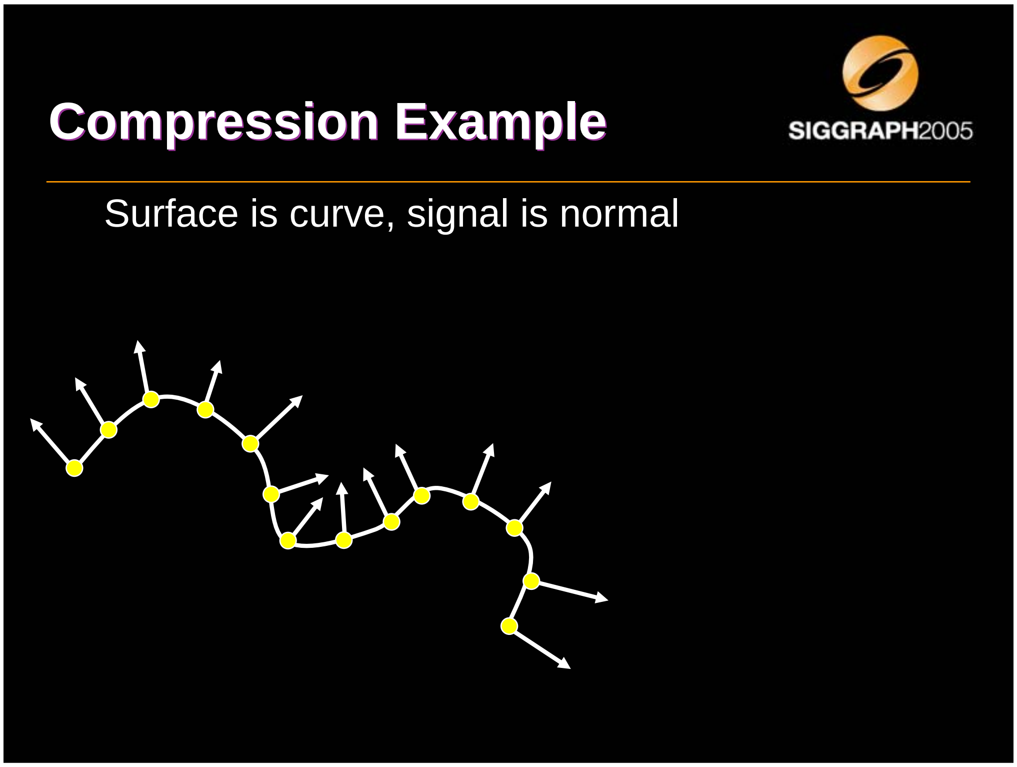
- Decode efficiently
  - As much on the GPU as possible
  - Render compressed representation directly
- Increase rendering performance
  - Make non-diffuse case practical
- Reduce memory consumption
  - Not just on disk

Our chief goal for compression is efficient decoding.

As much of this work has to be done on the GPU as possible and ideally we would render directly from the compressed representation.

This could lead to increased rendering performance making the glossy case practical.

Also it’s important to reduce the run time memory requirements, not just the size of the datasets on disk.



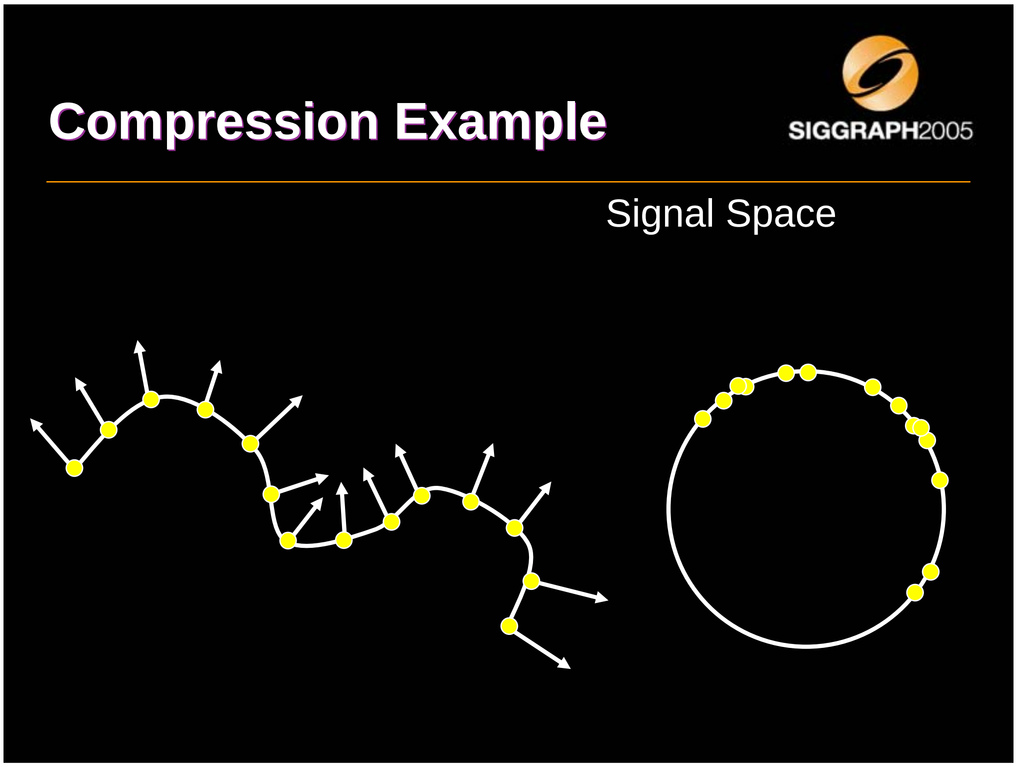
## Compression Example

SIGGRAPH2005

Surface is curve, signal is normal

These high dimensional signals are difficult to visualize, so I will use a much simpler problem to describe the compression techniques.

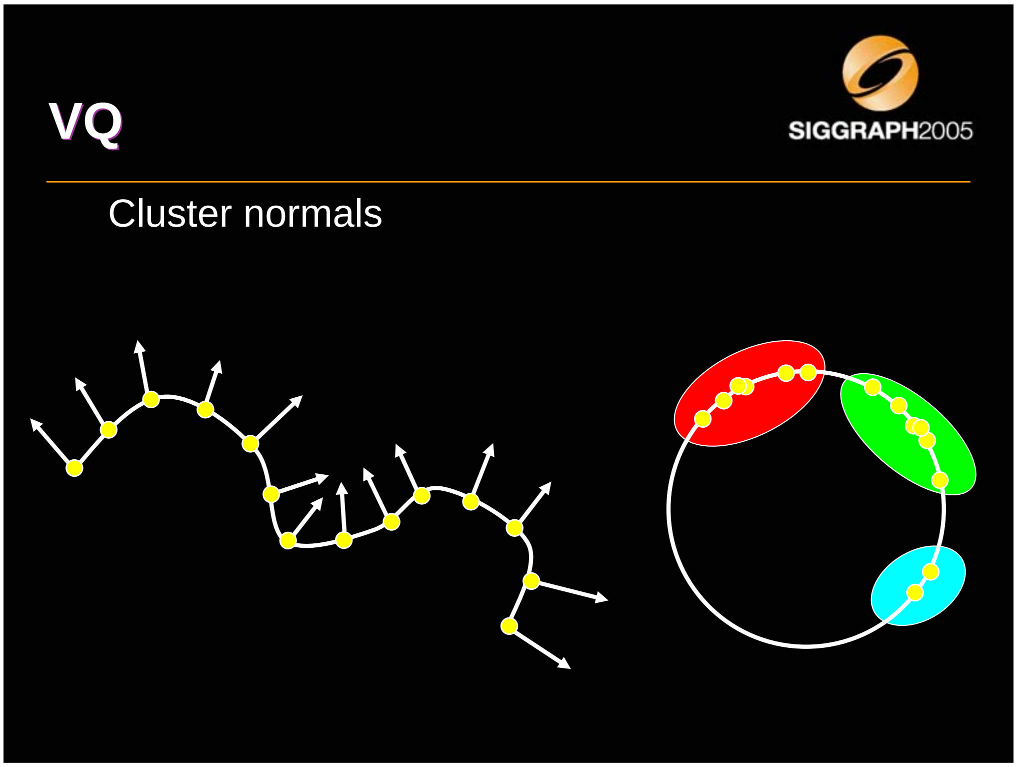
Our surface here is a curve, and the signal we are compressing is the normal at each point – a 2D signal.



## Compression Example

Signal Space

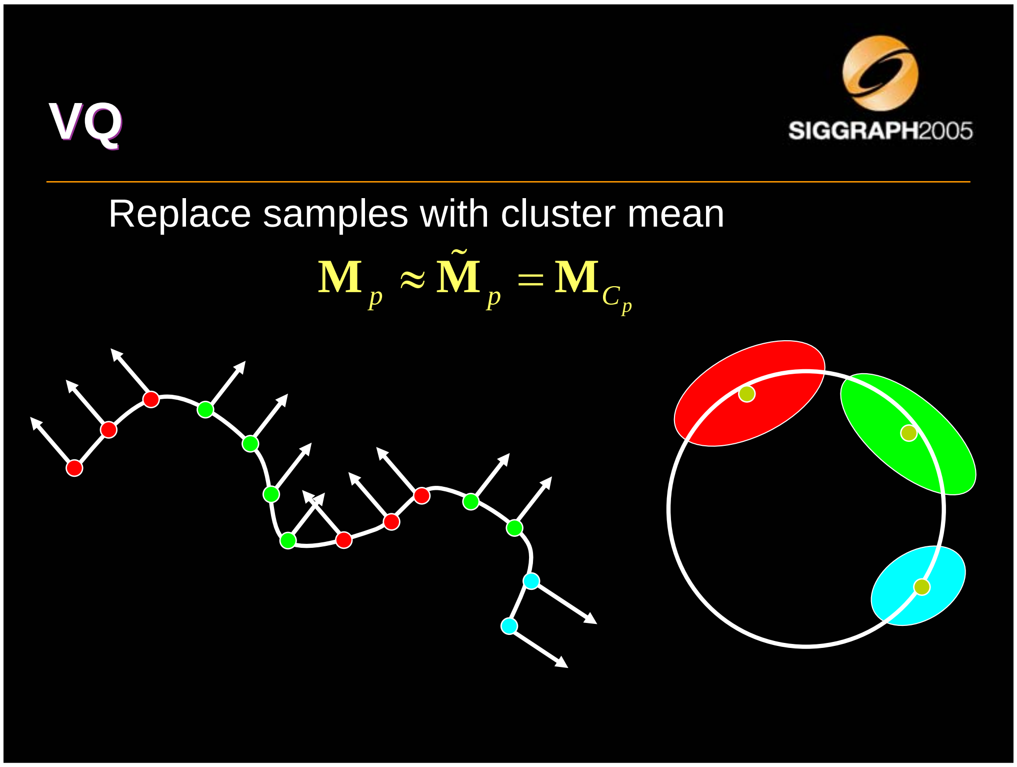
On the right, we visualize the signal space.  Each normal maps onto a point on the unit circle; this is the gauss map of the curve.



# VQ

Cluster normals

Vector quantization is a simple but common compression technique.  It groups the signal into a small number of clusters of similar samples, 3 in this example.



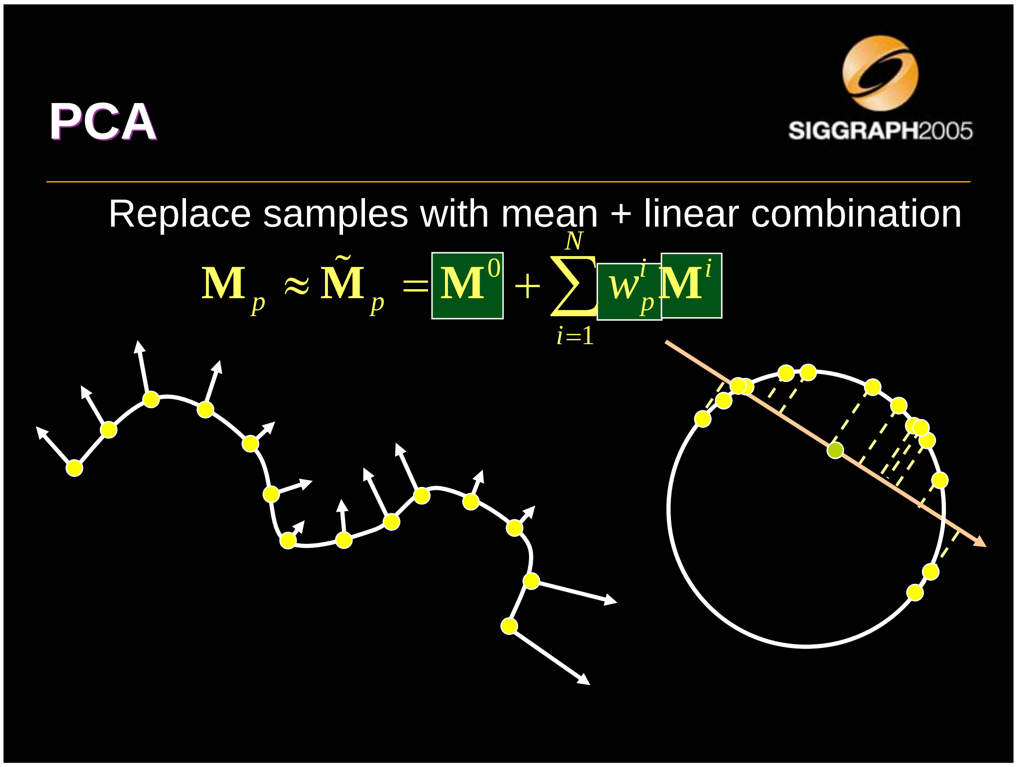
## VQ

Replace samples with cluster mean

$$\mathbf{M}_p \approx \tilde{\mathbf{M}}_p = \mathbf{M}_{C_p}$$

VQ approximates each sample as the mean of the cluster it’s a member of.

Note that the clustering is done completely in signal space; the spatial relationships of the samples is ignored.



## PCA

Replace samples with mean + linear combination

$$\mathbf{M}_p \approx \tilde{\mathbf{M}}_p = \boxed{\mathbf{M}^0} + \sum_{i=1}^{N} \boxed{w_p^i}\boxed{\mathbf{M}^i}$$

Principle component analysis is another common compression technique.

Given a signal in some high dimensional space, PCA computes an optimal lower dimensional linear approximation to the signal, in a least squares sense.

This is done by computing the mean of the signal, and a set of basis vectors.

In this case we are projecting the signal from 2D down to a single line.  Each sample is then replaced by its projection coefficients in this new basis.

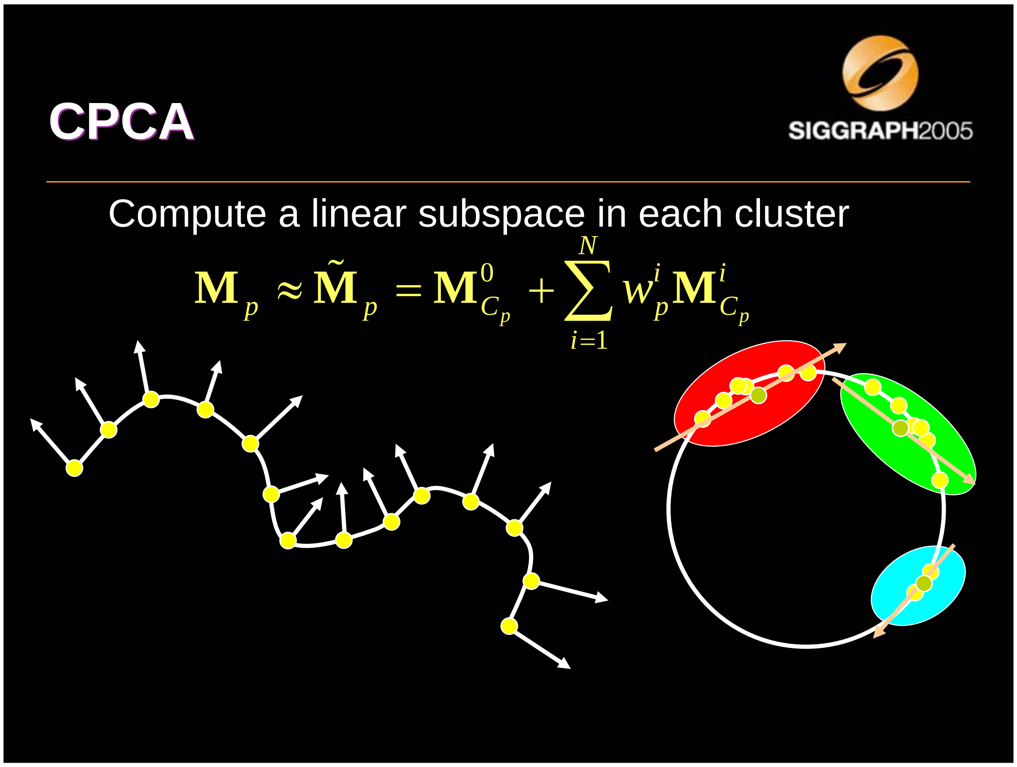
The chief limitation is evident from this example.

While the signal is a 1D manifold, it is clearly not globally linear.

But a linear approximation *is* appropriate locally – on the circle the tangent is a good local approximation to the signal.

Our signal is a high dimensional vector mapped to a surface in 2D.

If the signal is smooth, locally there should be a 2D tangent plane in 625D signal space at each point, so a local linear approximation should be appropriate.



## CPCA

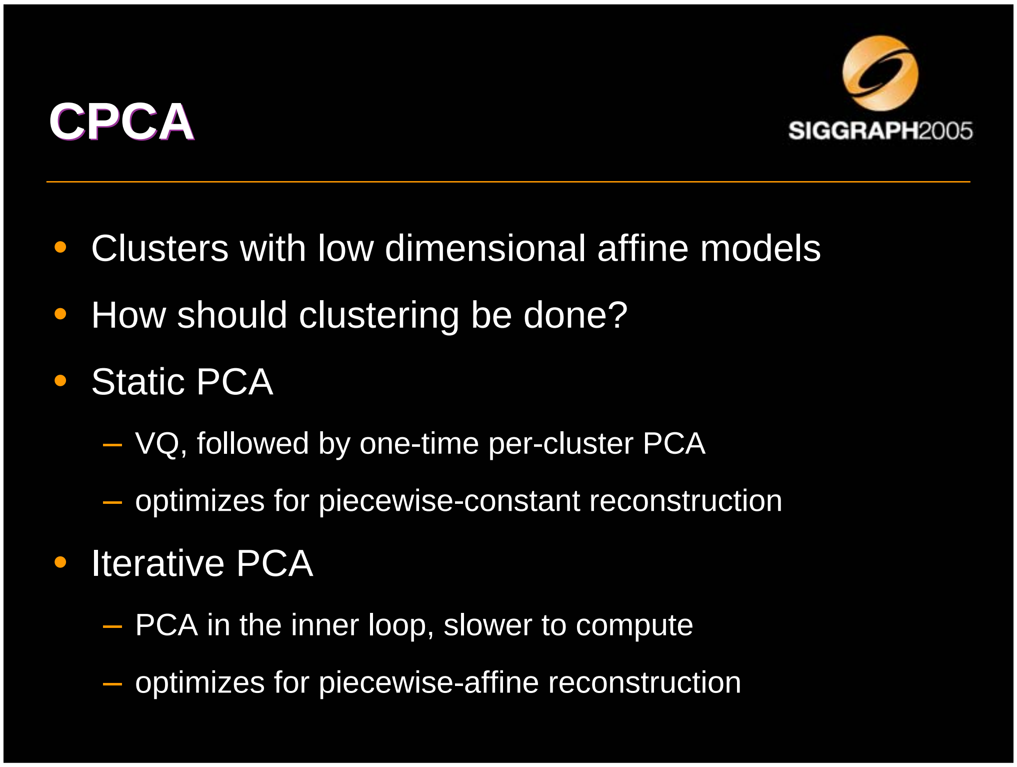
SIGGRAPH2005

Compute a linear subspace in each cluster

$$\mathbf{M}_p \approx \tilde{\mathbf{M}}_p = \mathbf{M}^0_{C_p} + \sum_{i=1}^{N} w^i_p \mathbf{M}^i_{C_p}$$

Our technique, called CPCA for “clustered principal analysis”, exploits this by computing a linear subspace in each cluster.

Each surface point needs to store an index into a cluster, and a set of projection weights for the PCA basis in the corresponding cluster.



## CPCA

SIGGRAPH2005

- Clusters with low dimensional affine models
- How should clustering be done?
- Static PCA
  - VQ, followed by one-time per-cluster PCA
  - optimizes for piecewise-constant reconstruction
- Iterative PCA
  - PCA in the inner loop, slower to compute
  - optimizes for piecewise-affine reconstruction

So CPCA approximates samples in each cluster using a low dimensional affine model.

There are a couple of ways to do the clustering.

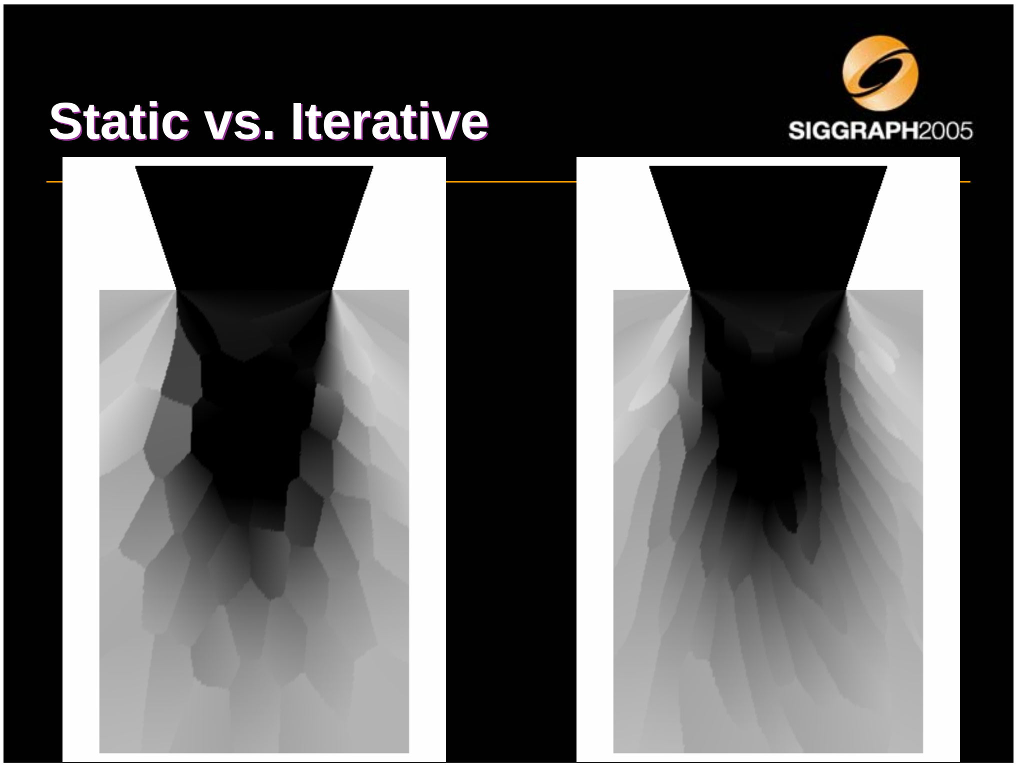
The most straightforward approach is just to cluster using VQ and then compute a PCA basis in each cluster, which we call “static PCA”.

VQ optimizes for piecewise constant reconstruction, while we use piecewise-affine reconstruction, so this is clearly suboptimal.

“Iterative PCA” clusters using distance to affine sub-space instead of distance to mean.

It computes PCA in the inner loop and so is slower than static PCA.

But it optimizes for the reconstruction we use during rendering.



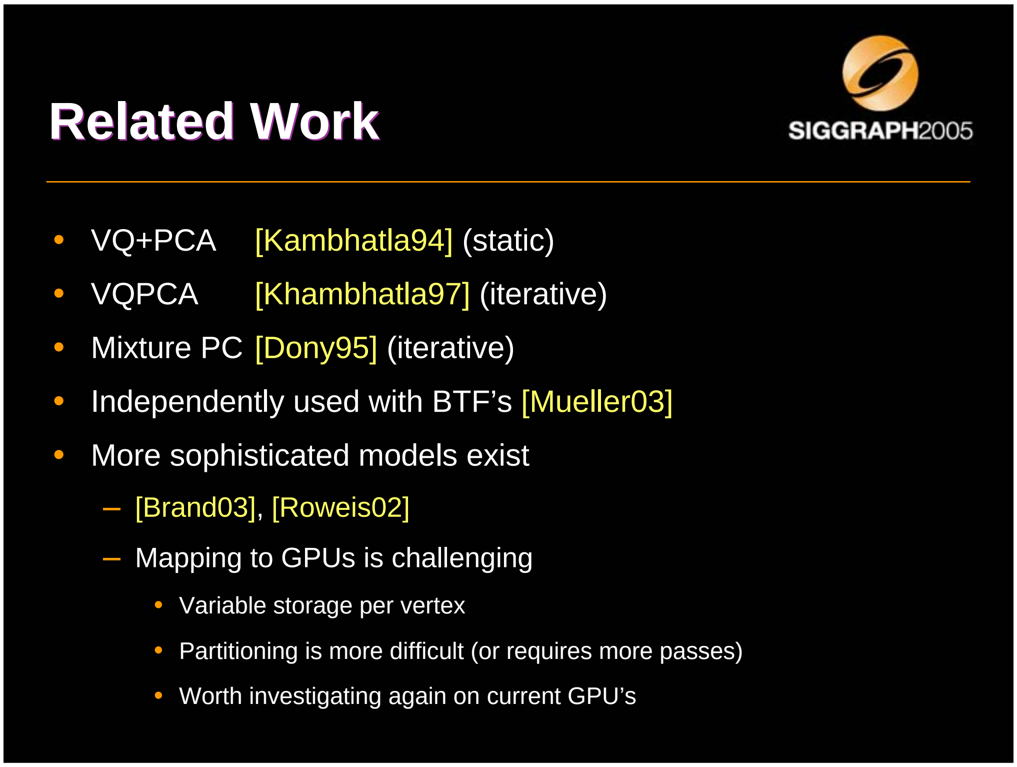
## Static vs. Iterative

SIGGRAPH2005

Here we see two images clustered with the different techniques using a single PCA vector.

When shading, using a single PCA vector means each cluster will have a linear gradient.

Static PCA creates clusters that are relatively isotropic, while iterative PCA shapes the clusters based on probable shadow gradient directions.



## Related Work

- VQ+PCA [Kambhatla94] (static)
- VQPCA [Khambhatla97] (iterative)
- Mixture PC [Dony95] (iterative)
- Independently used with BTF's [Mueller03]
- More sophisticated models exist
  - [Brand03], [Roweis02]
  - Mapping to GPUs is challenging
    - Variable storage per vertex
    - Partitioning is more difficult (or requires more passes)
    - Worth investigating again on current GPU's

While this is new to CG, it's been done in machine learning.

Kambhatla used the term VQ+PCA to define the static case, and VQPCA to define the iterative case.

Dony settled on the term Mixture of PC to define the iterative case.

Mueller and collegues used similar techniques in a paper at VMV in 2003.

There are more sophisticated techniques for non-linear dimensionality reduction.

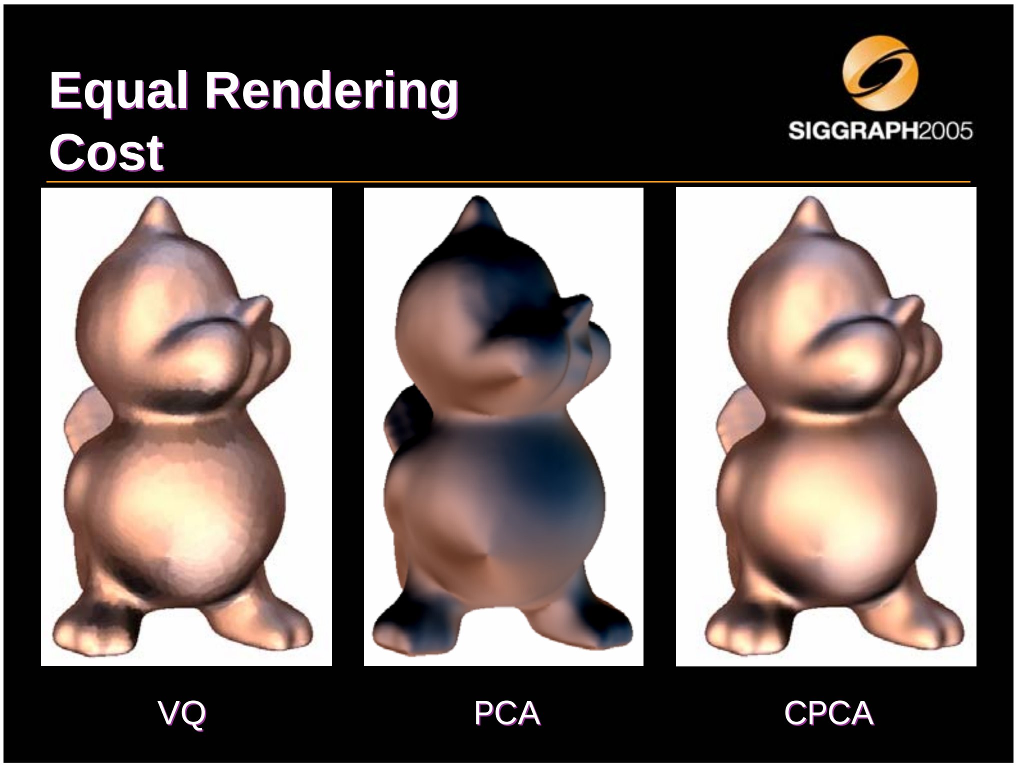
Matt Brand's work is used in a paper presented at siggraph in 2003 for example.

Mapping these techniques to the previous generation of GPU's is challenging.

Since samples are classified in a variable number of clusters, variable storage is required.

Partitioning the mesh into independent sets of data is also more difficult.

As features like dependent textures become more performant, it will be worth investigating these other algorithms as well.



## Equal Rendering Cost

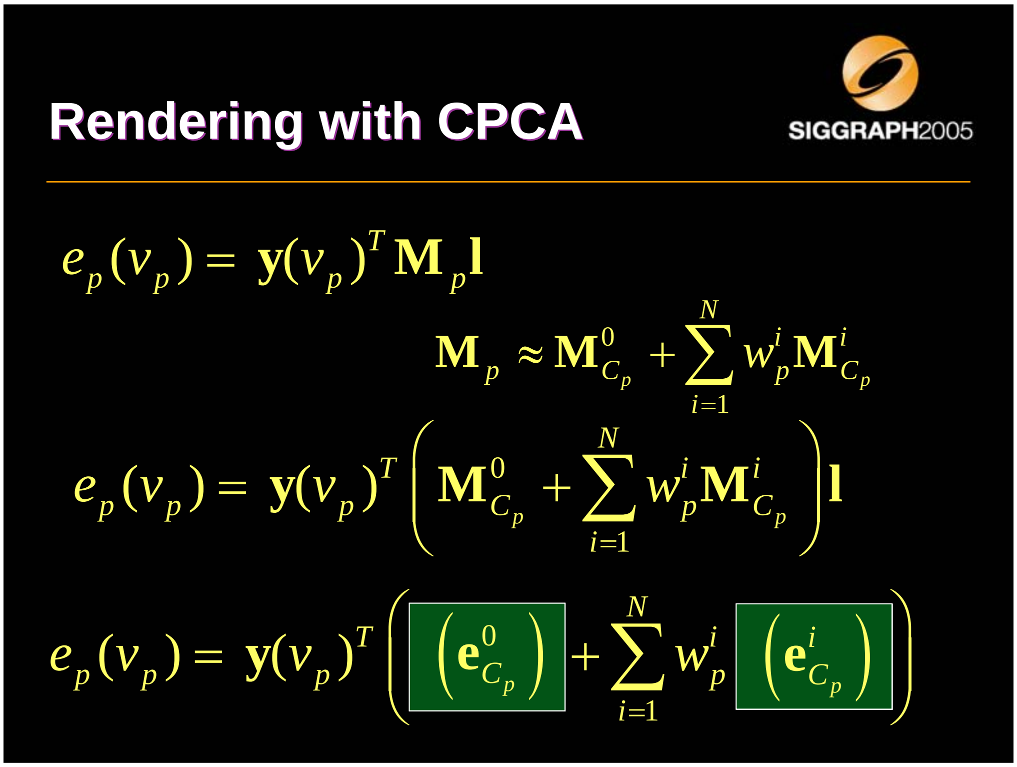
VQ PCA CPCA

Here we see quality obtained when a bird model is rendered at equal frame rates using 3 compression techniques.

With VQ, piecewise-constant reconstruction artifacts are apparent.

This signal is clearly not represented well with global PCA – there is substantial energy loss.

CPCA does a good job of approximating the signal, with fairly low error.



## Rendering with CPCA

SIGGRAPH2005

$$e_p(v_p) = \mathbf{y}(v_p)^T \mathbf{M}_p \mathbf{l}$$

$$\mathbf{M}_p \approx \mathbf{M}^0_{C_p} + \sum_{i=1}^{N} w^i_p \mathbf{M}^i_{C_p}$$

$$e_p(v_p) = \mathbf{y}(v_p)^T \left( \mathbf{M}^0_{C_p} + \sum_{i=1}^{N} w^i_p \mathbf{M}^i_{C_p} \right) \mathbf{l}$$

$$e_p(v_p) = \mathbf{y}(v_p)^T \left( \boxed{\left(\mathbf{e}^0_{C_p}\right)} + \sum_{i=1}^{N} w^i_p \boxed{\left(\mathbf{e}^i_{C_p}\right)} \right)$$

Rendering with CPCA is straight forward.

First, approximate each transfer matrix with the mean from its cluster and a linear combination of the cluster’s basis vectors.

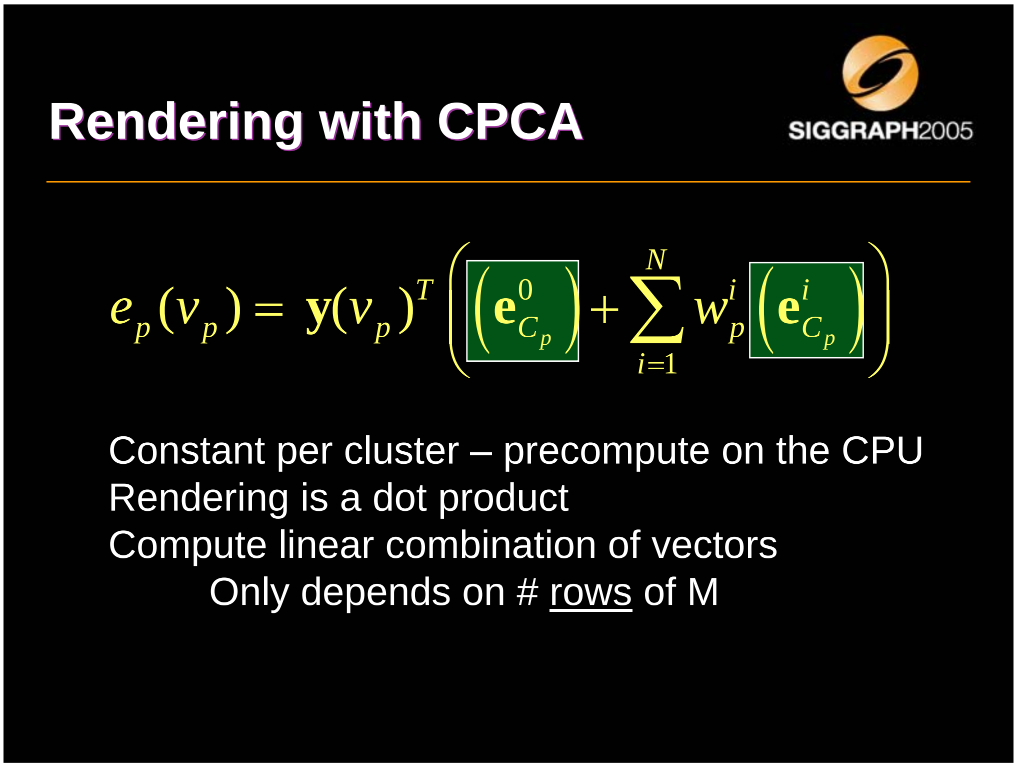
In this equation the Cp subscript represents the cluster for the corresponding point.

We can plug this directly into the rendering equation.

While this expression looks more complex then the original equation, we can distribute the dot produce with the light vector “l” and arrive at the following equation.

The highlighted expression are just exit radiance column vectors.

So scalar exit radiance is computed by taking a linear combination of exit radiance vectors and dotting it with the exit basis function evaluated in the view direction.



## Rendering with CPCA

SIGGRAPH2005

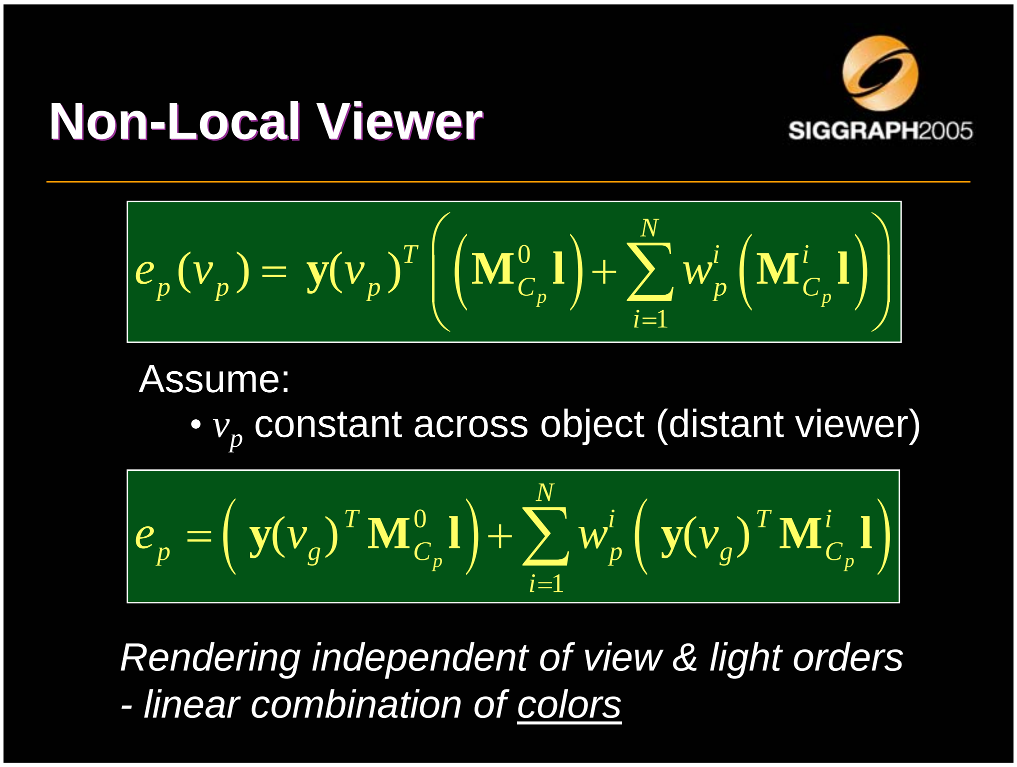
$$e_p(v_p) = \mathbf{y}(v_p)^T \left( \boxed{\left(\mathbf{e}^0_{C_p}\right)} + \sum_{i=1}^{N} w^i_p \boxed{\left(\mathbf{e}^i_{C_p}\right)} \right)$$

Constant per cluster – precompute on the CPU
Rendering is a dot product
Compute linear combination of vectors
Only depends on # rows of M

The important thing to note about this equation is that highlighted terms are constant per cluster, so they can be computed on the CPU at each frame.

The dimensionality of these vectors only depends on the number of exit radiance basis functions, so the math on the GPU is independent of the light's dimension.

For our glossy transfer matrices, this represents a computational savings if N is lower than 25.



## Non-Local Viewer

SIGGRAPH2005

$$e_p(v_p) = \mathbf{y}(v_p)^T \left( \left( \mathbf{M}_{C_p}^0 \mathbf{l} \right) + \sum_{i=1}^{N} w_p^i \left( \mathbf{M}_{C_p}^i \mathbf{l} \right) \right)$$

Assume:

- $v_p$ constant across object (distant viewer)

$$e_p = \left( \mathbf{y}(v_g)^T \mathbf{M}_{C_p}^0 \mathbf{l} \right) + \sum_{i=1}^{N} w_p^i \left( \mathbf{y}(v_g)^T \mathbf{M}_{C_p}^i \mathbf{l} \right)$$

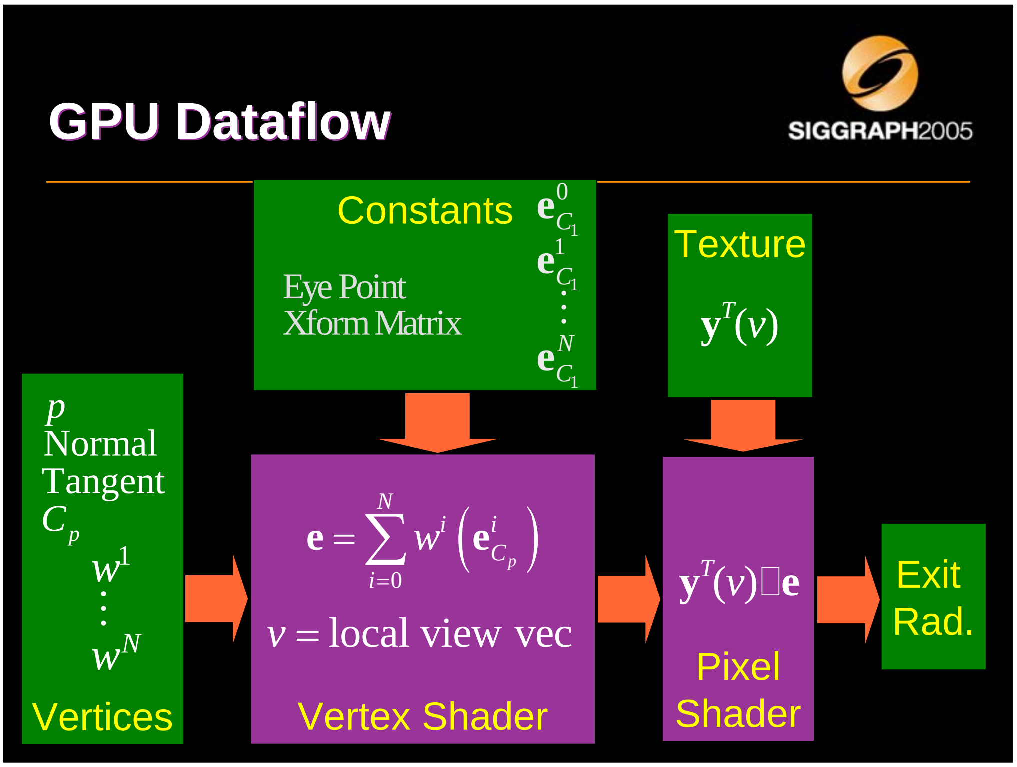
*Rendering independent of view & light orders*
*- linear combination of colors*

Using a non-local viewer model is a common technique in computer graphics.

The assumption is that the view does not change much across the object, so a common view direction can be used for all surface locations.

This allows us to distribute the constant view direction into the precomputed expressions, making the shader independent of both view and light orders.

All we need to render is a simple linear combination of N colors.



## GPU Dataflow

SIGGRAPH2005

Constants $\mathbf{e}^0_{C_1}$ $\mathbf{e}^1_{C_1}$ ⋮ $\mathbf{e}^N_{C_1}$

Eye Point
Xform Matrix

Texture

$\mathbf{y}^T(v)$

$p$
Normal
Tangent
$C_p$
$w^1$
⋮
$w^N$

Vertices

$$\mathbf{e} = \sum_{i=0}^{N} w^i \left(\mathbf{e}^i_{C_p}\right)$$

$v = \text{local view vec}$

Vertex Shader

$\mathbf{y}^T(v)\square\mathbf{e}$

Pixel Shader

Exit Rad.

Here's the GPU dataflow diagram. A static vertex buffer is computed which contains the position, local frame, and cluster index.

Each vertex's data also includes a binary value, called "incluster", which is 1 if the vertex is in the current cluster/supercluster and zero if it is a support vertex outside the cluster.

The projection coefficients are also stored at each vertex.

At every frame, constants in the vertex shader are loaded.

These constants include the eye point in object space and the transformation matrix. Also the per-cluster exit radiance vectors are computed on the CPU and loaded as well.

The vertex shader then computes the view vector in the local frame, and the exit radiance vector "e" passing them to the pixel shader.

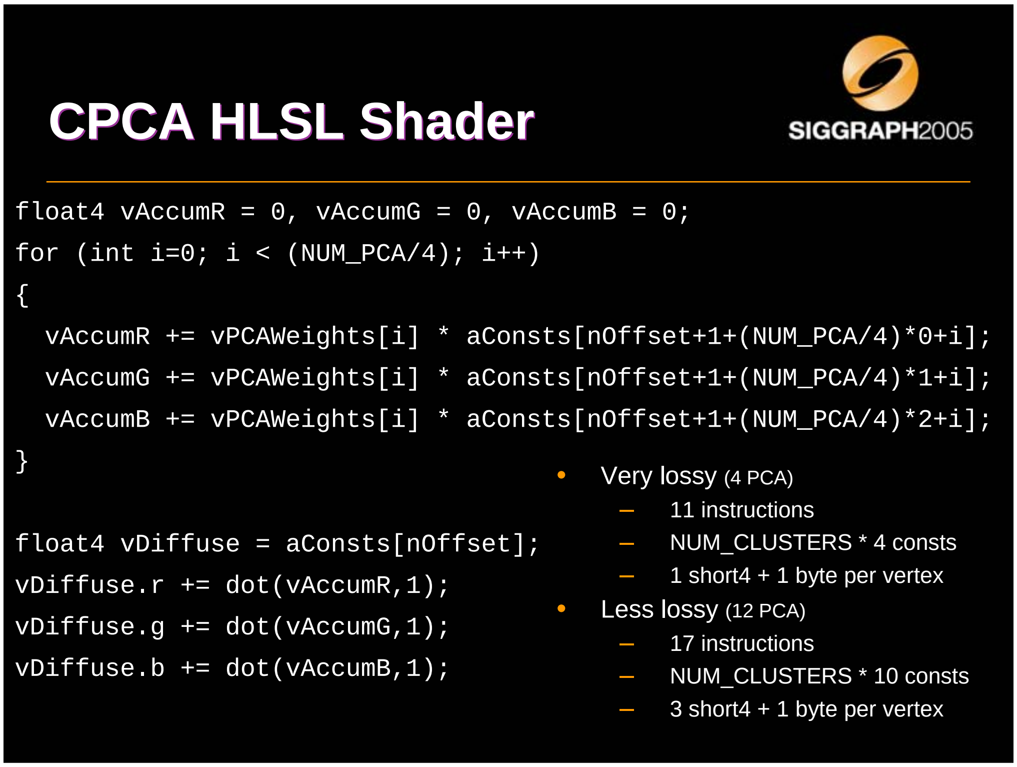
The pixel shader evaluates the exit basis functions, which we store as a texture, in the interpolated view direction.

It dots that with the interpolated exit radiance vector "e" to compute scalar exit radiance. This is done once each for R/G/B.

demo

Compression Demo

[walk through various numbers of PCA vectors, show difference between local/non-local viewer, etc.]



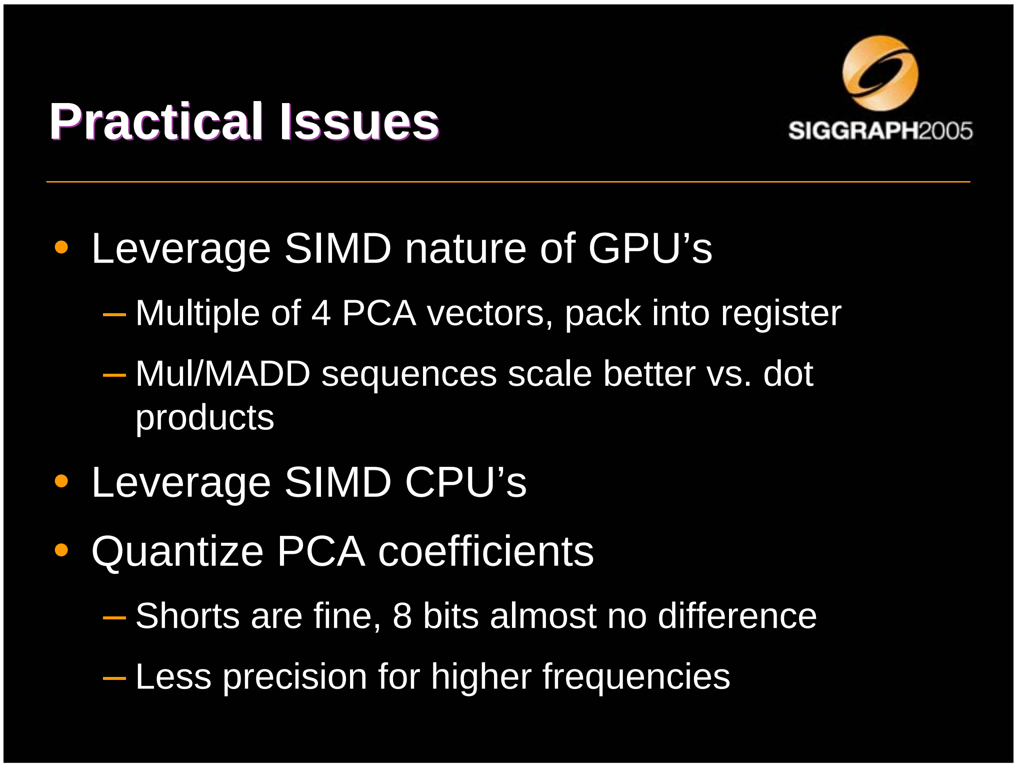
## CPCA HLSL Shader

SIGGRAPH2005

```
float4 vAccumR = 0, vAccumG = 0, vAccumB = 0;
for (int i=0; i < (NUM_PCA/4); i++)
{
  vAccumR += vPCAWeights[i] * aConsts[nOffset+1+(NUM_PCA/4)*0+i];
  vAccumG += vPCAWeights[i] * aConsts[nOffset+1+(NUM_PCA/4)*1+i];
  vAccumB += vPCAWeights[i] * aConsts[nOffset+1+(NUM_PCA/4)*2+i];
}

float4 vDiffuse = aConsts[nOffset];
vDiffuse.r += dot(vAccumR,1);
vDiffuse.g += dot(vAccumG,1);
vDiffuse.b += dot(vAccumB,1);
```

- Very lossy (4 PCA)
  - 11 instructions
  - NUM_CLUSTERS * 4 consts
  - 1 short4 + 1 byte per vertex
- Less lossy (12 PCA)
  - 17 instructions
  - NUM_CLUSTERS * 10 consts
  - 3 short4 + 1 byte per vertex

Here is the HLSL shader for diffuse rendering, the exact details aren’t that important, but depending on the number of clusters/PCA vectors you use, the shader is quite manegable.



## Practical Issues

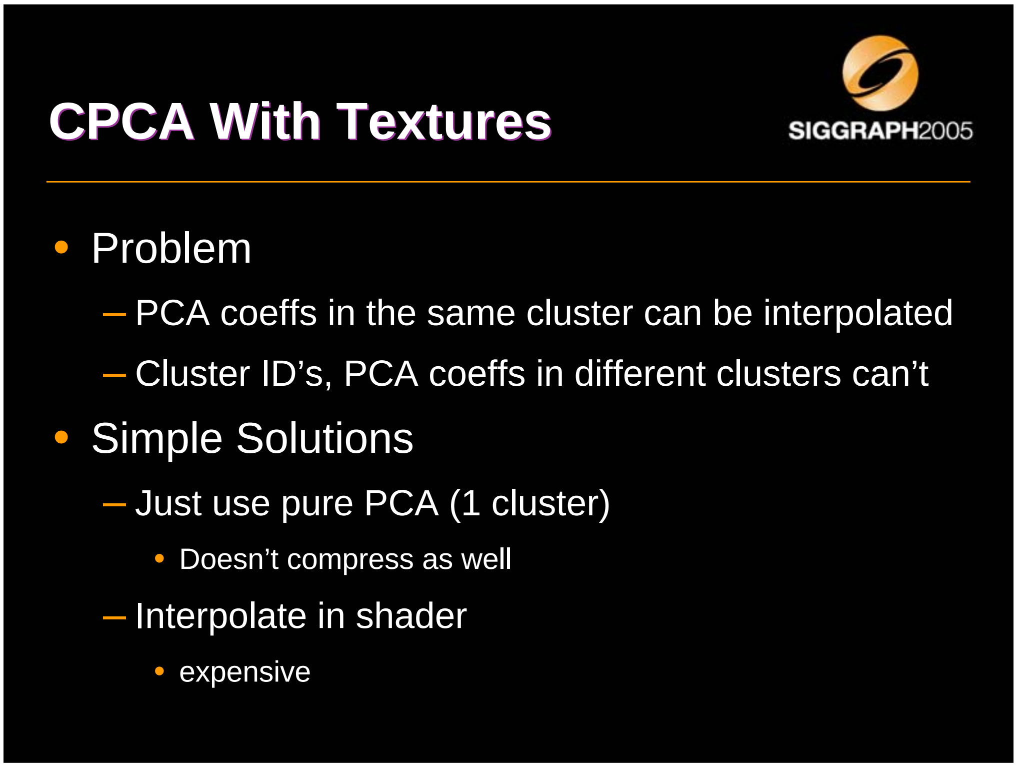
SIGGRAPH2005

- Leverage SIMD nature of GPU's
  - Multiple of 4 PCA vectors, pack into register
  - Mul/MADD sequences scale better vs. dot products
- Leverage SIMD CPU's
- Quantize PCA coefficients
  - Shorts are fine, 8 bits almost no difference
  - Less precision for higher frequencies

When writing shaders for CPCA, you want to leverage the SIMD nature of GPU's, always use a multiple of 4 PCA vectors, and split 4 coefficients for the same color in a register, instead of treating them as RGB colors to maximize the 4-way SIMD nature. When accumulating coefficients, it is more efficient to use a mul, and a string of mad's instead of a bunch of 4D dot products – because it generalizes to higher dimensions much better.

It's also worth leveraging the SIMD nature of CPU's, the dot products that compute shader constants can be mapped into SIMD CPU instructions quite easily.

When quantizing PCA coefficients, full 32 bit values are not needed. It is easy to normalize the data into [-1,1], just scale the corresponding basis vector by the inverse – this pushes precision into the constant registers, which are always higher precision. Even signed 8 bit values look fine, but it is possible to use higher precision for the first couple of PCA values, since they contribute the most to the final result.



## CPCA With Textures

SIGGRAPH2005

- Problem
  - PCA coeffs in the same cluster can be interpolated
  - Cluster ID's, PCA coeffs in different clusters can't
- Simple Solutions
  - Just use pure PCA (1 cluster)
    - Doesn't compress as well
  - Interpolate in shader
    - expensive

If you want to use CPCA with textures, care must be taken.

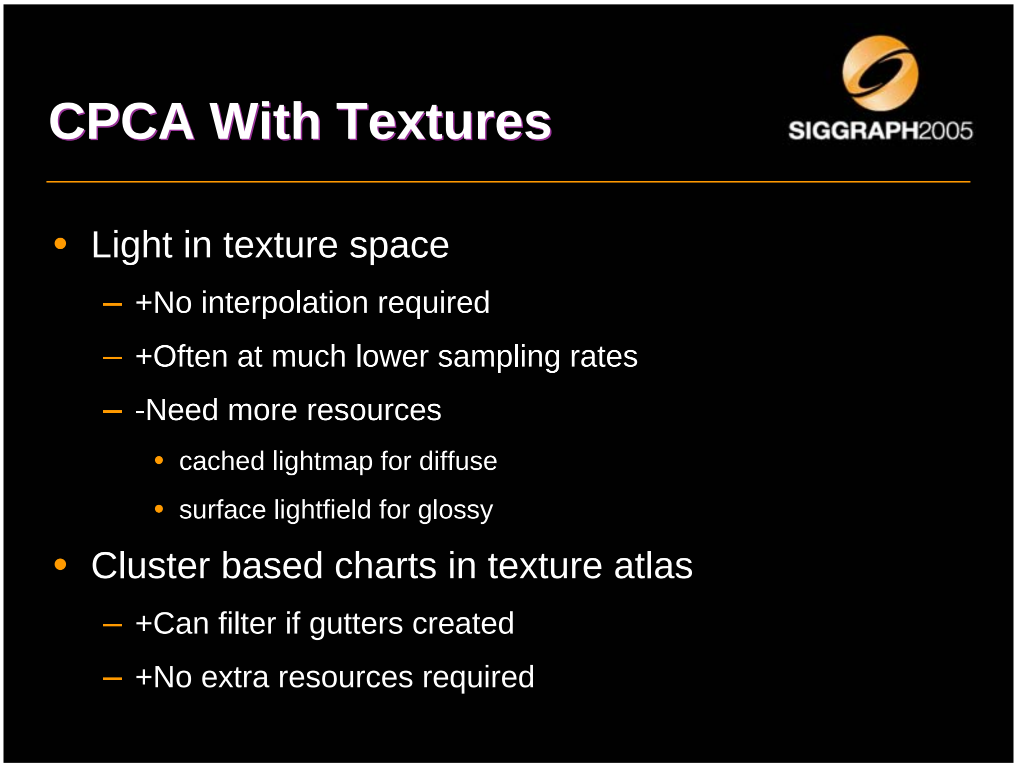
PCA coefficients in the same cluster can be interpolated, mip-mapped, etc.

However cluster ID's, are coefficients from different clusters should not be, so naïve bi-linear filtering will not work.

There are two simple solutions.

Just use pure PCA (ie: a single cluster), this filters fine, but it won't compress well.

You can do the interpolation in the shader – ie: evaluate the 4 taps and blend the shaded result.  This is extremely expensive though.



## CPCA With Textures

SIGGRAPH2005

- Light in texture space
  - +No interpolation required
  - +Often at much lower sampling rates
  - -Need more resources
    - cached lightmap for diffuse
    - surface lightfield for glossy
- Cluster based charts in texture atlas
  - +Can filter if gutters created
  - +No extra resources required

A better solution is to evaluate the shading directly in texture space.

No interpolation is required, because evaluation will only occur at texel centers.

PRT textures are often stored at much lower spatial sampling rates, so it can significantly save computation.

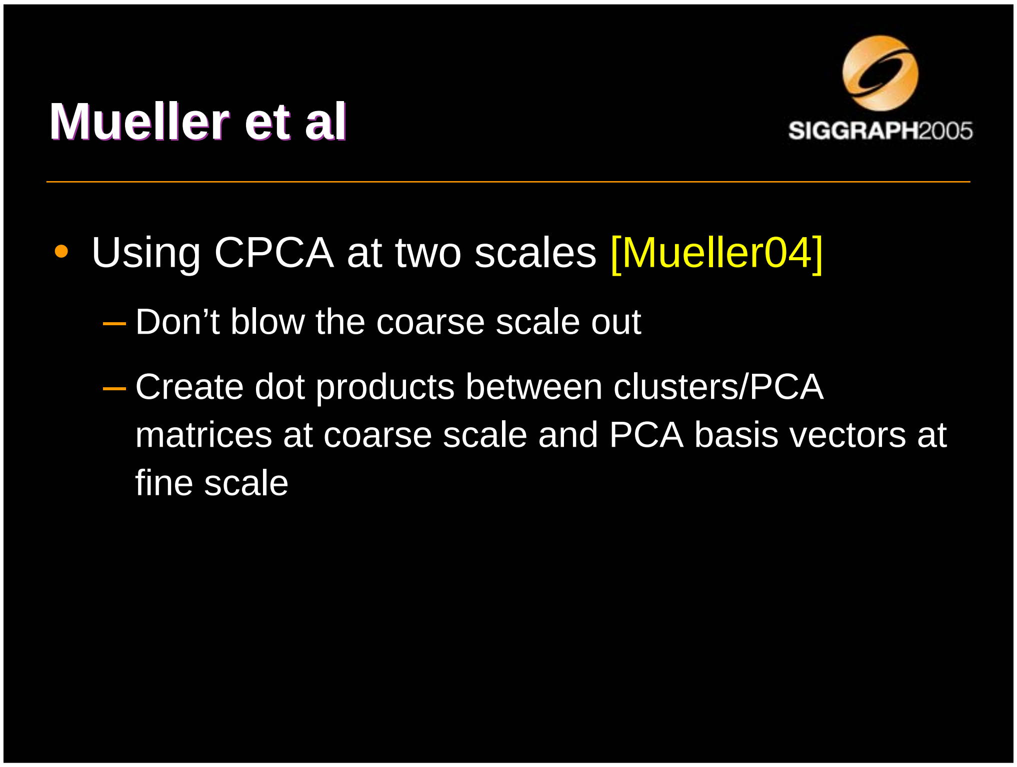
However more resourses are needed, in the diffuse case this effectively builds a dynamic light map, and in the glossy case a dynamic surface light field.

These can be updated at a lower frequency though – for example for a day/night cycle in a game.

Another approach is to cluster based on charts in a texture atlas

This guarantees that all pixels that will be reconstructed are in the same cluster (but charts can be in different clusters), and no temporary surfaces are required.

The compressed results might be sub-optimal however, and it might be interesting incorporating clustering for compression into the parameterization process.

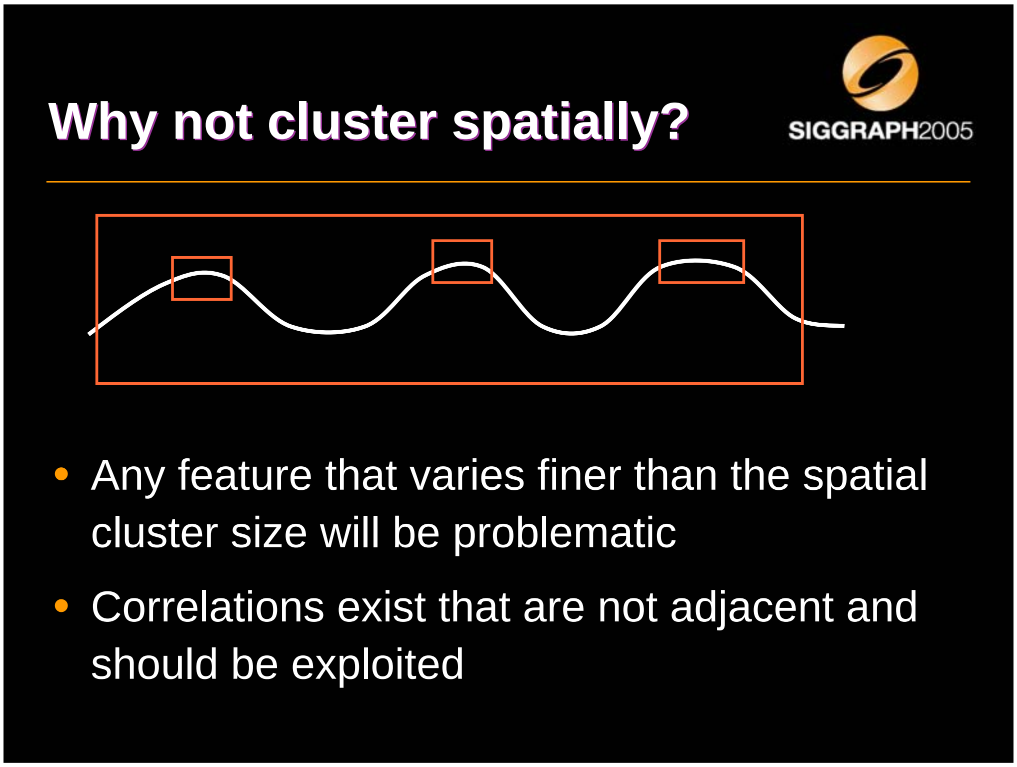

SIGGRAPH2005

## Mueller et al

- Using CPCA at two scales [Mueller04]
  - Don’t blow the coarse scale out
  - Create dot products between clusters/PCA matrices at coarse scale and PCA basis vectors at fine scale

There was another nice paper by Mueller and collegues where they used datasets that had information at two scales (coarse transfer matrices and fine scale BTF’s) and encoded both using local PCA. The naïve approach to doing this would be to blow out the SH signal in between the scales, and reprojet this spatially varying function at the finer scale – however this would be extremely slow.

They obvserved that you can directly project from coefficients on one CPCA space to coefficients in another by simply projecting all of the basis vectors for relighting the BTF’s through the basis matrices used for coarse scale transfer. These coefficients would then be updated when the lighting change and everything would remain performant.



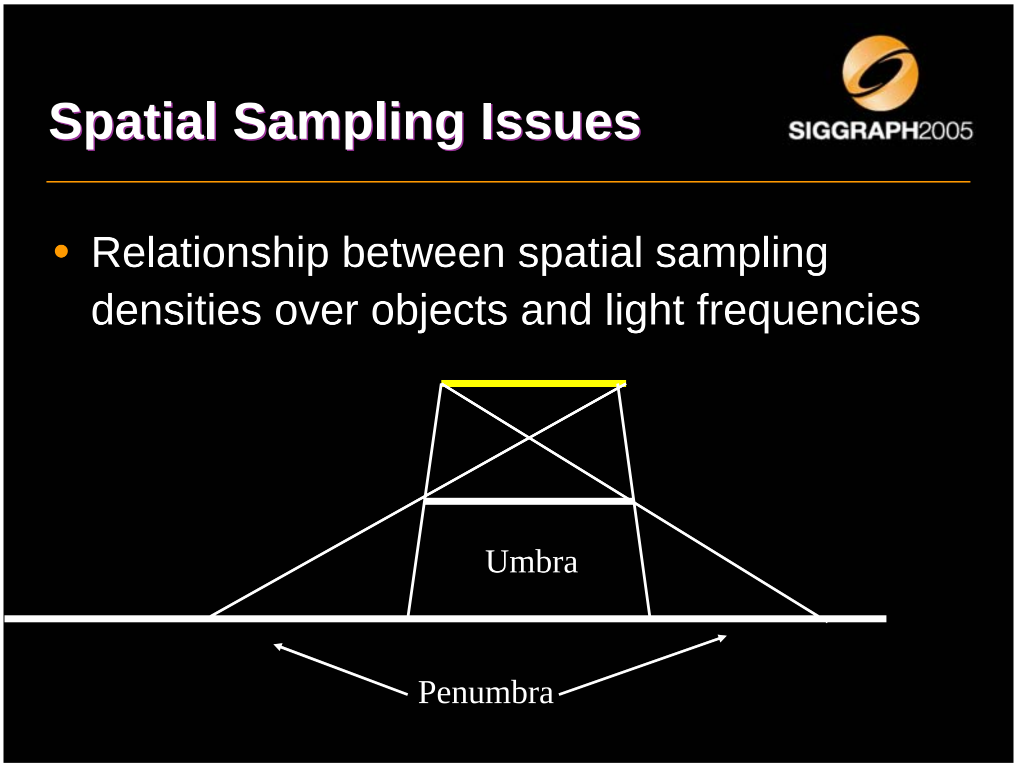
## Why not cluster spatially?

- Any feature that varies finer than the spatial cluster size will be problematic
- Correlations exist that are not adjacent and should be exploited

Instead of learning the clusters, you could just try and partition them spatially (this has been done in image re-lighting work.)

If the geometry varies quickly relative to the cluster size, it will require a higher dimensional linear space to approximate the data.

By clustering regions that are not spatially adjacent, for example the un-occluded region with a vertical normal highlighted here, you can get away with fewer clusters and lower dimensional linear spaces.



## Spatial Sampling Issues

SIGGRAPH2005

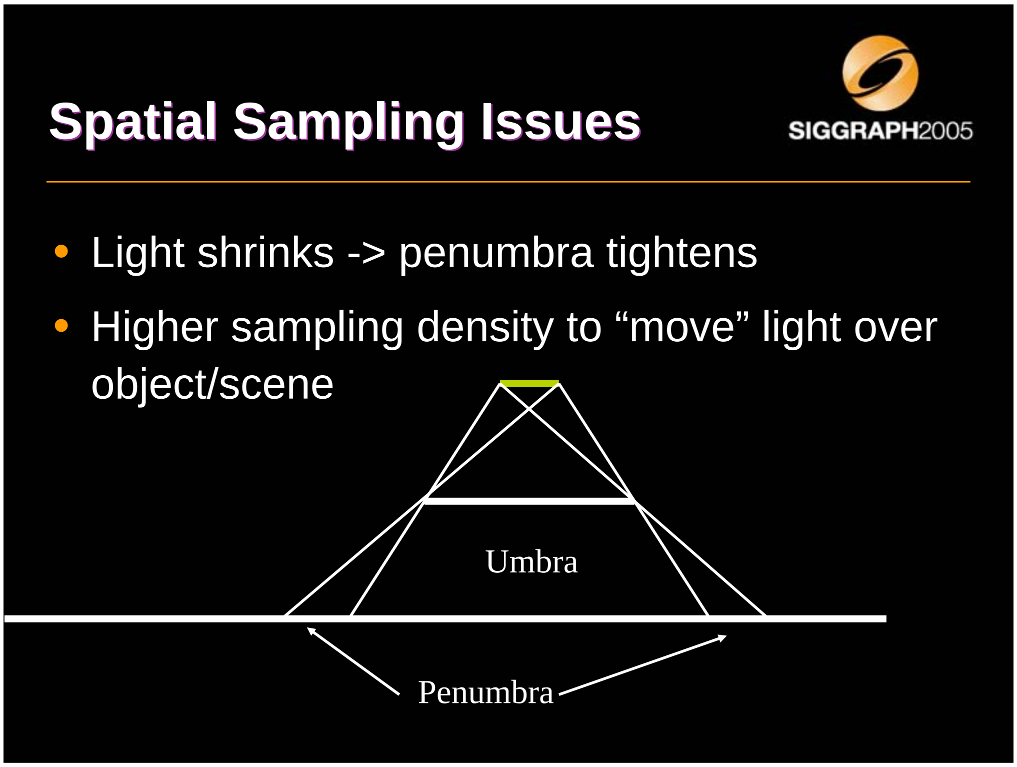
- Relationship between spatial sampling densities over objects and light frequencies

When looking at PRT in general, it is worth thinking about the relationship between light frequency and spatial sampling rates.

PRT generally deals with “steerable” lighting models, but it is worth thinking about the “finest” light that can be modeled when determining a bounds on spatial sampling densities.

Given a simple scene, with a large area light, a simple blocker and a simple receiver, the intensity on the receiver is broken into 3 categores.

Regions that are completely lit, regions that are completely in shadow (the umbra) and a transition region (the penumbra.) Large area lights have slow transitions, this means they induce lower spatial sampling rates.

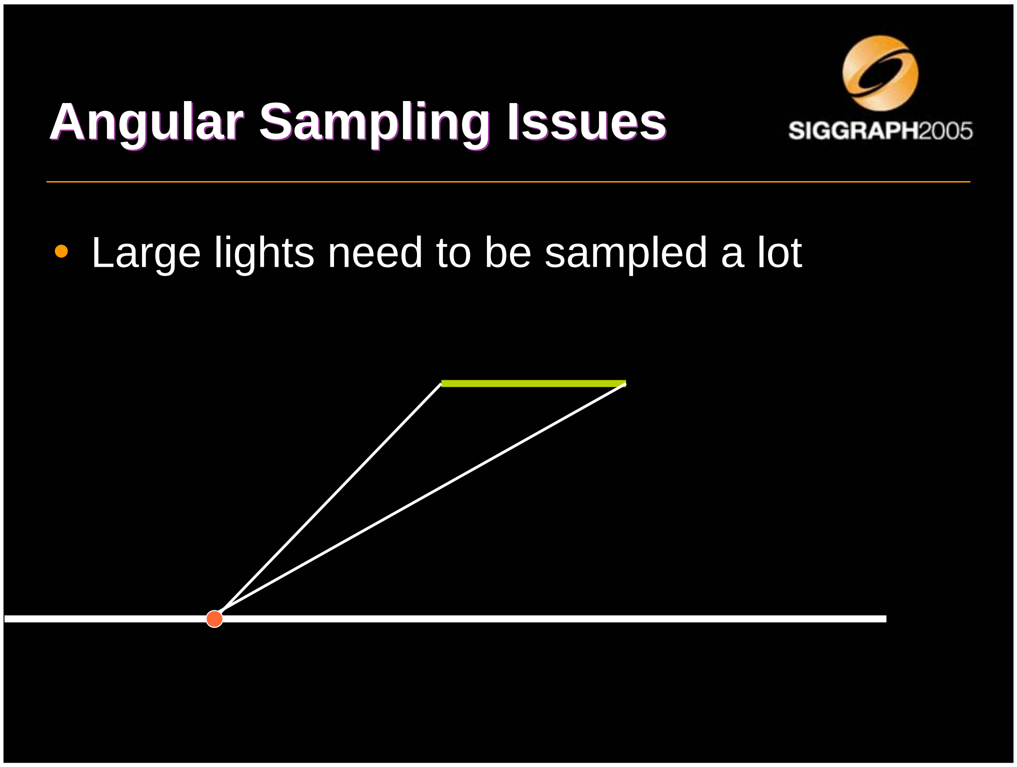


## Spatial Sampling Issues

SIGGRAPH2005

- Light shrinks -> penumbra tightens
- Higher sampling density to “move” light over object/scene

However for smaller light sources, the penumbras tighten, which means the transitions are more rapid, so the sampling rates have to be higher.



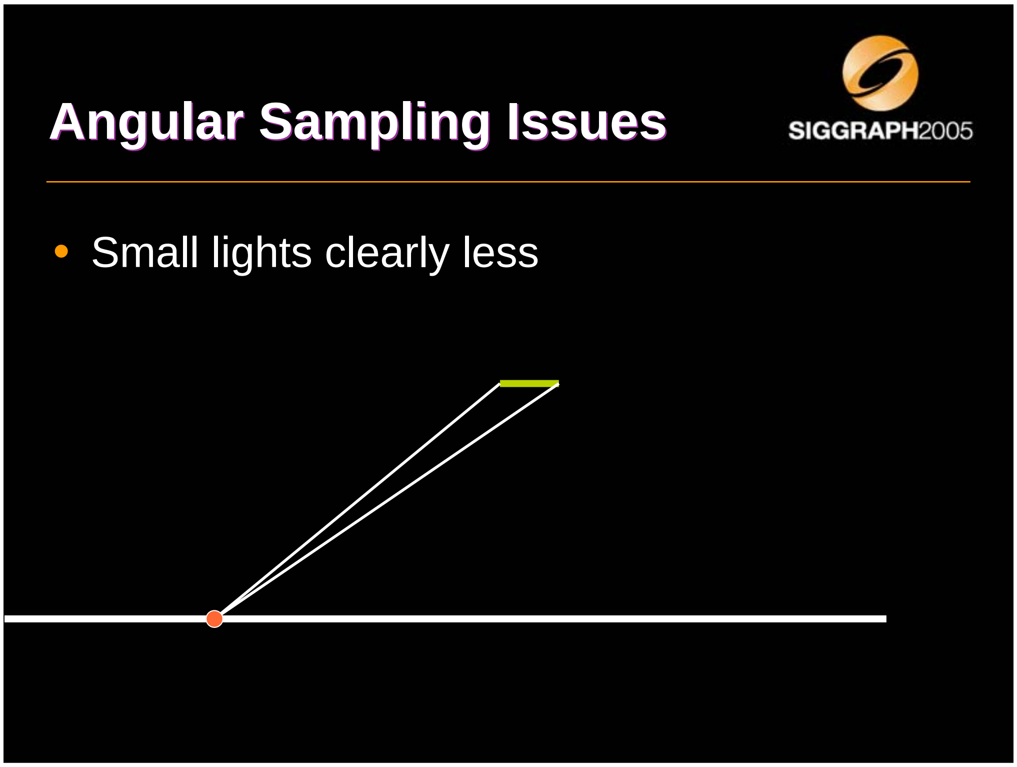
## Angular Sampling Issues

SIGGRAPH2005

- Large lights need to be sampled a lot

It is also worth thinking about angular, or directional sampling issues. Large/smooth lights subtend large solid angles, so to get an accurate estimate of the illumination arriving at a point, many directional samples have to be taken.

This means that traditional interactive techniques for shadowing (for example), would require many passes to generate an accurate model of the illumination.

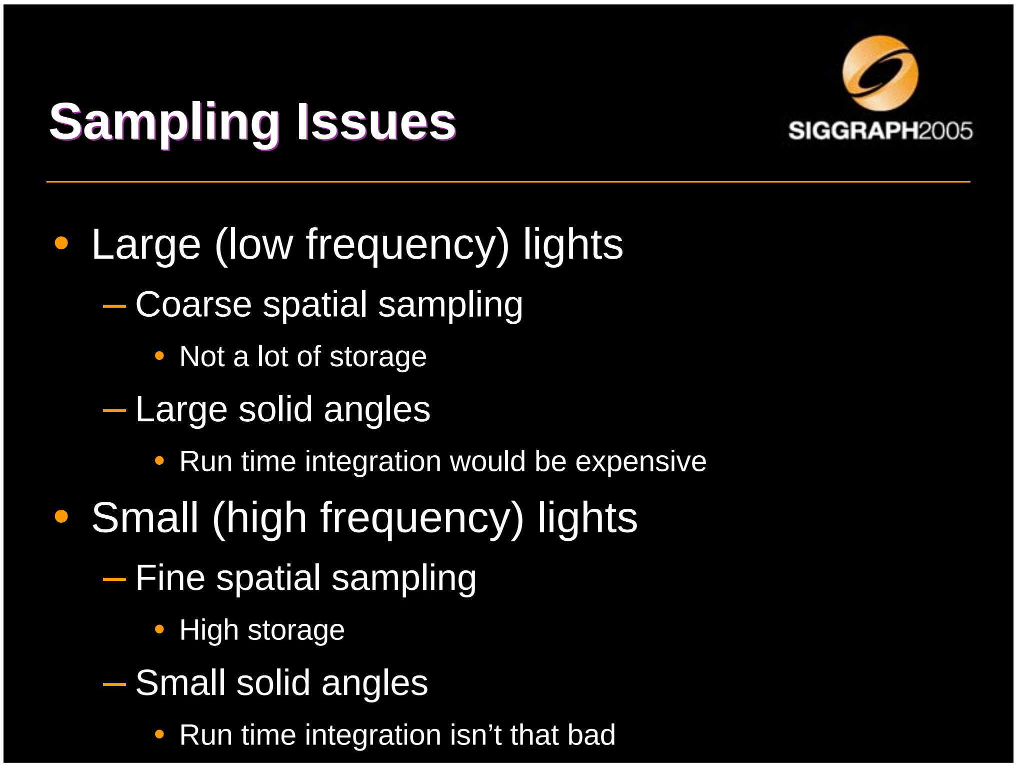


## Angular Sampling Issues

SIGGRAPH2005

- Small lights clearly less

Small light sources clearly require fewer samples, which means they are more feasible to work with at run time (assuming a small number of small light sources.)



## Sampling Issues

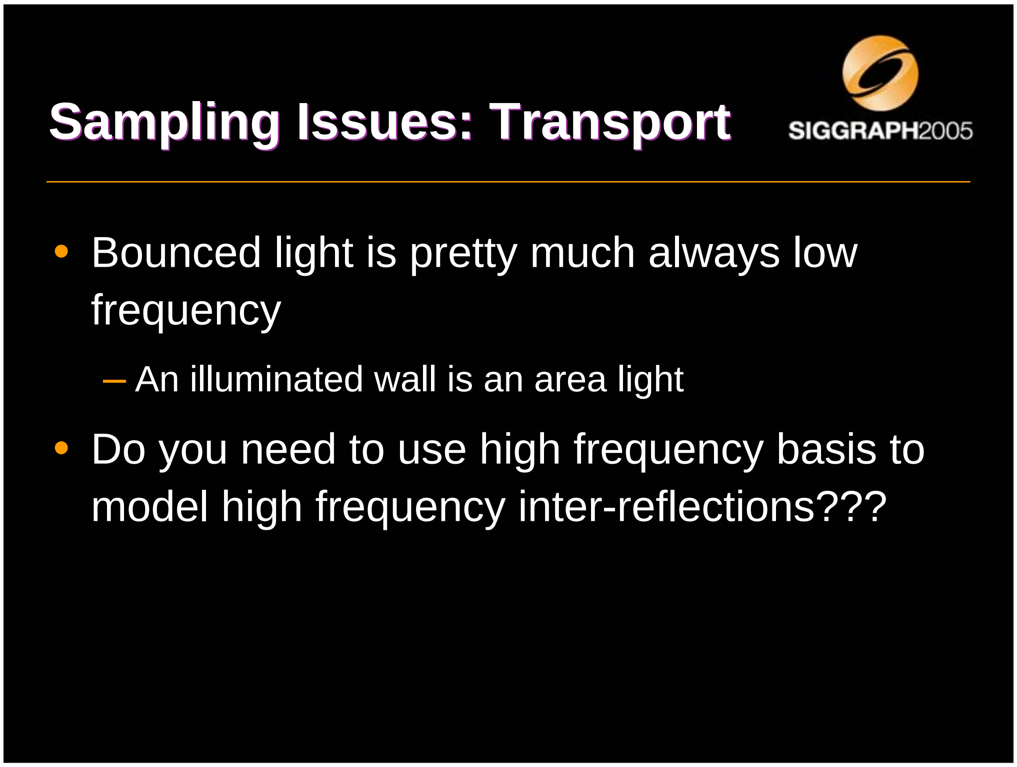
SIGGRAPH2005

- Large (low frequency) lights
  - Coarse spatial sampling
    - Not a lot of storage
  - Large solid angles
    - Run time integration would be expensive
- Small (high frequency) lights
  - Fine spatial sampling
    - High storage
  - Small solid angles
    - Run time integration isn’t that bad

So for direct lighting, large low frequency lights induce coarse spatial sampling rates – which means less storage is required and precomputing makes more sense.

They are difficult to handle using run time integration due to the large solid angles.

Small lights induce high spatial sampling densities, and are more amenable to run time integration.



## Sampling Issues: Transport

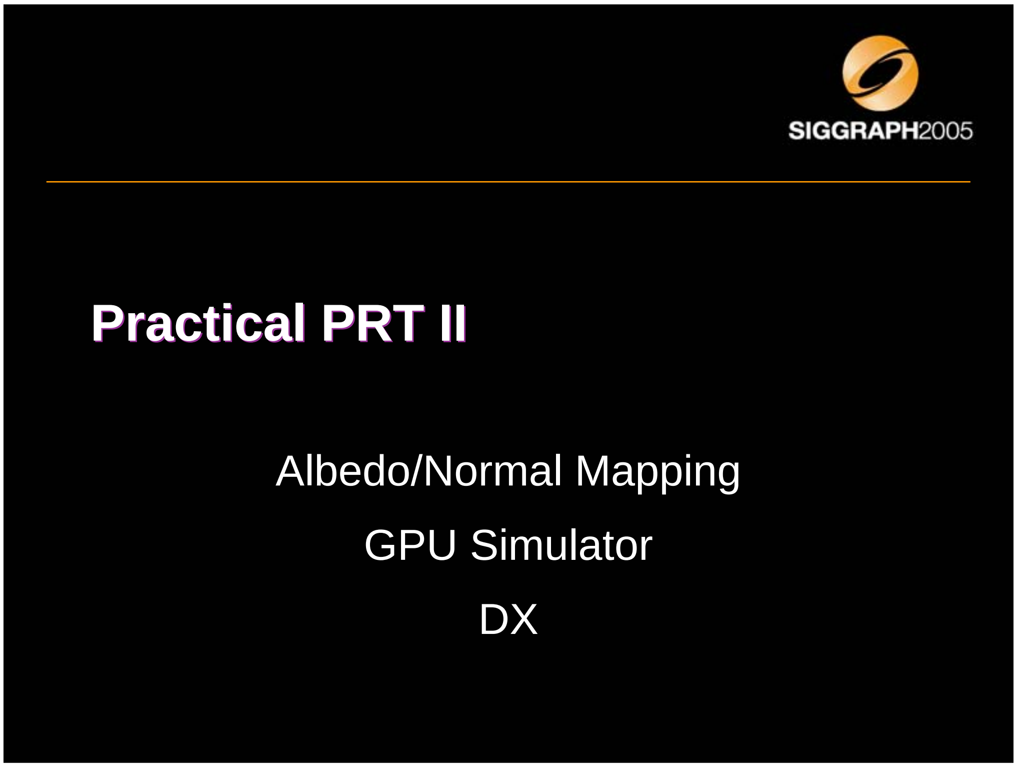
SIGGRAPH2005

- Bounced light is pretty much always low frequency
  - An illuminated wall is an area light
- Do you need to use high frequency basis to model high frequency inter-reflections???

When looking at more complex transport effects (beyond direct lighting) it is worth noting that they pretty much always behave as low frequency light sources – a wall illuminated by a tight light source generally acts as a large emitter.

So precomputing indirect lighting from high frequency lighting seems like a reasonable idea in general.

But that than begs the question, is it feasible to simulate the indirect lighting from high frequency lighting using a low frequency projection of that lighting?  This could be similar to the duality discussed by ravi in his siggraph 2001 paper between light and material frequency.

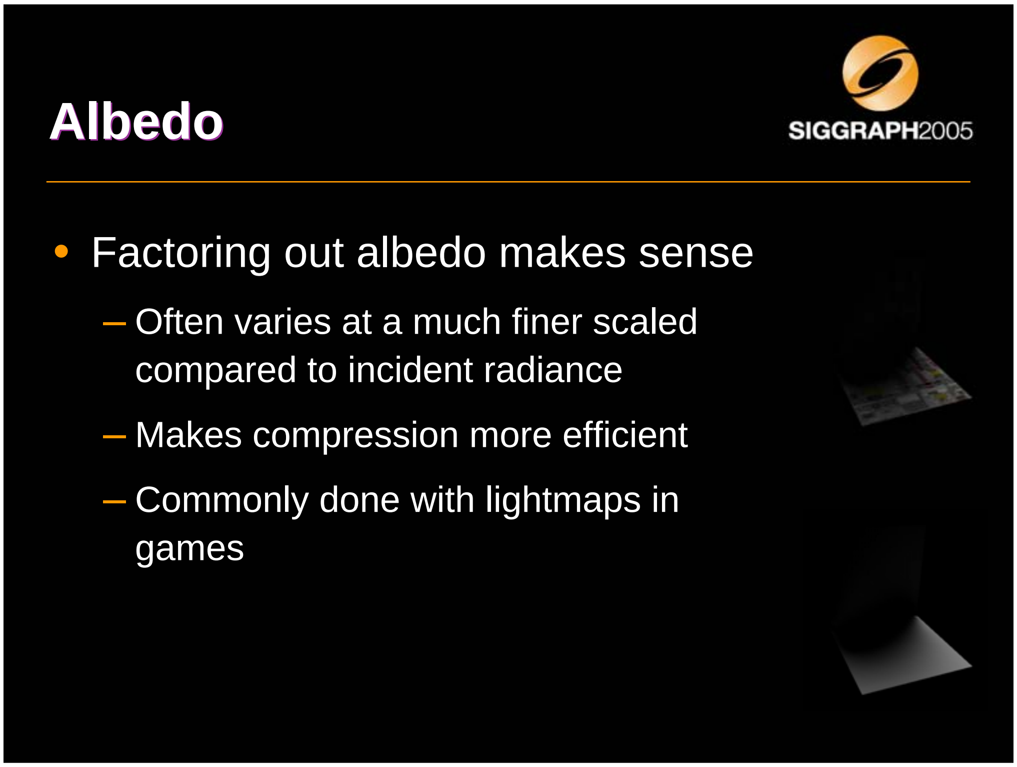


## Practical PRT II

Albedo/Normal Mapping

GPU Simulator

DX

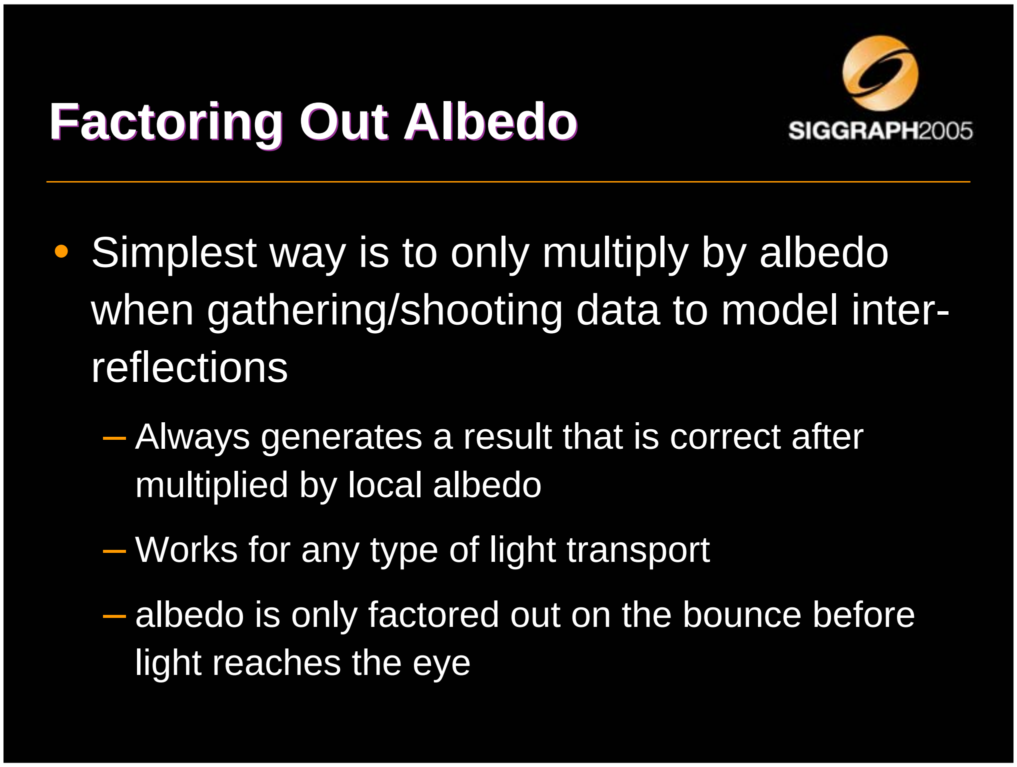


## Albedo

- Factoring out albedo makes sense
  - Often varies at a much finer scaled compared to incident radiance
  - Makes compression more efficient
  - Commonly done with lightmaps in games

The albedo should be factored out of the transfer vectors/matrices.

The incident lighting is almost always much smoother than reflectivity, so it makes sense to not conflate the two.



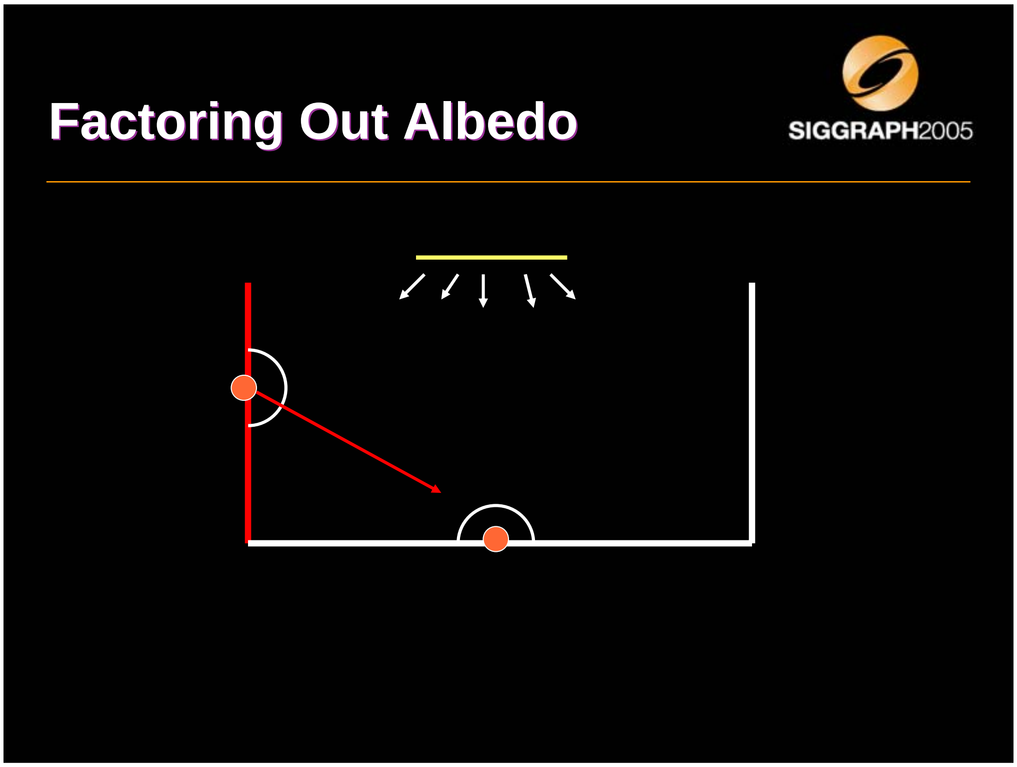
## Factoring Out Albedo

SIGGRAPH2005

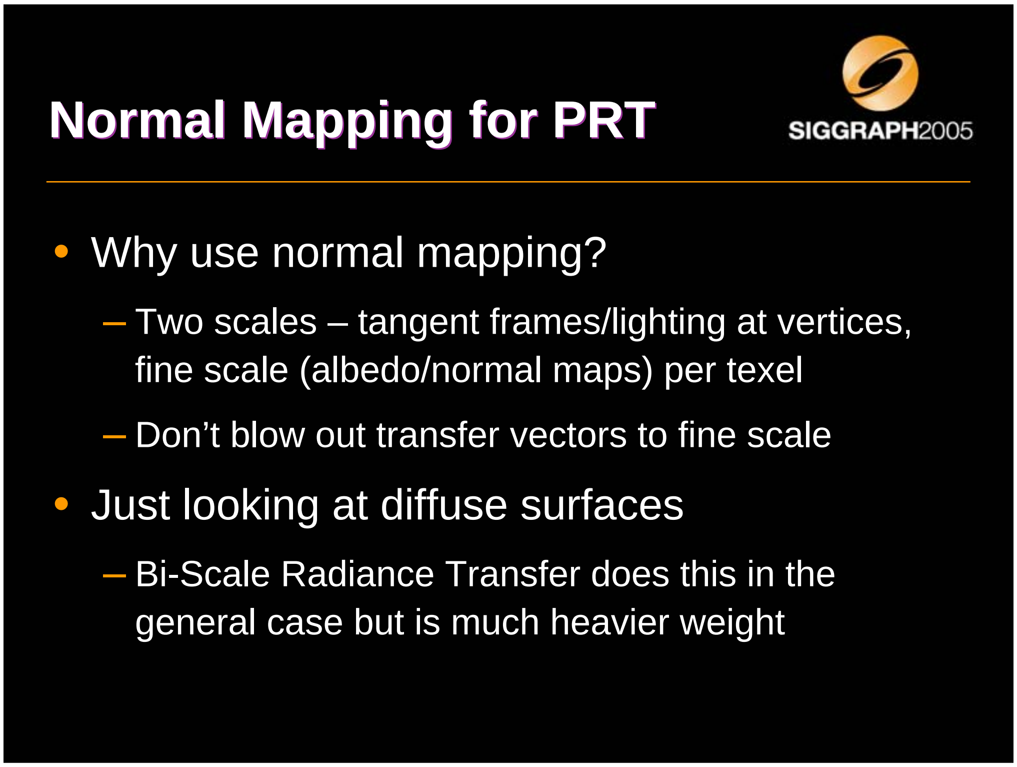
- Simplest way is to only multiply by albedo when gathering/shooting data to model inter-reflections
  - Always generates a result that is correct after multiplied by local albedo
  - Works for any type of light transport
  - albedo is only factored out on the bounce before light reaches the eye

The easiest way to do this is to only multiply by albedo when “gathering” (or before shooting) during transport simulation.

So all transfer vectors (for any transport path) model everythign except the multiply by the albedo at a point on the surface.



## Factoring Out Albedo



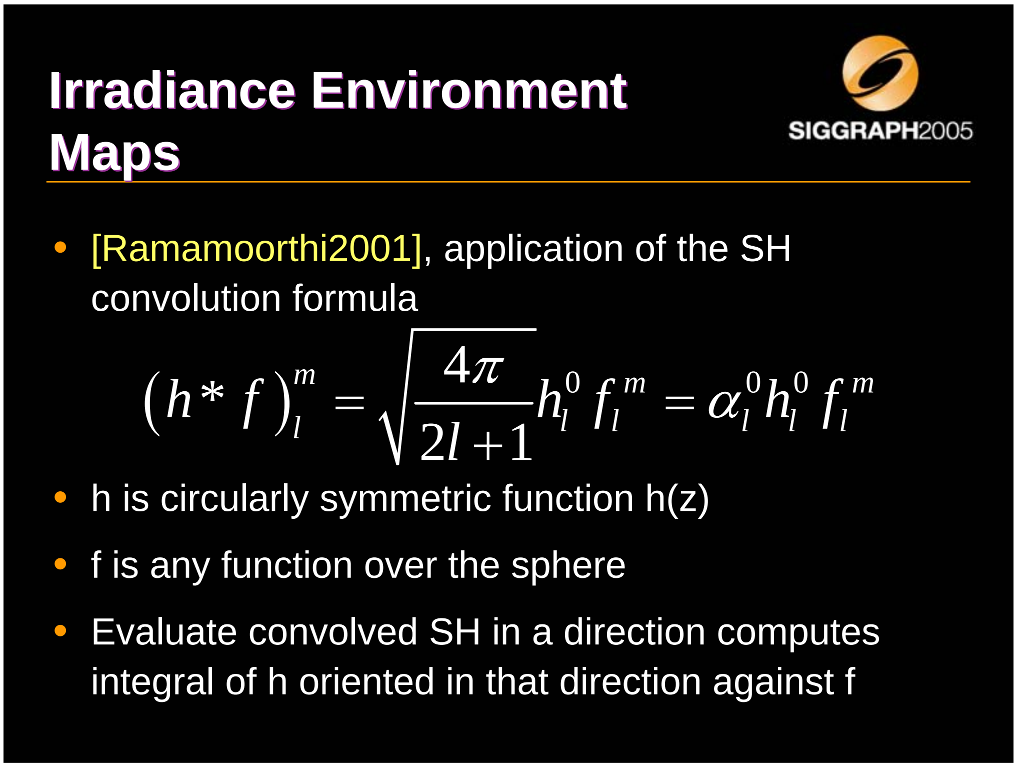
## Normal Mapping for PRT

SIGGRAPH2005

- Why use normal mapping?
  - Two scales – tangent frames/lighting at vertices, fine scale (albedo/normal maps) per texel
  - Don't blow out transfer vectors to fine scale
- Just looking at diffuse surfaces
  - Bi-Scale Radiance Transfer does this in the general case but is much heavier weight

One of the most beneficial properties of normal mapping is that it enables modeling things at multiple scales. Coarse information, either per vertex or in a lower resolution texture (like incident lighting) should be stored at the appropriate sampling rates. Normal maps model fine scale perturbations of the surface normal – but ideally you wouldn't blow out transfer vectors/matrices to that fine scale.

The technique we are about to describe is only for diffuse materials, the bi-scale radiance transfer paper covers the more general case, but is a much "heavier" technique (both in terms of computation and data…)



## Irradiance Environment Maps

- [Ramamoorthi2001], application of the SH convolution formula

$$\left(h*f\right)_l^m = \sqrt{\frac{4\pi}{2l+1}}\, h_l^0 f_l^m = \alpha_l^0 h_l^0 f_l^m$$

- h is circularly symmetric function h(z)
- f is any function over the sphere
- Evaluate convolved SH in a direction computes integral of h oriented in that direction against f

The normal mapping technique is based on the irradiance environment map paper from siggraph 2001.

This paper is essentially an application of the spherical harmonic convolution formula.

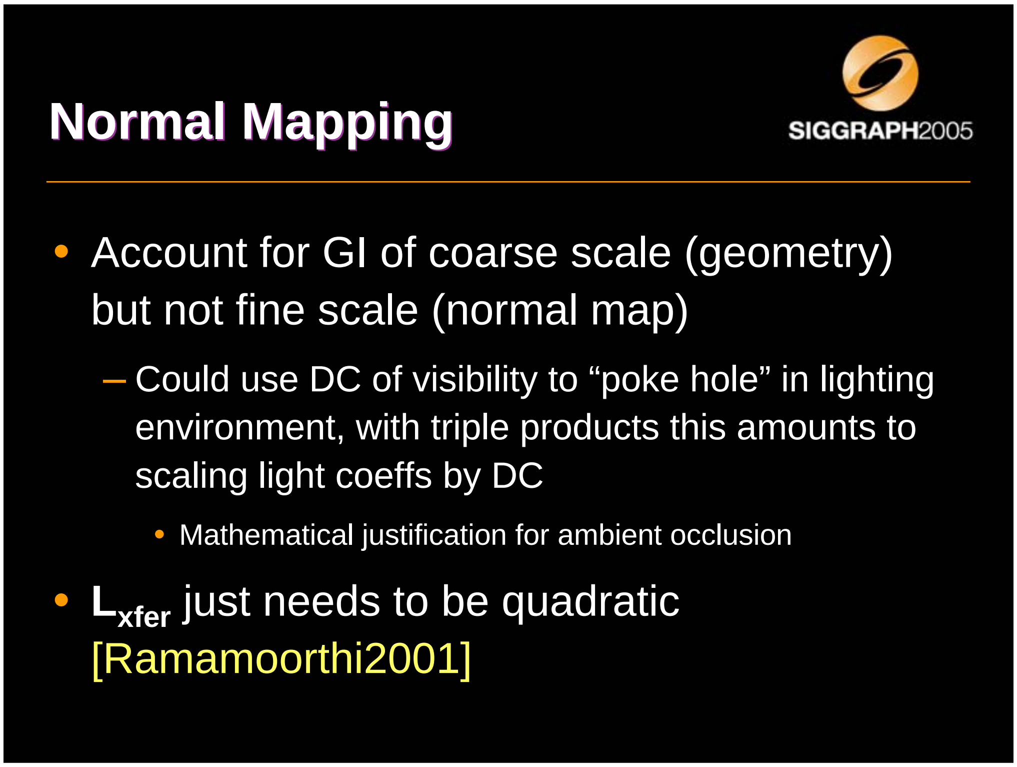
Which is simple the convolution of a circularly symmetric kernel h (can be thought of as a function of Z only when oriented with the Z axis) with an arbitrary function f represented in spherical harmonics can be represented by scaling each of the coefficients in a given band of f (l) by a constant and the coefficient of m=0 in h at that band.

Circular symmetric functions of z only have one non-zero projection coefficient (corresponding to m=0)

Evaluating a convolved function in a given direction gives you the integral of the original function against the kernel oriented in that direction.

The reason the kernel has to be circularly symmetric is because functions with circular symmetry are parameterized by 2 DOF, so the result of the convolution can be represented on the unit sphere – a general kernel would generate a function on SO3.

In ravi's paper he observes that the clamped cosine kernel can be represented in closed form and that most of the energy is in the coefficients through the quadratics. This is the same motivation used in image processing – large kernels are more efficiently convolved in frequency space.



## Normal Mapping

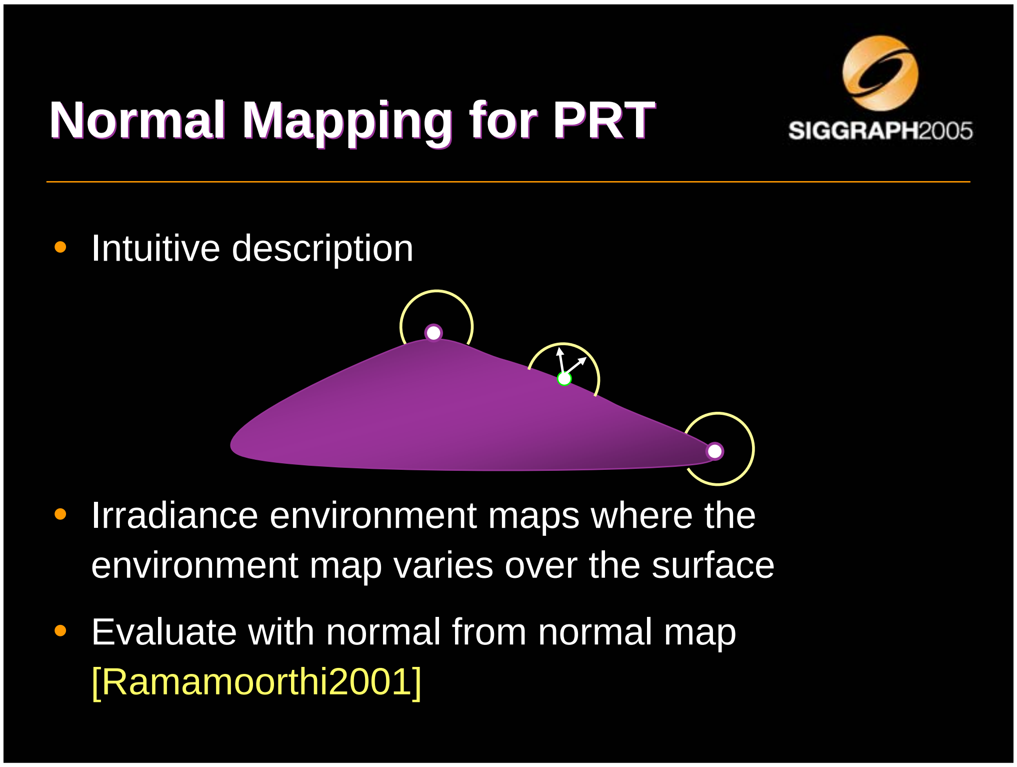
SIGGRAPH2005

- Account for GI of coarse scale (geometry) but not fine scale (normal map)
  - Could use DC of visibility to “poke hole” in lighting environment, with triple products this amounts to scaling light coeffs by DC
    - Mathematical justification for ambient occlusion
- $\mathbf{L}_{\mathbf{xfer}}$ just needs to be quadratic [Ramamoorthi2001]

The technique presented will also only account for global illumination at the scale of the geometry, how bumps cast shadows/reflect light on themselves will not be modeled.

If you scaled the lighting environment by the DC term of the SH projection of visibility, that’s mathematically equivalent to taking the product of the lighting and the visibility projected into SH – which is a mathematical justification for ambient occlusion.

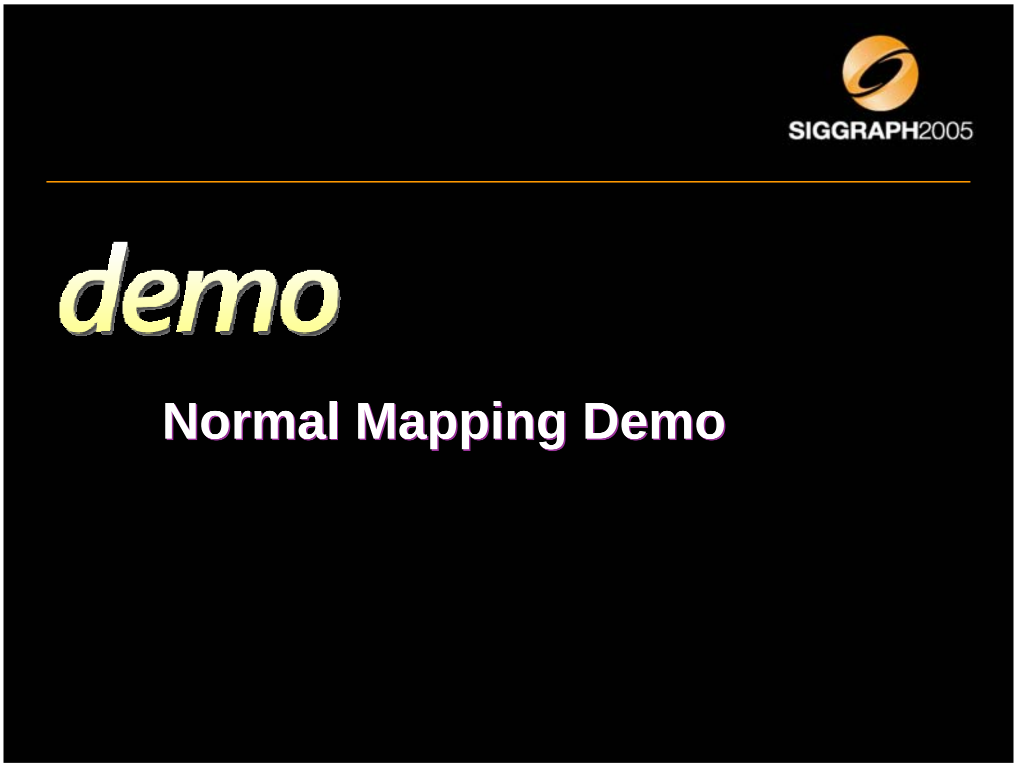
A result of the irradiance environment map paper is that transferred radiance only need to be quadratic if the shadowing of the bumps is not being taken into account.



## Normal Mapping for PRT

- Intuitive description
- Irradiance environment maps where the environment map varies over the surface
- Evaluate with normal from normal map [Ramamoorthi2001]

The easiest way to think about the technique that will be described is you have a smoothly varying irradiance environment map, that you want to evaluate using a normal looked up from the normal map.



SIGGRAPH2005

demo

Normal Mapping Demo

[ load simple scene ]

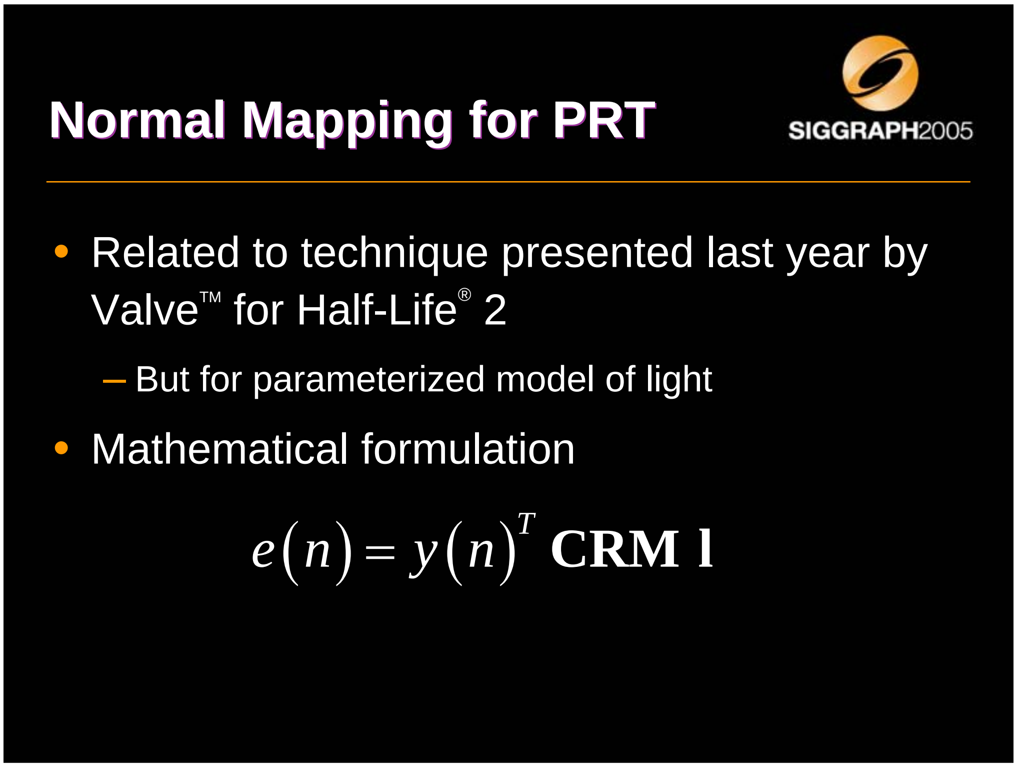
[ go into normal map mode ]

[ turn off light probe, turn on directional light ]

Here is a fairly coarsely tesselated mesh with a normal map, the precomputation knows nothing about the actual normal map being used. We can change the scale of the normal map, or switch in a different normal map, and use the same simulation results.

This achieves the desired effect – a coars PRT simulation is combined with a high frequency normal map.

Here I will toggle between the “ideal” case and an approximation that I will describe in a bit. This approximation is quite good, but there is clearly some error near the silhoutte – which might be a result of compression.



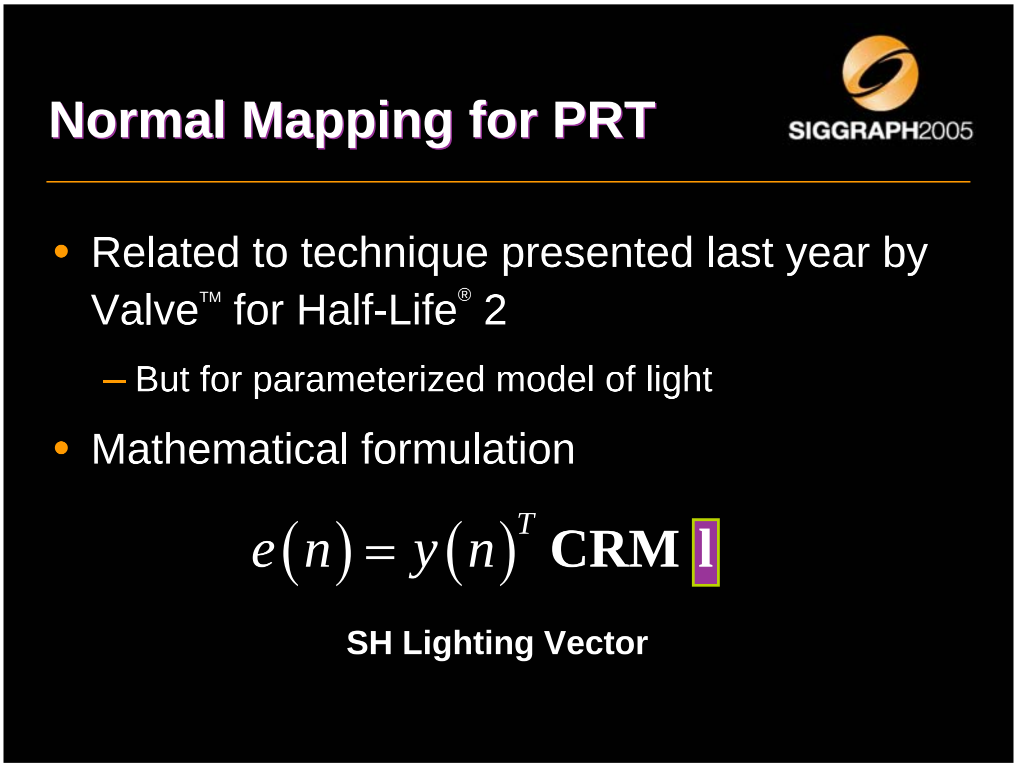
## Normal Mapping for PRT

SIGGRAPH2005

- Related to technique presented last year by Valve™ for Half-Life® 2
  - But for parameterized model of light
- Mathematical formulation

$$e(n) = y(n)^T \, \mathbf{CRM} \; \mathbf{l}$$

Valve showed a similar technique at GDC04, but just for static lighting. The idea presented here is for a parameterized model of lighting.



## Normal Mapping for PRT

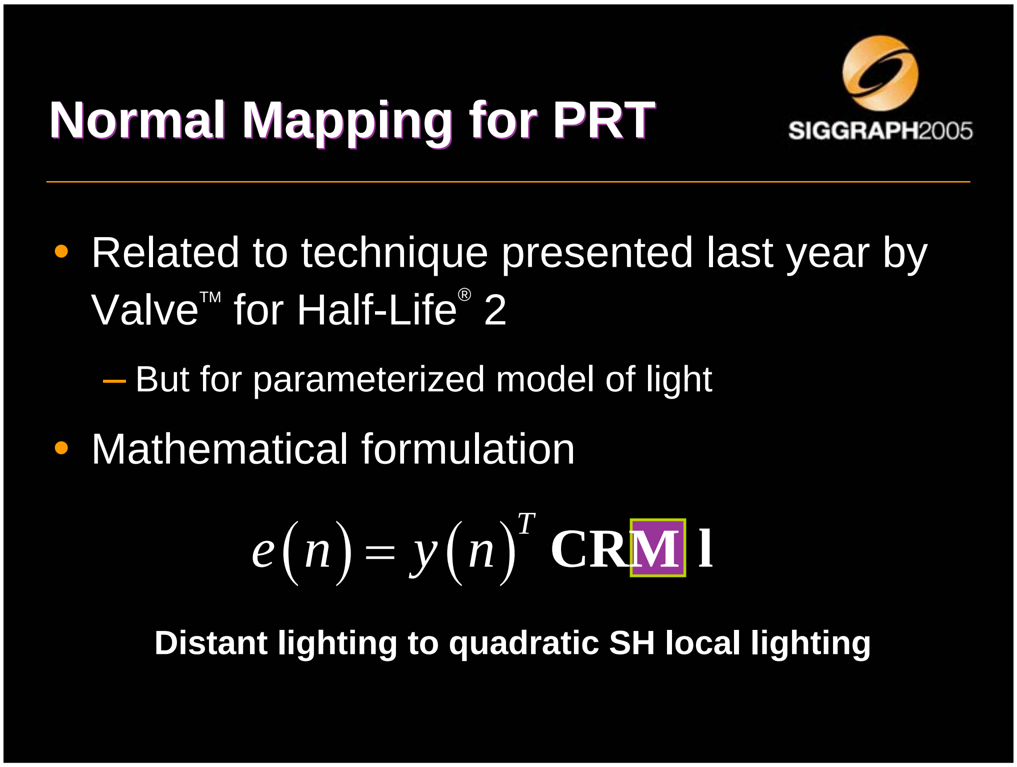
SIGGRAPH2005

- Related to technique presented last year by Valve™ for Half-Life® 2
  - But for parameterized model of light
- Mathematical formulation

$$e(n) = y(n)^T \,\mathbf{CRM}\, \boxed{\mathbf{l}}$$

**SH Lighting Vector**

The first term is the distant environment map lighting.



## Normal Mapping for PRT

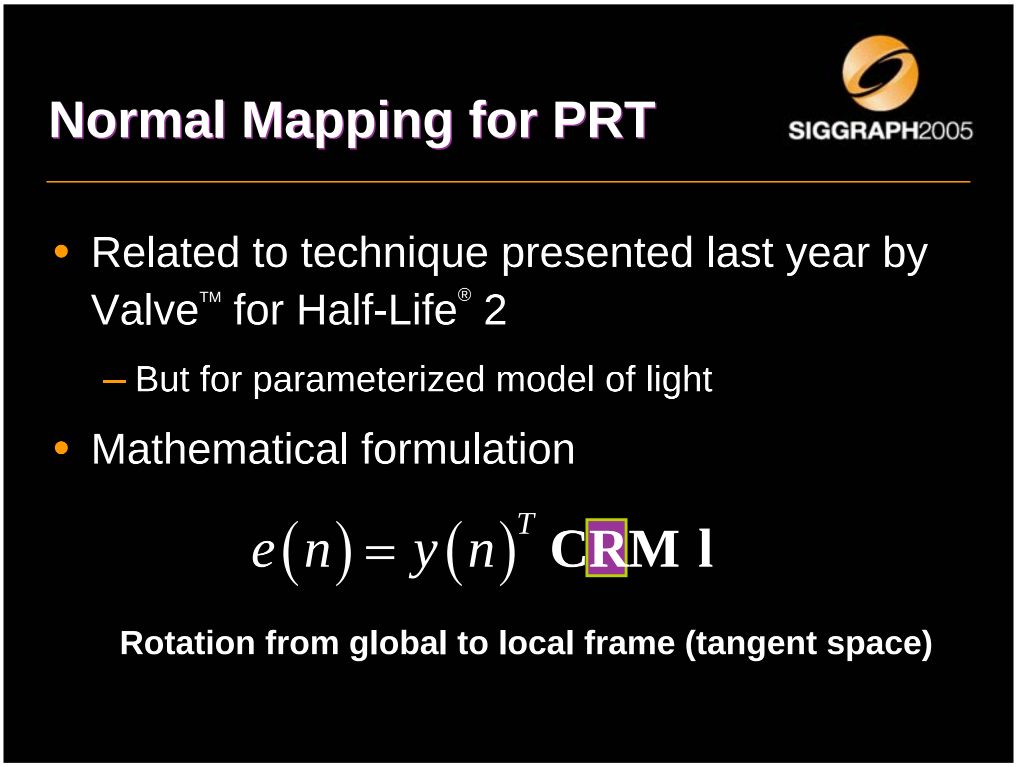
SIGGRAPH2005

- Related to technique presented last year by Valve™ for Half-Life® 2
  - But for parameterized model of light
- Mathematical formulation

$$e(n) = y(n)^T \mathbf{C}\mathbf{R}\boxed{\mathbf{M}}\,\mathbf{l}$$

**Distant lighting to quadratic SH local lighting**

The next term is a transfer matrix that maps distant lighting to quadratic transferred incident lighting



## Normal Mapping for PRT

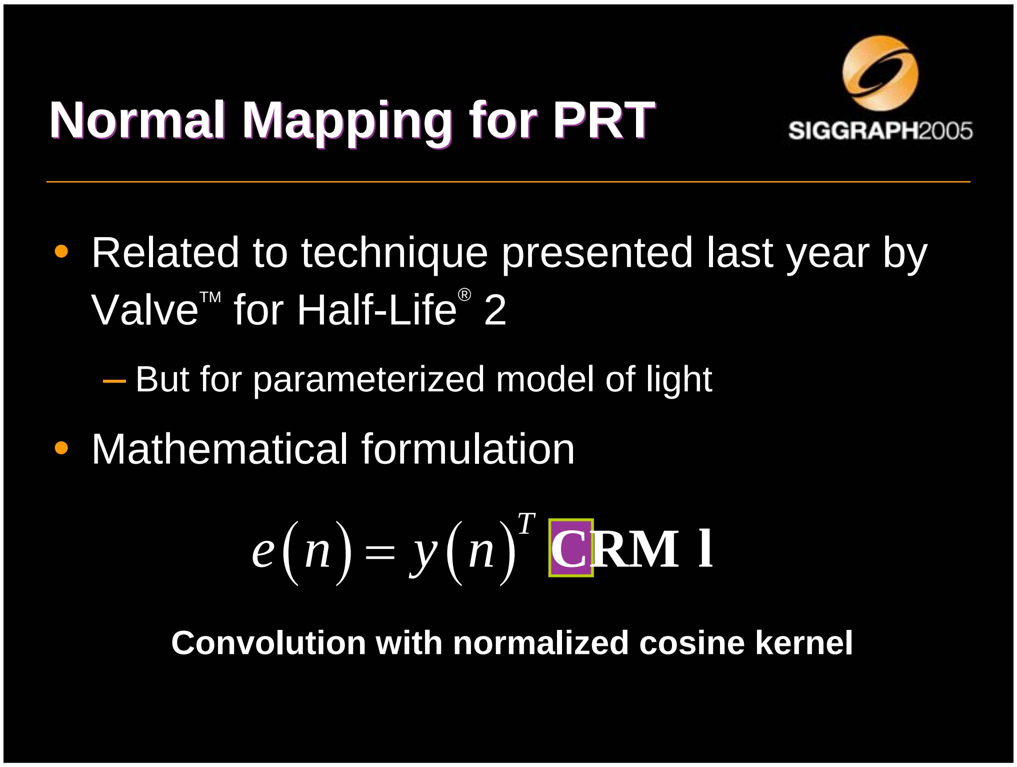
SIGGRAPH2005

- Related to technique presented last year by Valve™ for Half-Life® 2
  - But for parameterized model of light
- Mathematical formulation

$$e(n) = y(n)^T \mathbf{C}\boxed{\mathbf{R}}\mathbf{M}\,\mathbf{l}$$

**Rotation from global to local frame (tangent space)**

Then there is a rotation from the global to the local frame



## Normal Mapping for PRT

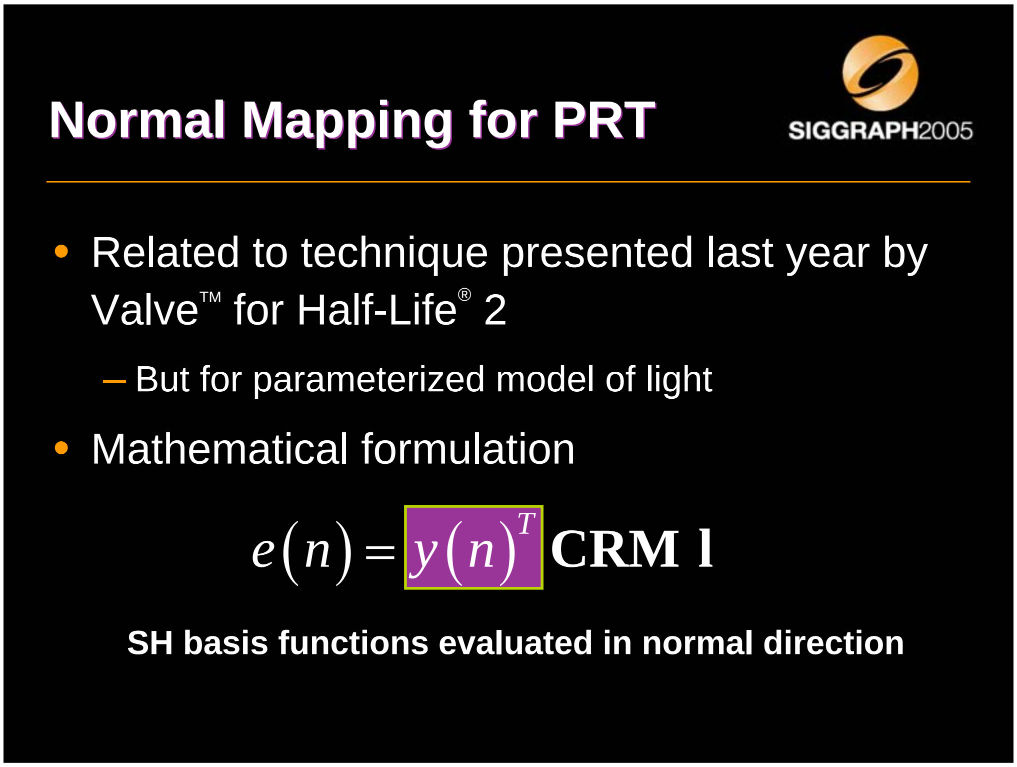
SIGGRAPH2005

- Related to technique presented last year by Valve™ for Half-Life® 2
  - But for parameterized model of light
- Mathematical formulation

$$e(n) = y(n)^T \mathbf{C}\mathbf{R}\mathbf{M}\,\mathbf{l}$$

**Convolution with normalized cosine kernel**

And finaly a diagonal matrix that convolves the signal with the coefficients for the normalized cosine kernel



## Normal Mapping for PRT

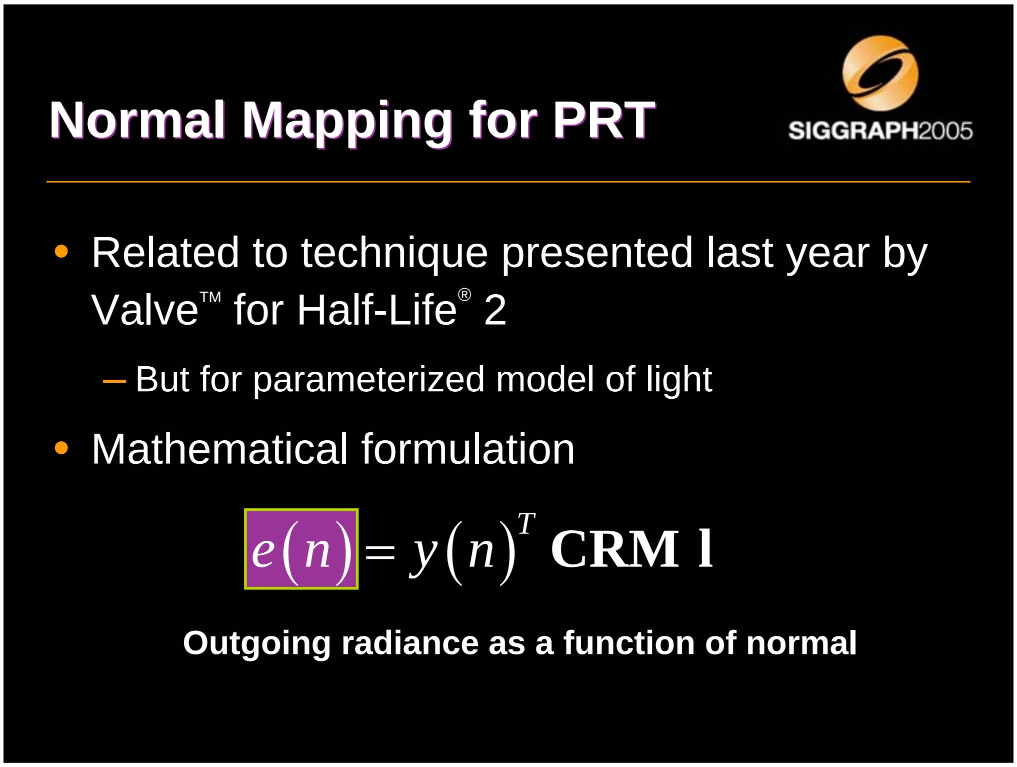
SIGGRAPH2005

- Related to technique presented last year by Valve™ for Half-Life® 2
  - But for parameterized model of light
- Mathematical formulation

$$e(n) = \boxed{y(n)^T} \mathbf{CRM}\,\mathbf{l}$$

**SH basis functions evaluated in normal direction**

You then have the SH basis functions (quadratic) evaluated in the normal direction.



## Normal Mapping for PRT

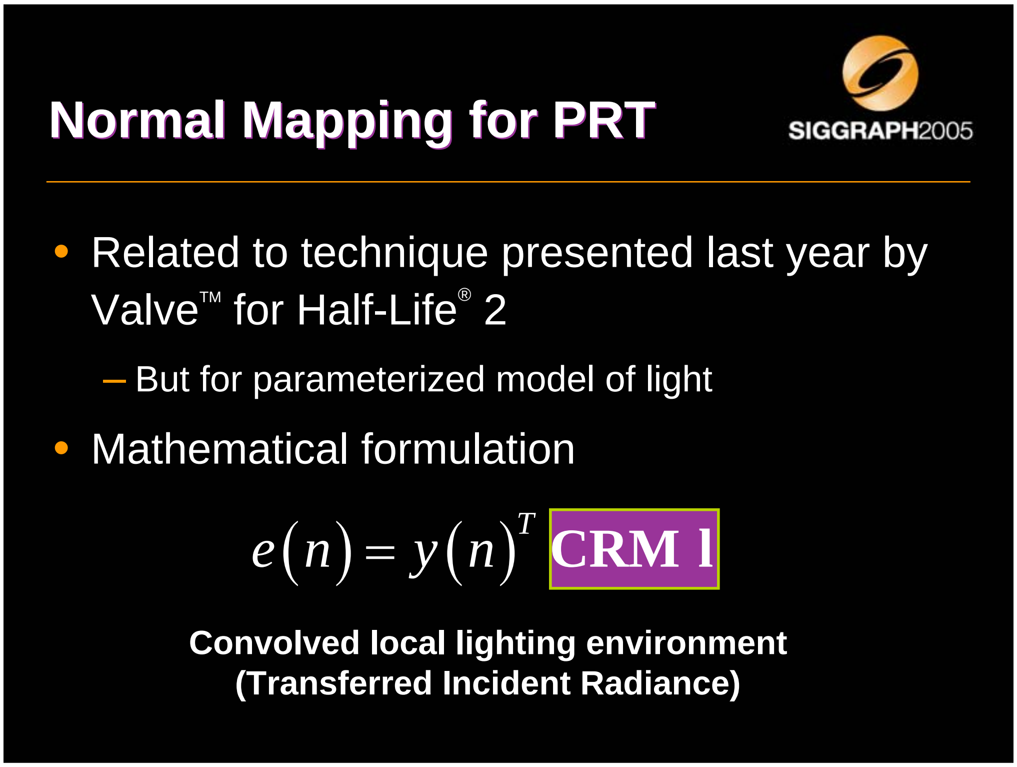
SIGGRAPH2005

- Related to technique presented last year by Valve™ for Half-Life® 2
  - But for parameterized model of light
- Mathematical formulation

$$\boxed{e(n)} = y(n)^T \mathbf{CRM}\ \mathbf{l}$$

**Outgoing radiance as a function of normal**

Finally this generates outgoing radiance for the given normal



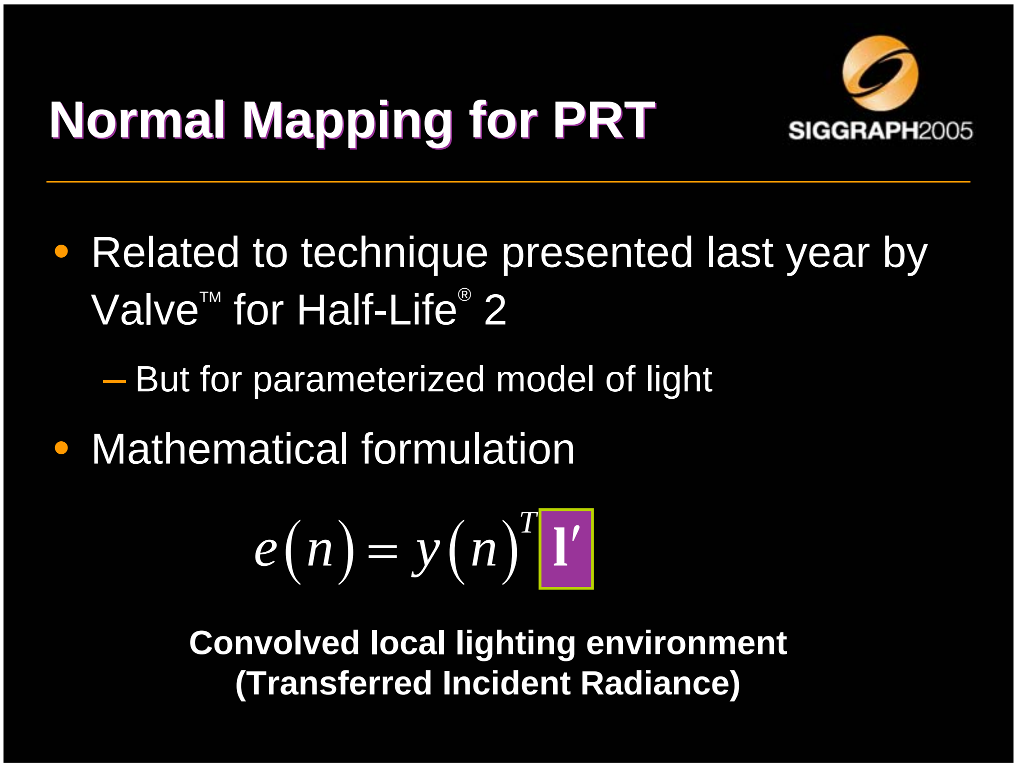
## Normal Mapping for PRT

SIGGRAPH2005

- Related to technique presented last year by Valve™ for Half-Life® 2
  - But for parameterized model of light
- Mathematical formulation

$$e(n) = y(n)^{T}\ \boxed{\text{CRM l}}$$

**Convolved local lighting environment (Transferred Incident Radiance)**



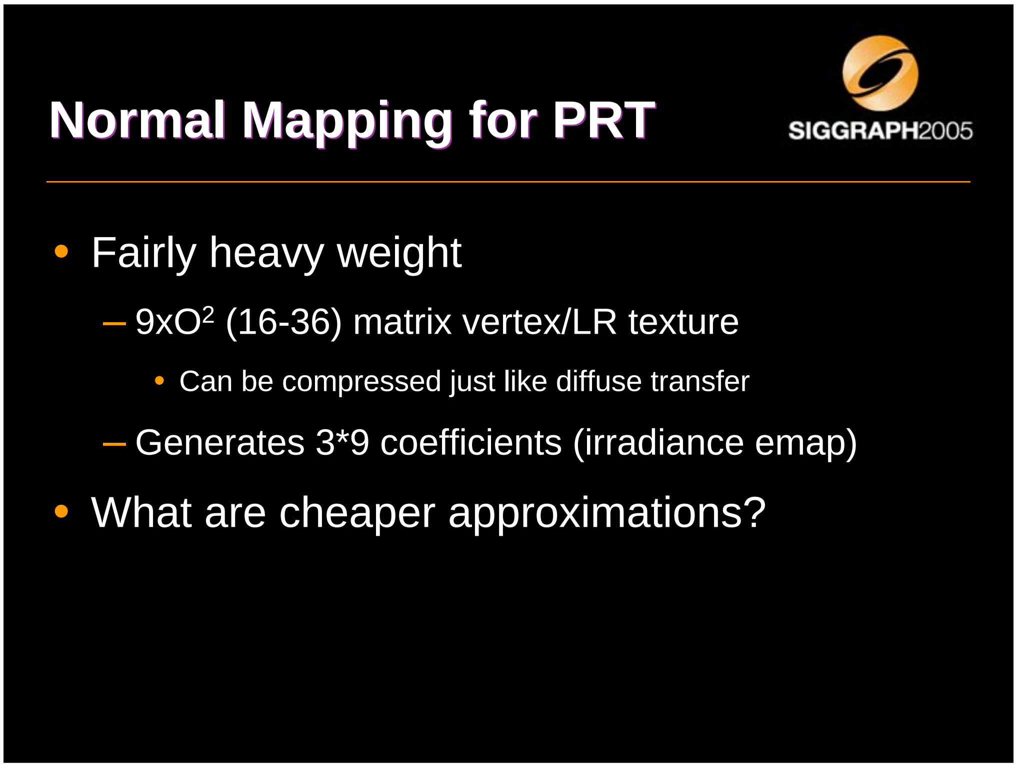
## Normal Mapping for PRT

SIGGRAPH2005

- Related to technique presented last year by Valve™ for Half-Life® 2
  - But for parameterized model of light
- Mathematical formulation

$$e(n) = y(n)^T \boxed{\mathbf{l}'}$$

**Convolved local lighting environment (Transferred Incident Radiance)**



## Normal Mapping for PRT

SIGGRAPH2005

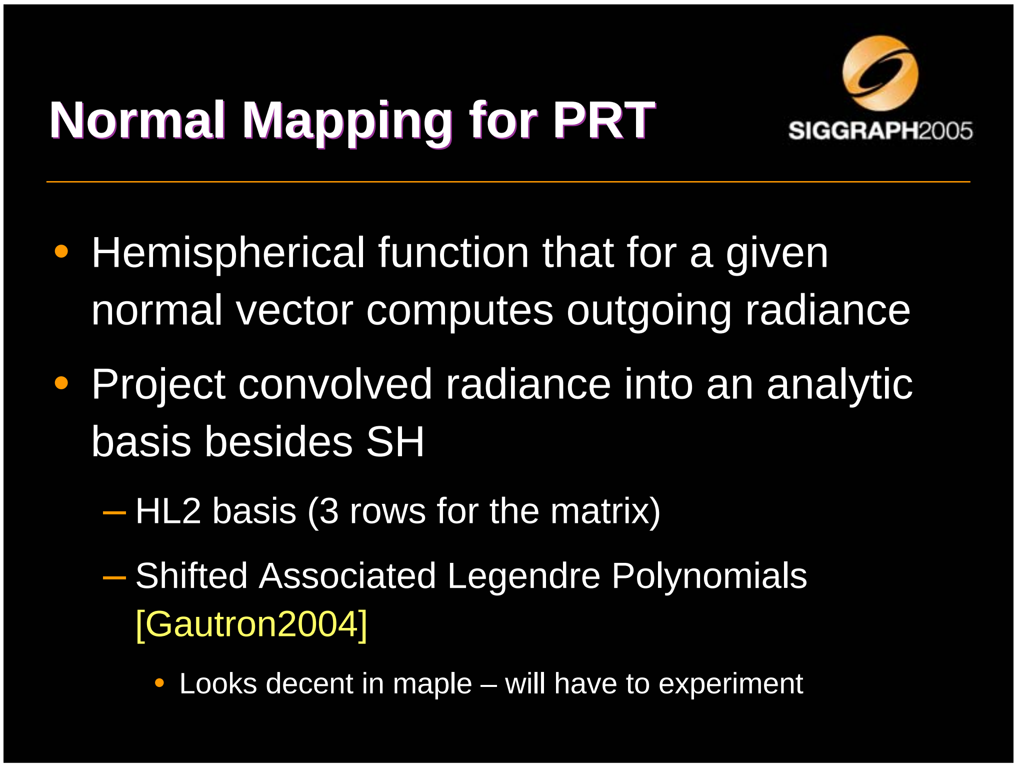
- Fairly heavy weight
  - 9xO$^2$ (16-36) matrix vertex/LR texture
    - Can be compressed just like diffuse transfer
  - Generates 3*9 coefficients (irradiance emap)
- What are cheaper approximations?

Evaluating the technique as outlined in the previous slides is still fairly heavy weight.

You require 9xorder^2 matrices per vertex or in a low res texture (which can be compressed.)

More importantly that generates 27 coefficients that are the irradiance environment map that have to then be evaluated.

So we are going to go over some “cheaper” approximations.



## Normal Mapping for PRT

SIGGRAPH2005

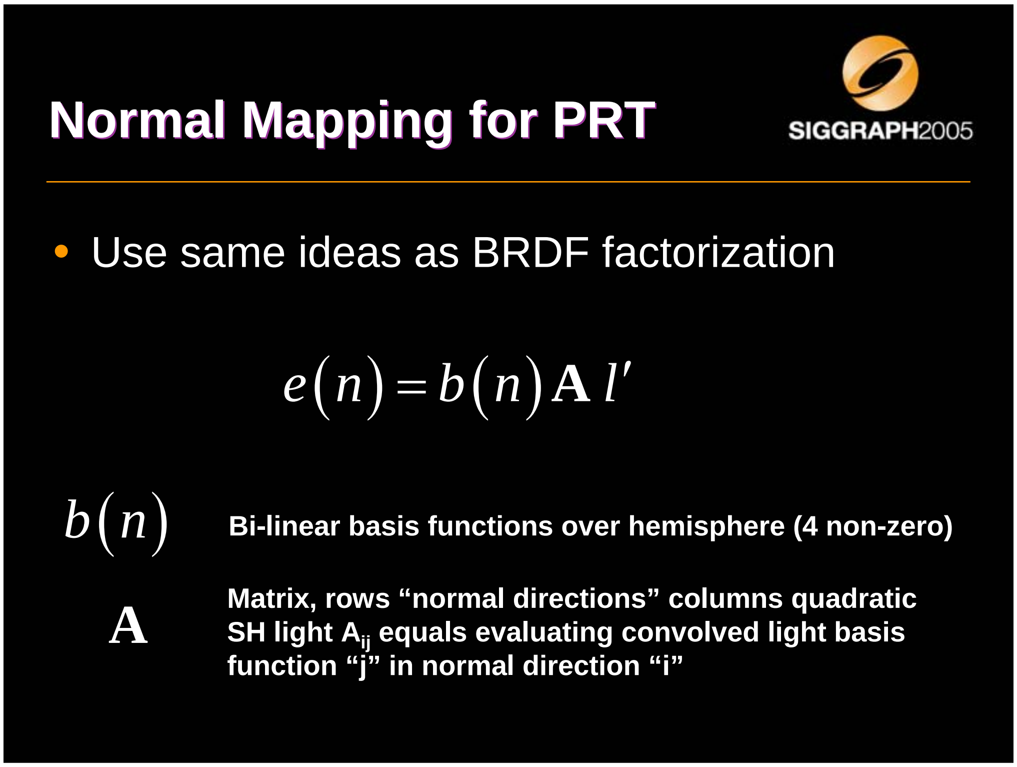
- Hemispherical function that for a given normal vector computes outgoing radiance
- Project convolved radiance into an analytic basis besides SH
  - HL2 basis (3 rows for the matrix)
  - Shifted Associated Legendre Polynomials [Gautron2004]
    - Looks decent in maple – will have to experiment

The basic problem is that you have a hemispherical function (outgoing radiance) that is a function of the normal (not view direction as we have done before.)

One simple approach would be to project this hemispherical function into some other basis.

HalfLife2 uses 3 quadratic polynomials on portions of the hemisphere.

You could also use the basis functions from last years EGSR paper, these look like than can do a reasonable job approximating convolved SH functions in maple, but more experimentation needs to be done.



## Normal Mapping for PRT

SIGGRAPH2005

- Use same ideas as BRDF factorization

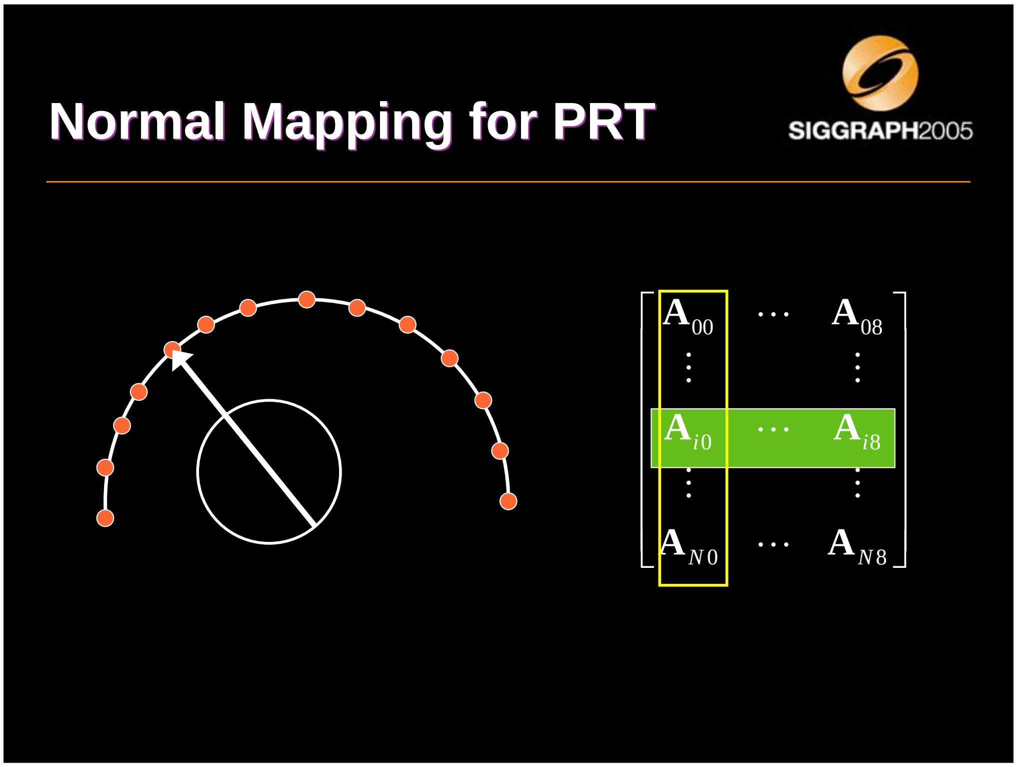
$$e(n) = b(n)\,\mathbf{A}\,l'$$

$b(n)$ Bi-linear basis functions over hemisphere (4 non-zero)

$\mathbf{A}$ Matrix, rows “normal directions” columns quadratic SH light $A_{ij}$ equals evaluating convolved light basis function “j” in normal direction “i”

Another approach, that we will discuss here is to use the same ideas that are used for BRDF factorization, which Jaakko talked about earlier.

That is build a matrix “A” sampling the convolved lighting environment over a hemisphere of normals. The rows represent the normal directions, the columns the lighting environment.



## Normal Mapping for PRT

SIGGRAPH2005

$$\begin{bmatrix} \mathbf{A}_{00} & \cdots & \mathbf{A}_{08} \\ \vdots & & \vdots \\ \mathbf{A}_{i0} & \cdots & \mathbf{A}_{i8} \\ \vdots & & \vdots \\ \mathbf{A}_{N0} & \cdots & \mathbf{A}_{N8} \end{bmatrix}$$

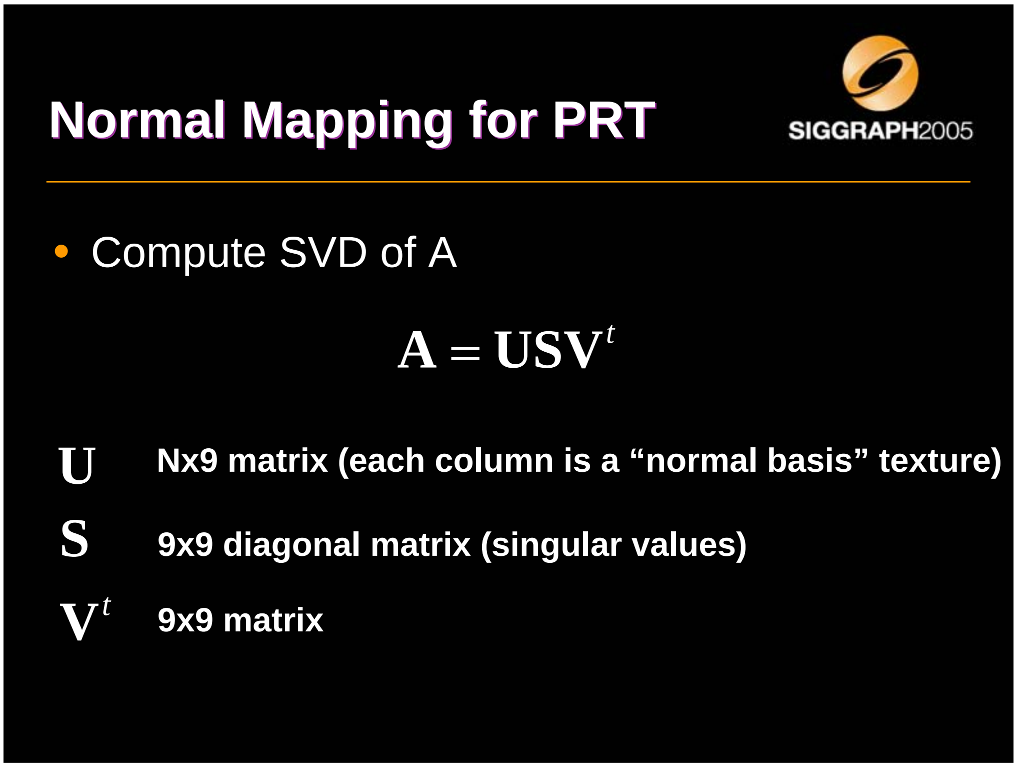
This is just trying to give more intuition to the matrix A.

You have some sampling of the hemisphere, each row corresponds to a sample.

The columns correspond to illumination from the 9 quadratic SH basis functions.

A coefficient in the matrix represents the integral of a cosine kernel in the given direction against the lighting environment. Not that the lighting environment should be clamped to the hemisphere before this integral happens.

Bi-linear basis functions are used on the unit disk, and then mapped to the hemisphere to generate a value at any point on the hemisphere (so there are at most 4 non-zero entries for a given normal.)



## Normal Mapping for PRT

SIGGRAPH2005

- Compute SVD of A

$$\mathbf{A} = \mathbf{U}\mathbf{S}\mathbf{V}^t$$

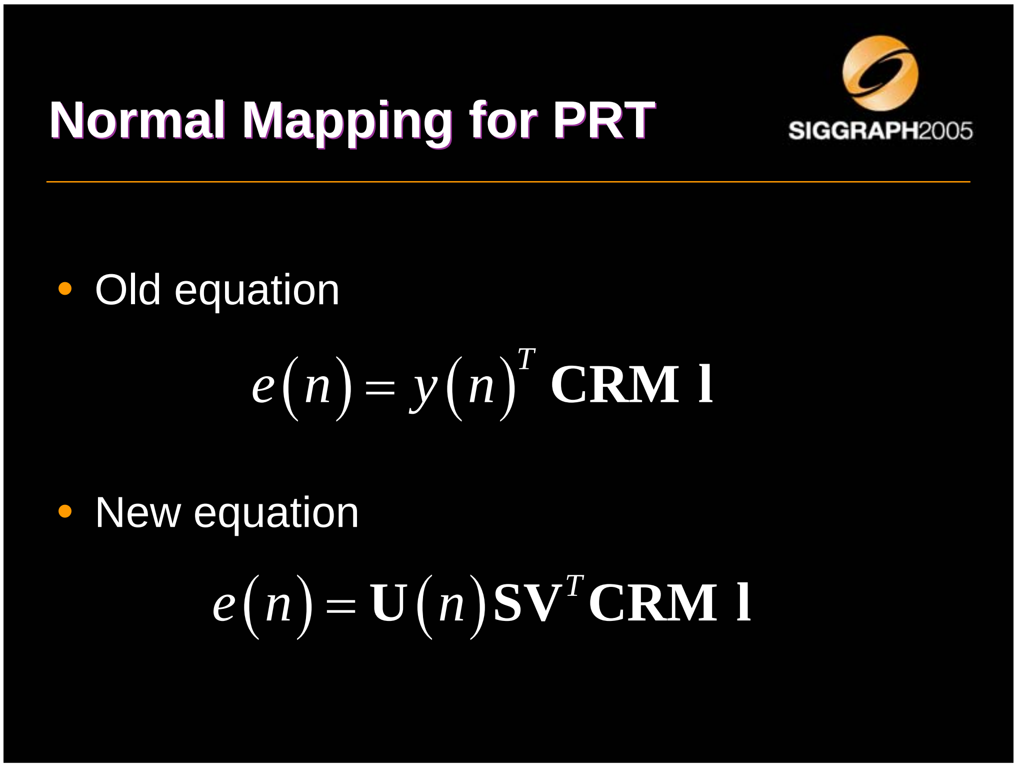
| | |
|---|---|
| $\mathbf{U}$ | **Nx9 matrix (each column is a “normal basis” texture)** |
| $\mathbf{S}$ | **9x9 diagonal matrix (singular values)** |
| $\mathbf{V}^t$ | **9x9 matrix** |

Then you compute the singular value decomposition of this matrix and you get 3 terms.

A matrix U, where each column represents a “normal basis” texture.

A diagonal matrix S (the singular values)

And a matrix Vt,which is 9x9 (for quadratic SH.)



## Normal Mapping for PRT

SIGGRAPH2005

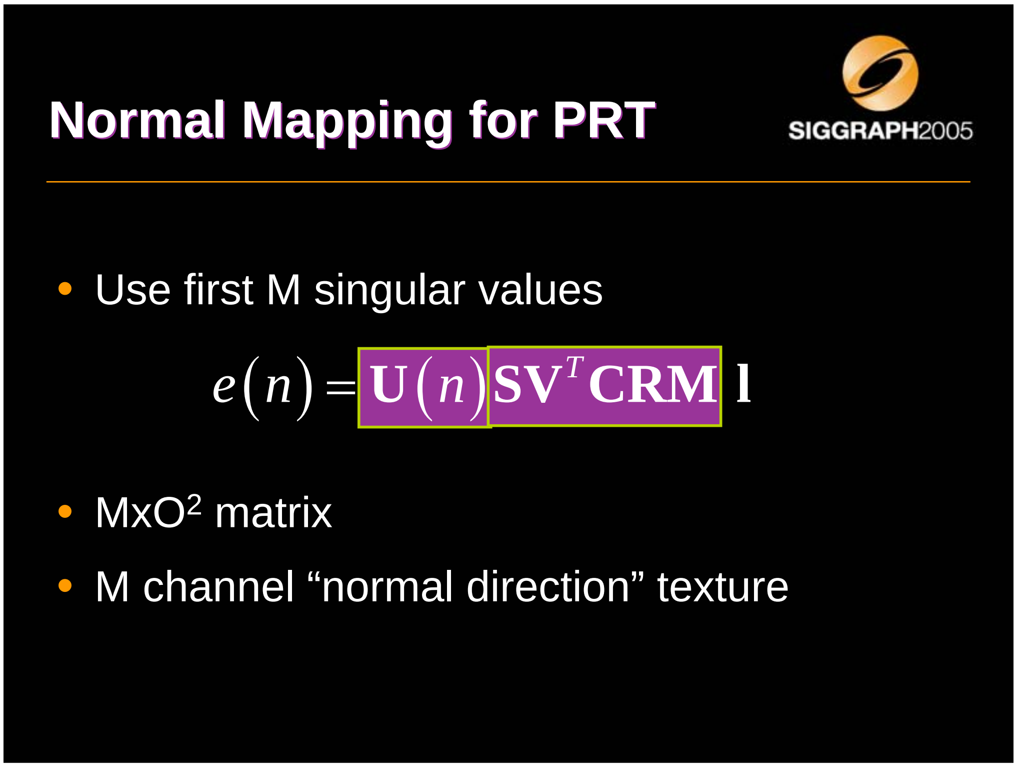
- Old equation

$$e(n) = y(n)^T \,\mathbf{CRM}\ \mathbf{l}$$

- New equation

$$e(n) = \mathbf{U}(n)\mathbf{S}\mathbf{V}^T\mathbf{CRM}\ \mathbf{l}$$

This generates a new equation that simply replaces the evaluation of the quadratic SH in the normal direction.



## Normal Mapping for PRT

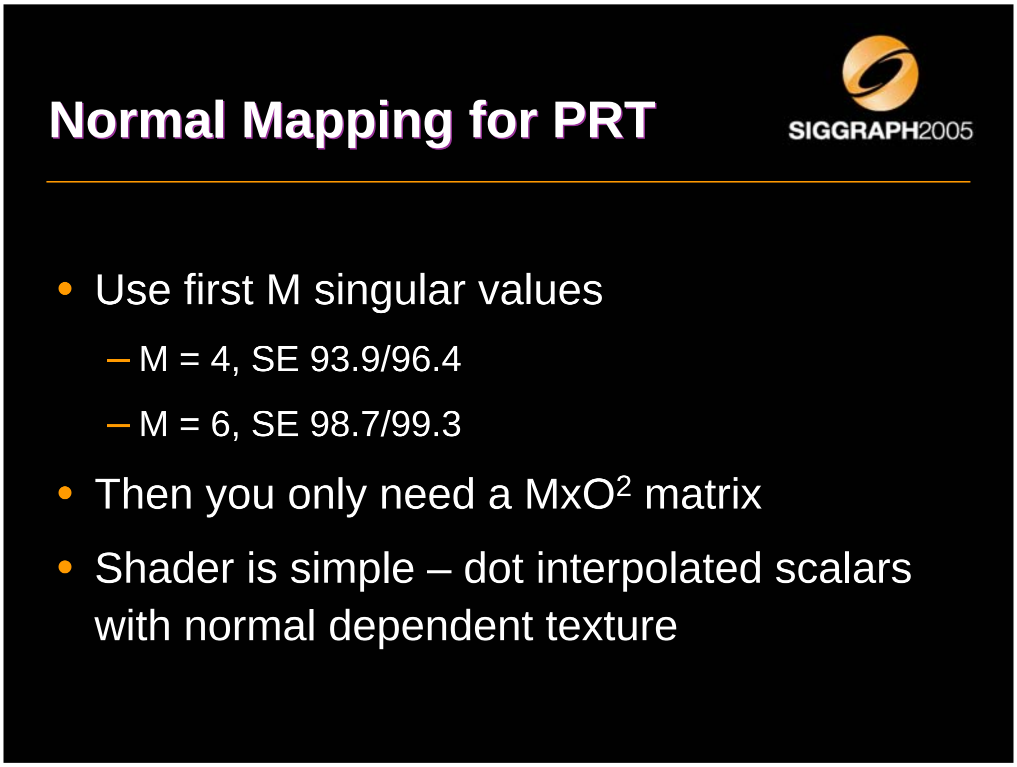
SIGGRAPH2005

- Use first M singular values

$$e(n) = \boxed{\mathbf{U}(n)}\boxed{\mathbf{S}\mathbf{V}^{T}\mathbf{C}\mathbf{R}\mathbf{M}}\,\mathbf{l}$$

- MxO$^2$ matrix
- M channel “normal direction” texture

Then instead of using the full matrix, just use the first M singular values.

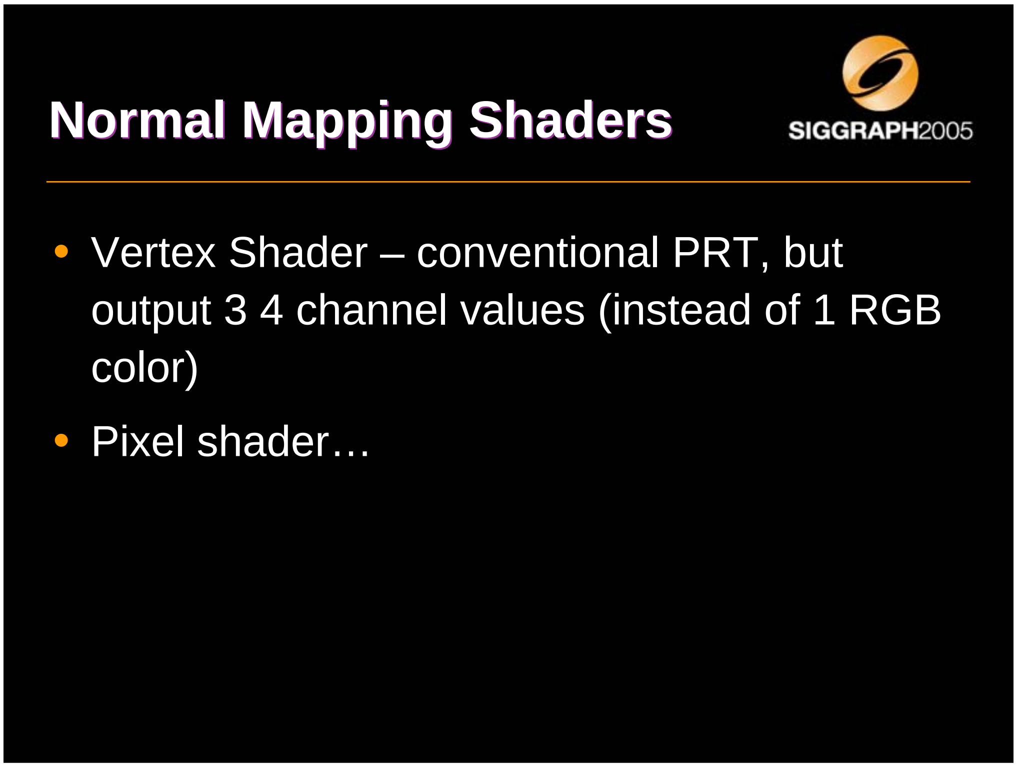


## Normal Mapping for PRT

SIGGRAPH2005

- Use first M singular values
  - M = 4, SE 93.9/96.4
  - M = 6, SE 98.7/99.3
- Then you only need a $\text{M} \times \text{O}^2$ matrix
- Shader is simple – dot interpolated scalars with normal dependent texture

Here you see the accuracy with uniform sampling over the hemisphere, and cosine weighted sampling (normals alligned with Z in tangent space are prefered.)  4 terms is fairly accurate.

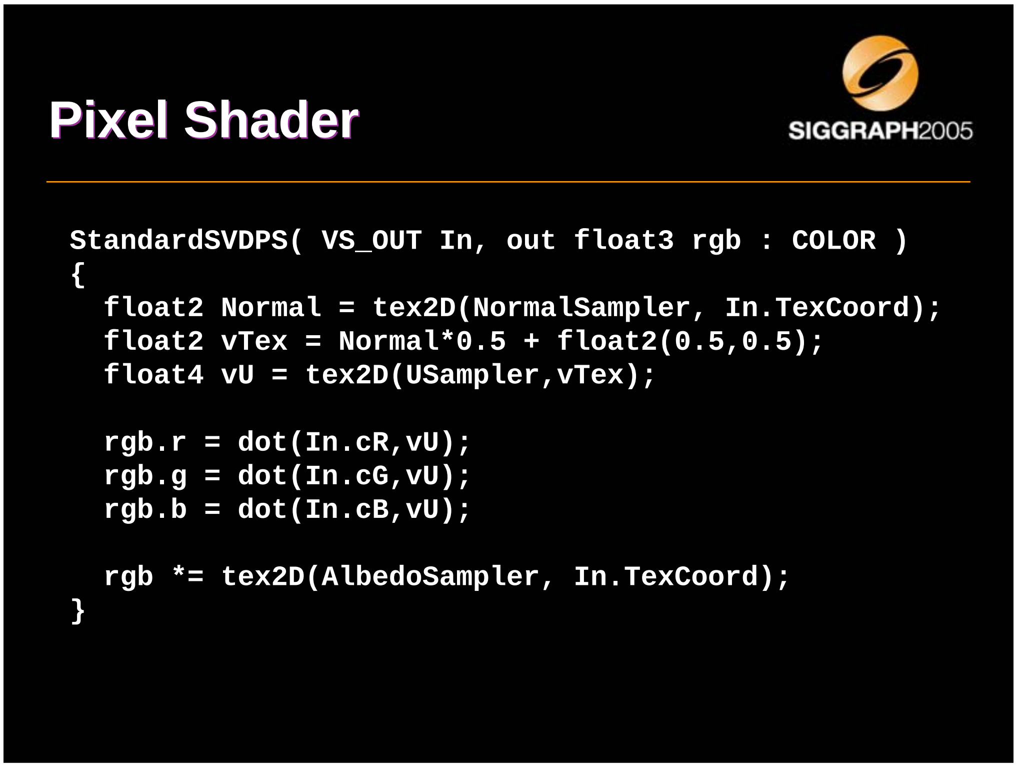


## Normal Mapping Shaders

SIGGRAPH2005

- Vertex Shader – conventional PRT, but output 3 4 channel values (instead of 1 RGB color)
- Pixel shader…

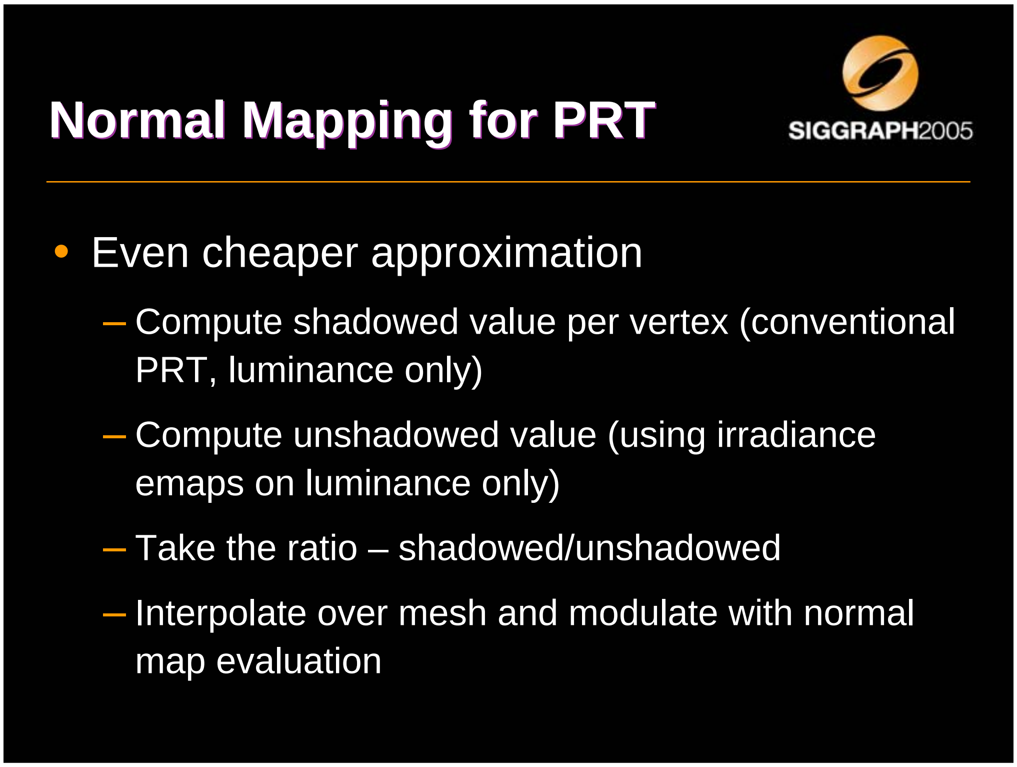
This is not much more complex than a conventional PRT shader.



## Pixel Shader

SIGGRAPH2005

```
StandardSVDPS( VS_OUT In, out float3 rgb : COLOR )
{
  float2 Normal = tex2D(NormalSampler, In.TexCoord);
  float2 vTex = Normal*0.5 + float2(0.5,0.5);
  float4 vU = tex2D(USampler,vTex);

  rgb.r = dot(In.cR,vU);
  rgb.g = dot(In.cG,vU);
  rgb.b = dot(In.cB,vU);

  rgb *= tex2D(AlbedoSampler, In.TexCoord);
}
```

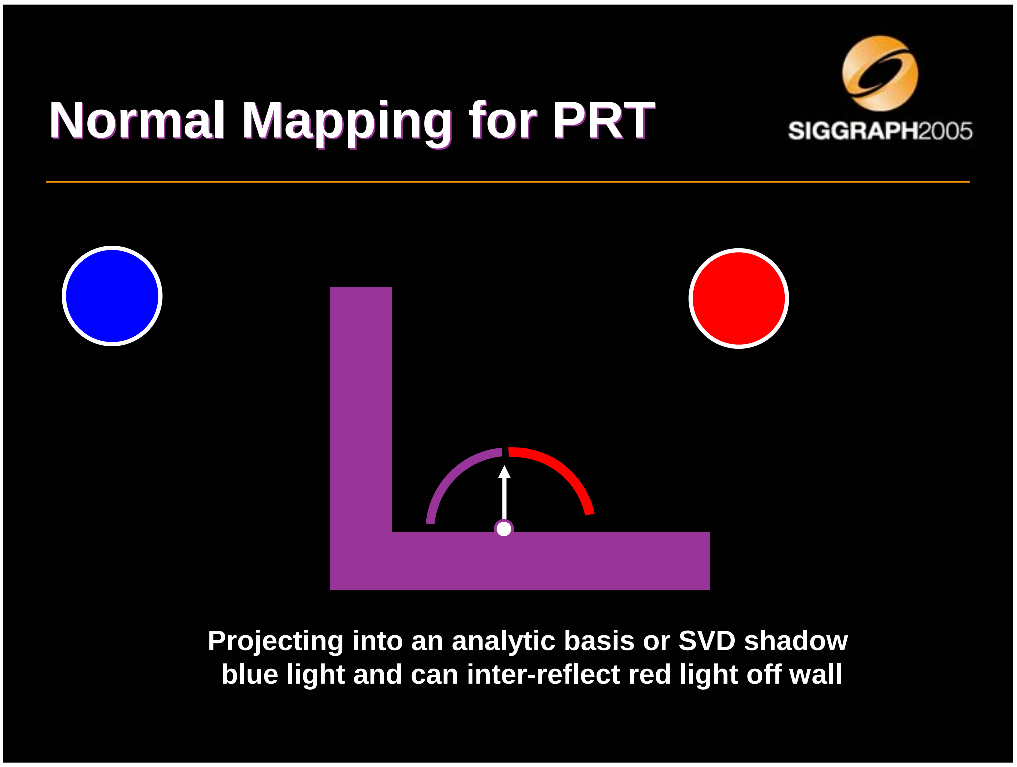


## Normal Mapping for PRT

SIGGRAPH2005

- Even cheaper approximation
  - Compute shadowed value per vertex (conventional PRT, luminance only)
  - Compute unshadowed value (using irradiance emaps on luminance only)
  - Take the ratio – shadowed/unshadowed
  - Interpolate over mesh and modulate with normal map evaluation

A much more approximate solution would be to just use a coarse PRT result to modulate a per-pixel evaluation.

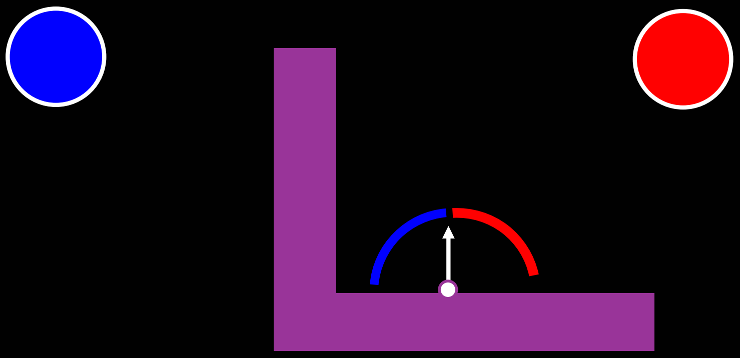


## Normal Mapping for PRT

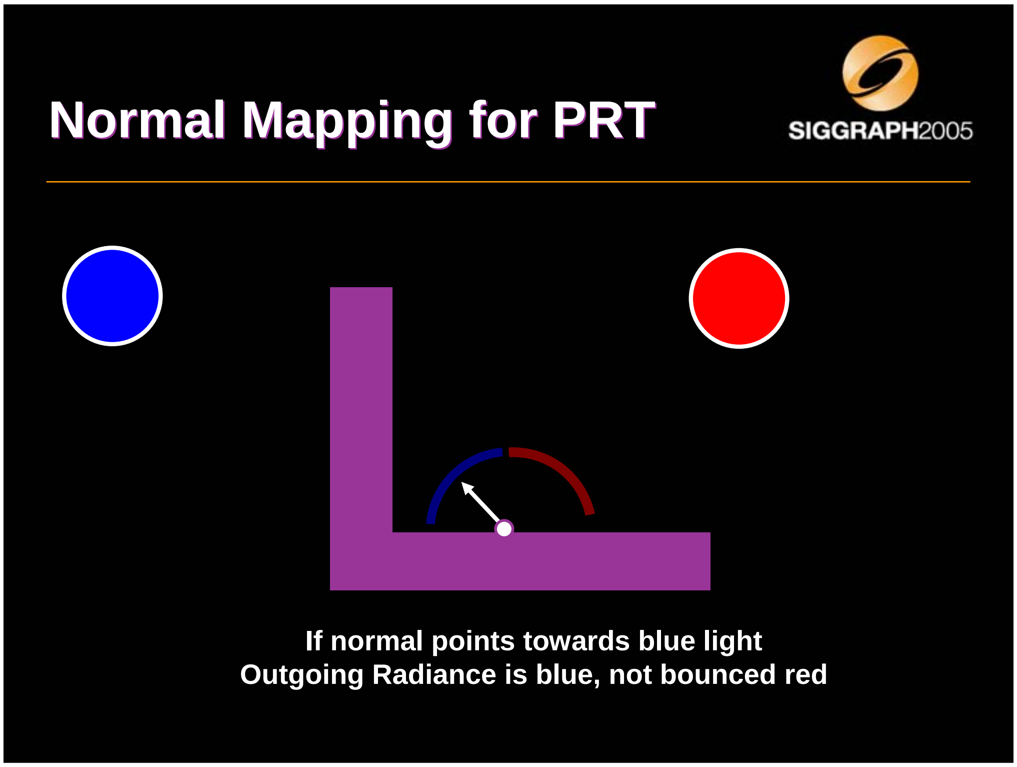
Projecting into an analytic basis or SVD shadow blue light and can inter-reflect red light off wall

## Normal Mapping for PRT

SIGGRAPH2005



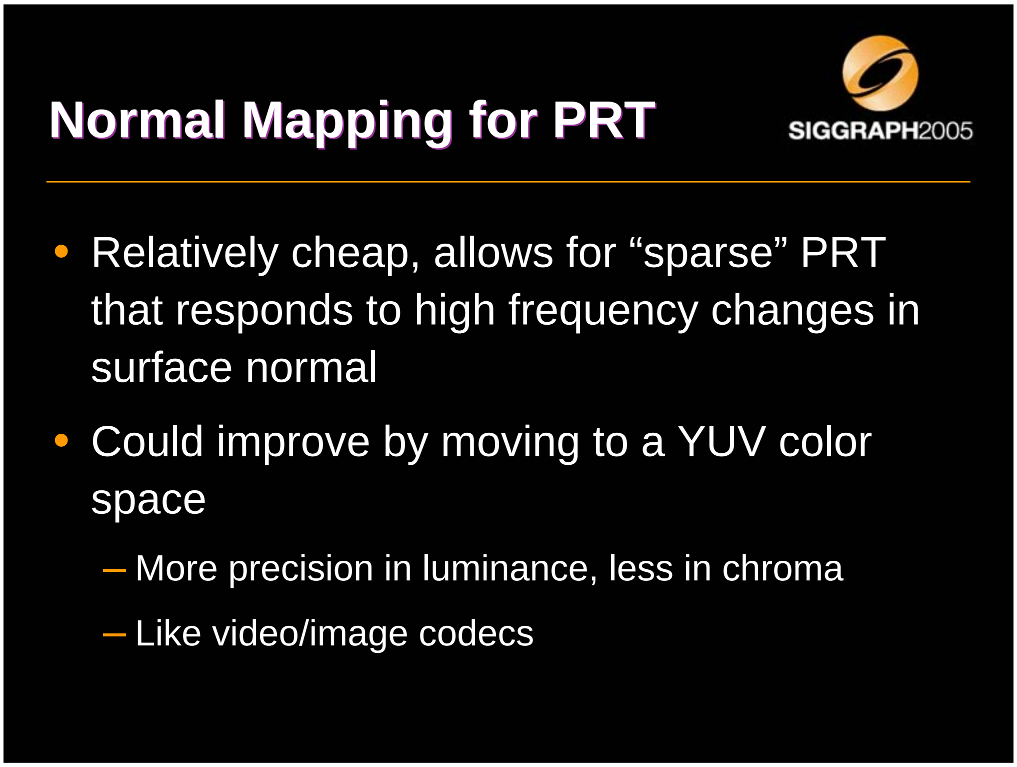
**Most approximate technique effectively
Just “dims” the light**



## Normal Mapping for PRT

SIGGRAPH2005

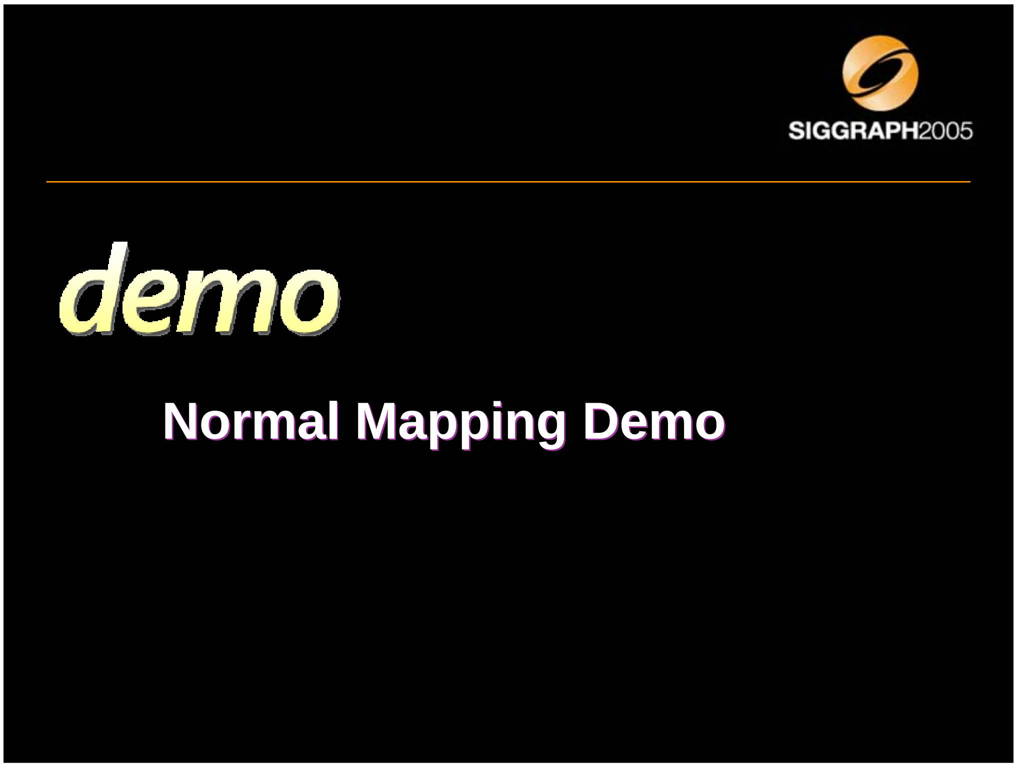
If normal points towards blue light
Outgoing Radiance is blue, not bounced red



## Normal Mapping for PRT

SIGGRAPH2005

- Relatively cheap, allows for “sparse” PRT that responds to high frequency changes in surface normal
- Could improve by moving to a YUV color space
  - More precision in luminance, less in chroma
  - Like video/image codecs



SIGGRAPH2005

demo

## Normal Mapping Demo

[ load simple scene ]

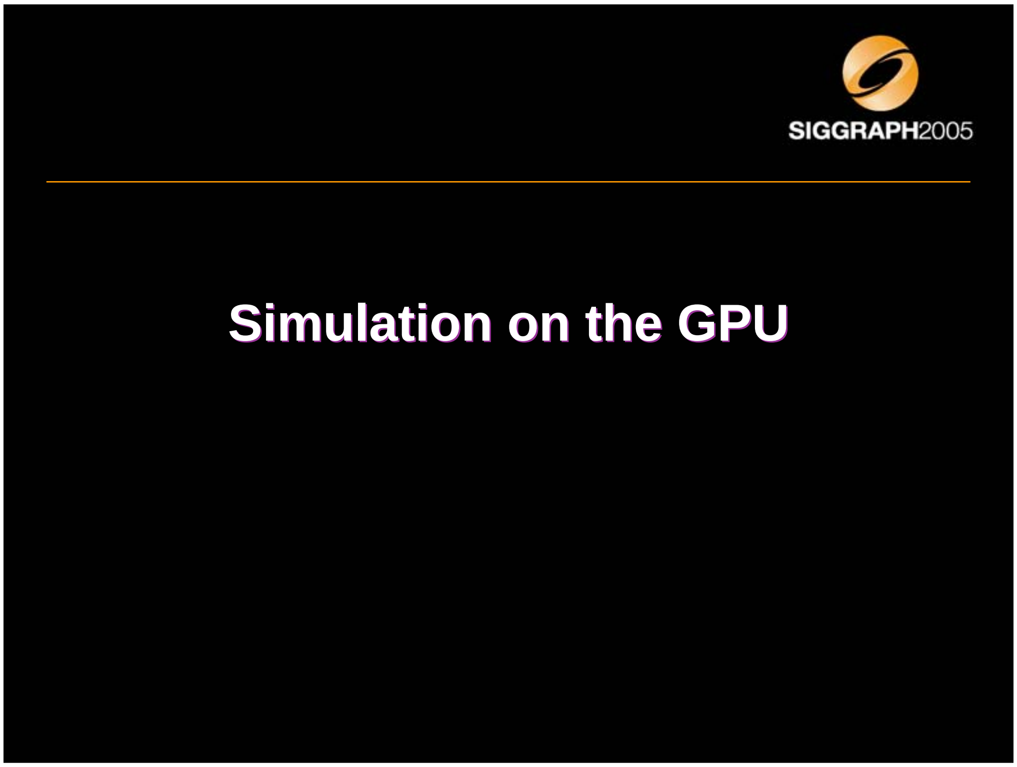
[ go into normal map mode ]

[ turn off light probe, turn on directional light ]

Here is a fairly coarsely tesselated mesh with a normal map, the precomputation knows nothing about the actual normal map being used. We can change the scale of the normal map, or switch in a different normal map, and use the same simulation results.

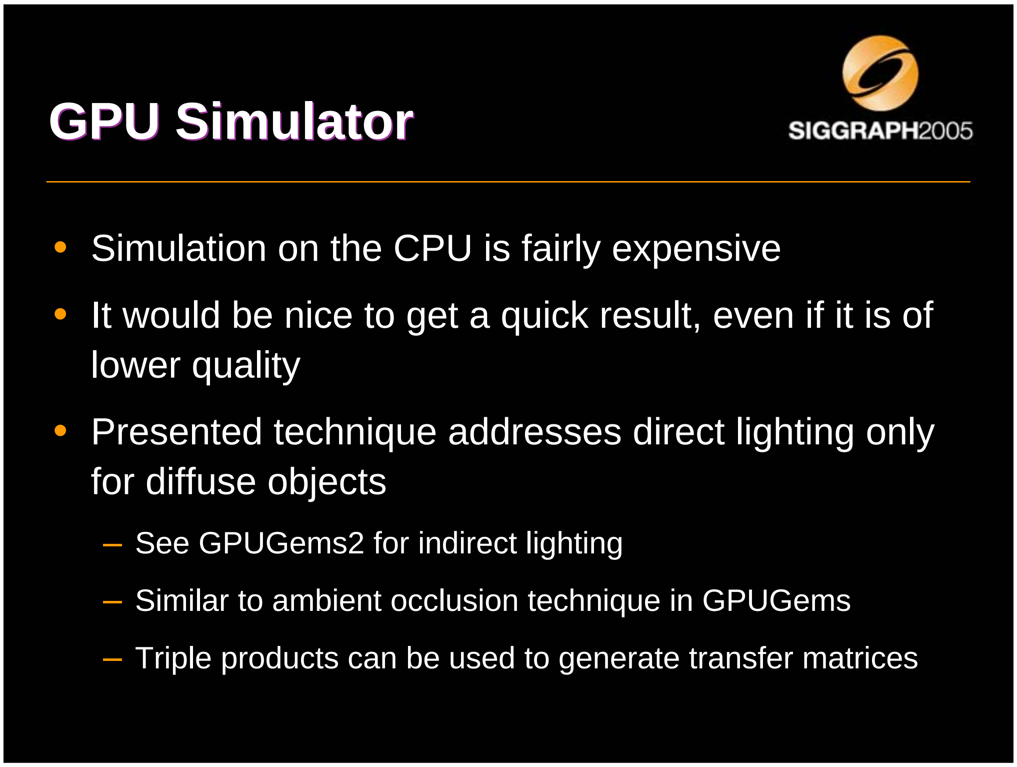
This achieves the desired effect – a coars PRT simulation is combined with a high frequency normal map.

Here I will toggle between the "ideal" case and an approximation that I will describe in a bit. This approximation is quite good, but there is clearly some error near the silhoutte – which might be a result of compression.



SIGGRAPH2005

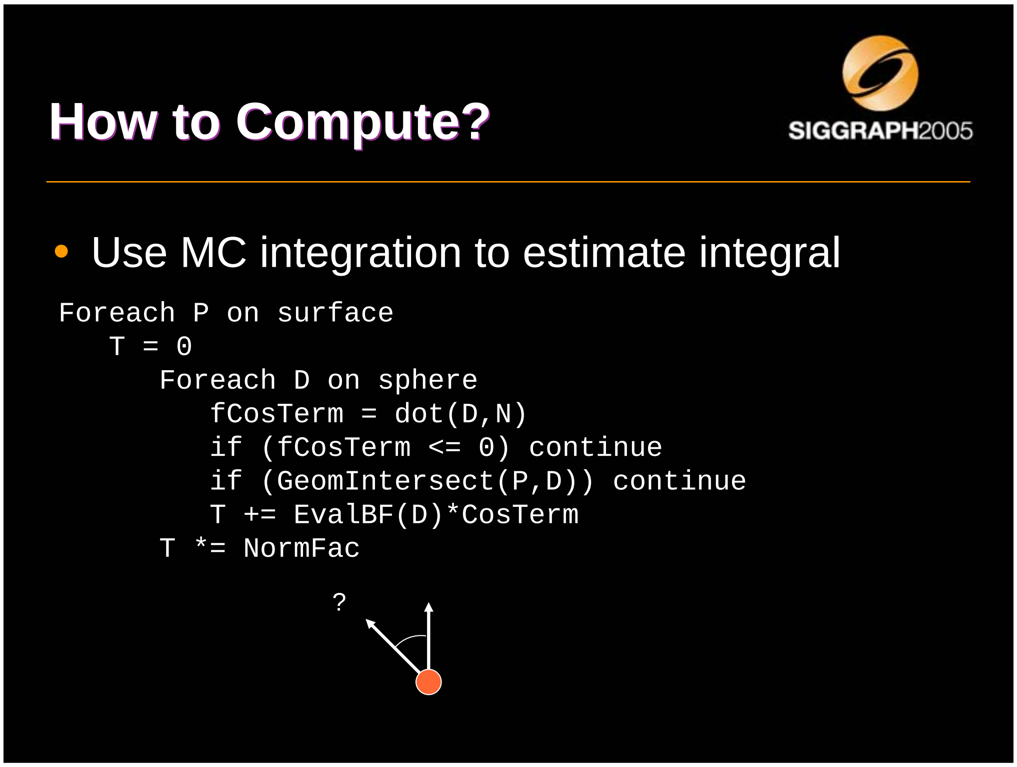
# Simulation on the GPU



## GPU Simulator

SIGGRAPH2005

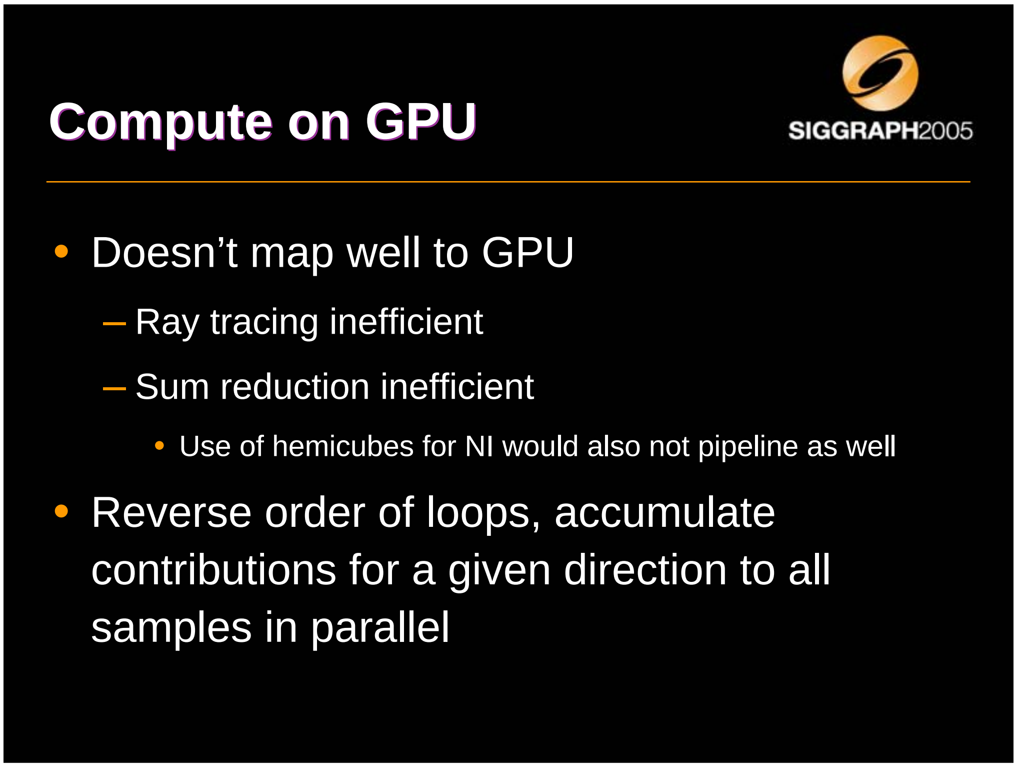
- Simulation on the CPU is fairly expensive
- It would be nice to get a quick result, even if it is of lower quality
- Presented technique addresses direct lighting only for diffuse objects
  - See GPUGems2 for indirect lighting
  - Similar to ambient occlusion technique in GPUGems
  - Triple products can be used to generate transfer matrices



## How to Compute?

- Use MC integration to estimate integral

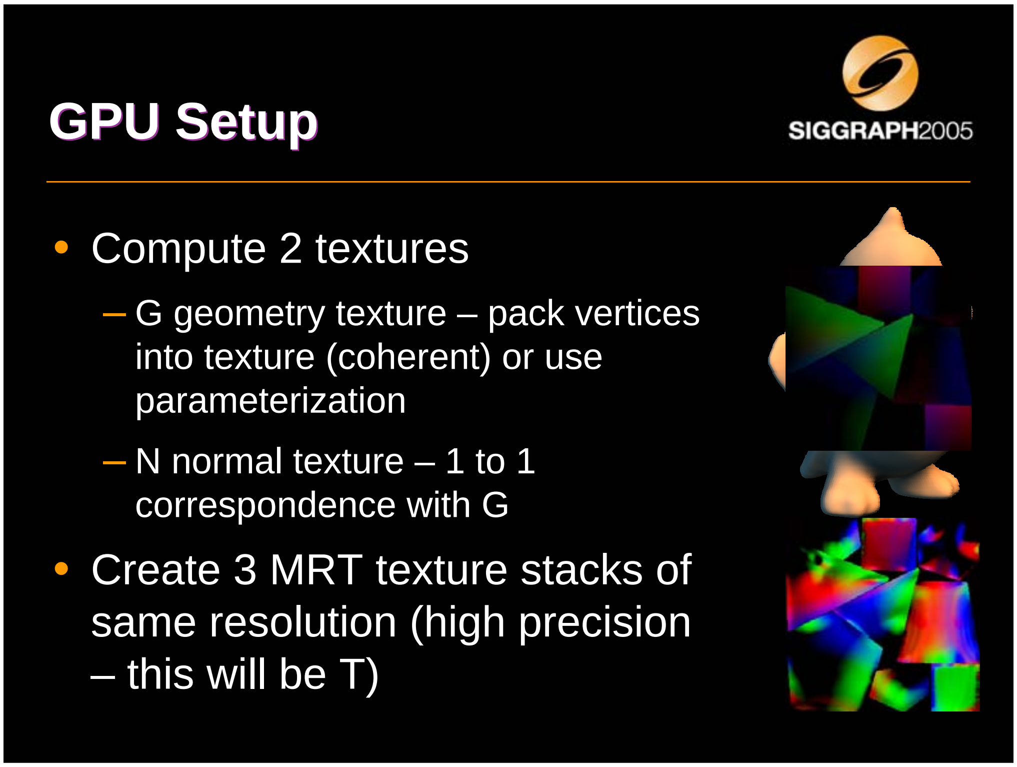
```
Foreach P on surface
   T = 0
      Foreach D on sphere
         fCosTerm = dot(D,N)
         if (fCosTerm <= 0) continue
         if (GeomIntersect(P,D)) continue
         T += EvalBF(D)*CosTerm
      T *= NormFac
```



## Compute on GPU

SIGGRAPH2005

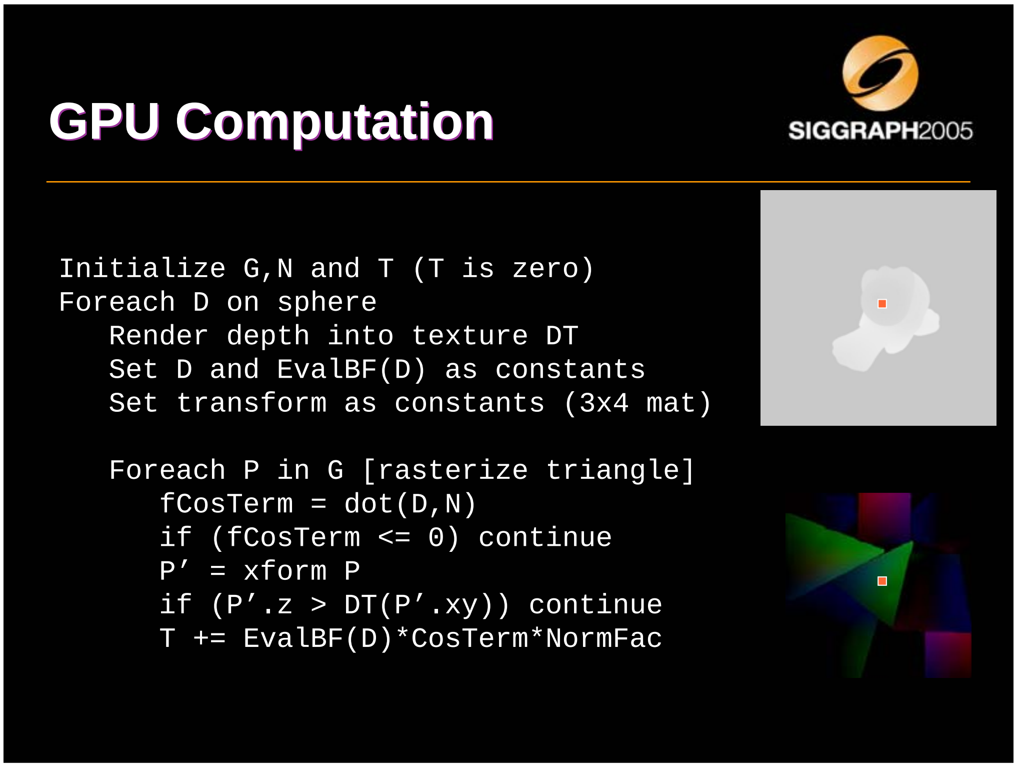
- Doesn’t map well to GPU
  - Ray tracing inefficient
  - Sum reduction inefficient
    - Use of hemicubes for NI would also not pipeline as well
- Reverse order of loops, accumulate contributions for a given direction to all samples in parallel



## GPU Setup

SIGGRAPH2005

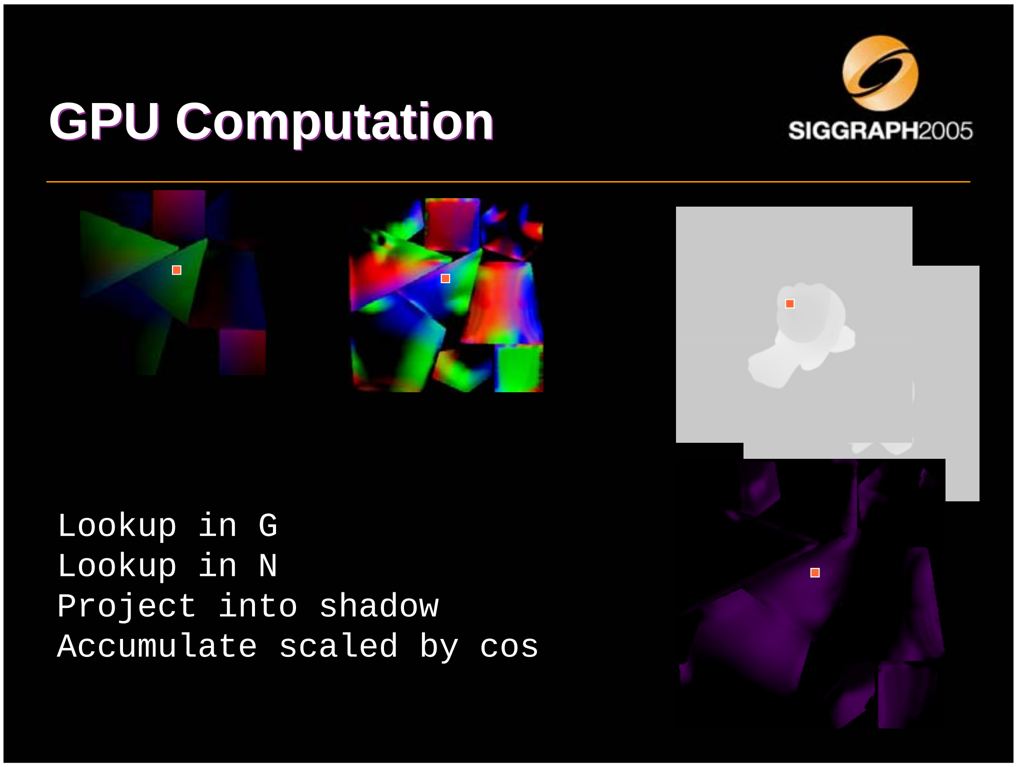
- Compute 2 textures
  - G geometry texture – pack vertices into texture (coherent) or use parameterization
  - N normal texture – 1 to 1 correspondence with G
- Create 3 MRT texture stacks of same resolution (high precision – this will be T)



## GPU Computation

SIGGRAPH2005

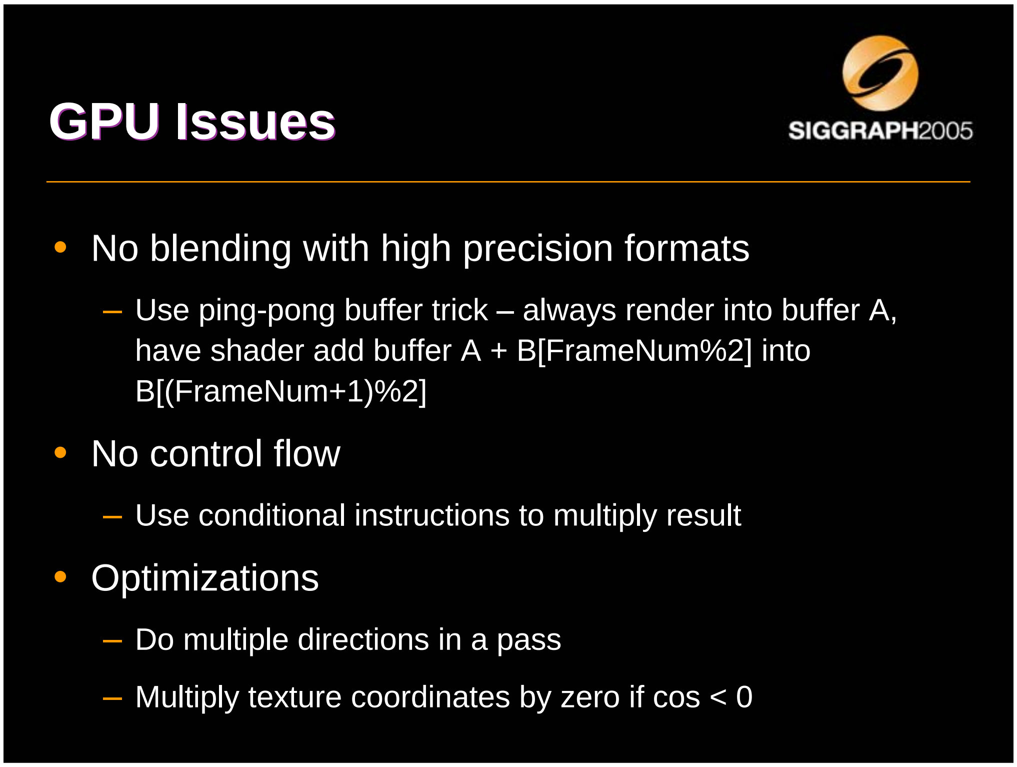
```
Initialize G,N and T (T is zero)
Foreach D on sphere
   Render depth into texture DT
   Set D and EvalBF(D) as constants
   Set transform as constants (3x4 mat)

   Foreach P in G [rasterize triangle]
      fCosTerm = dot(D,N)
      if (fCosTerm <= 0) continue
      P’ = xform P
      if (P’.z > DT(P’.xy)) continue
      T += EvalBF(D)*CosTerm*NormFac
```



## GPU Computation

SIGGRAPH2005

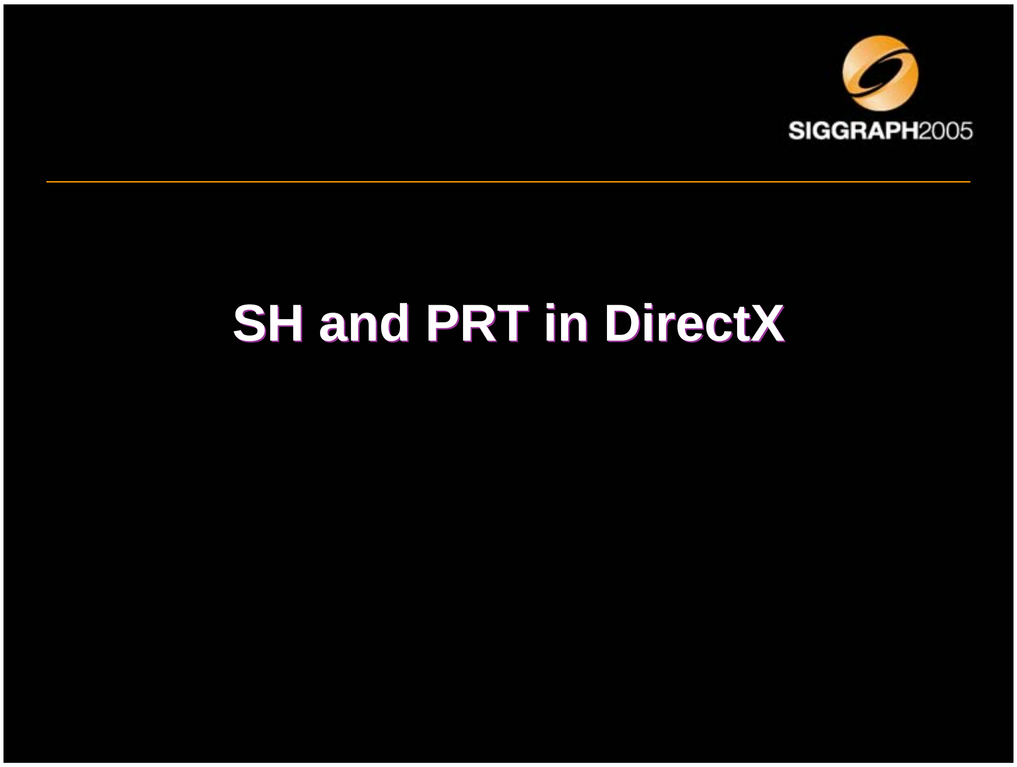
```
Lookup in G
Lookup in N
Project into shadow
Accumulate scaled by cos
```



## GPU Issues

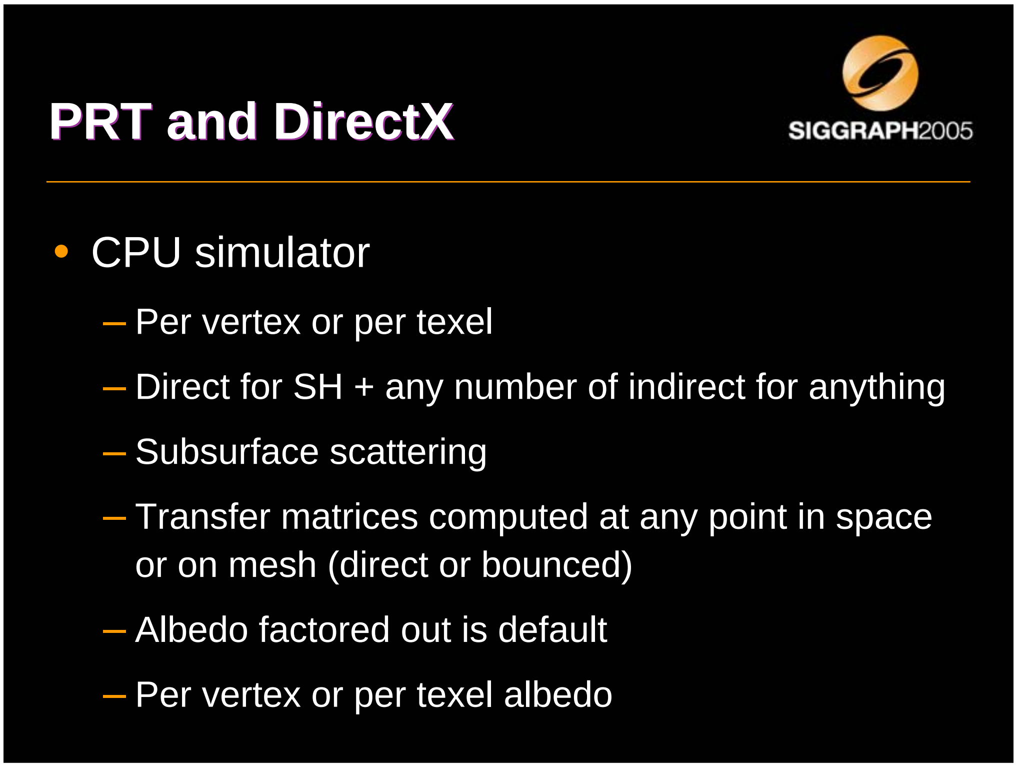
SIGGRAPH2005

- No blending with high precision formats
  - Use ping-pong buffer trick – always render into buffer A, have shader add buffer A + B[FrameNum%2] into B[(FrameNum+1)%2]
- No control flow
  - Use conditional instructions to multiply result
- Optimizations
  - Do multiple directions in a pass
  - Multiply texture coordinates by zero if cos < 0



SIGGRAPH2005

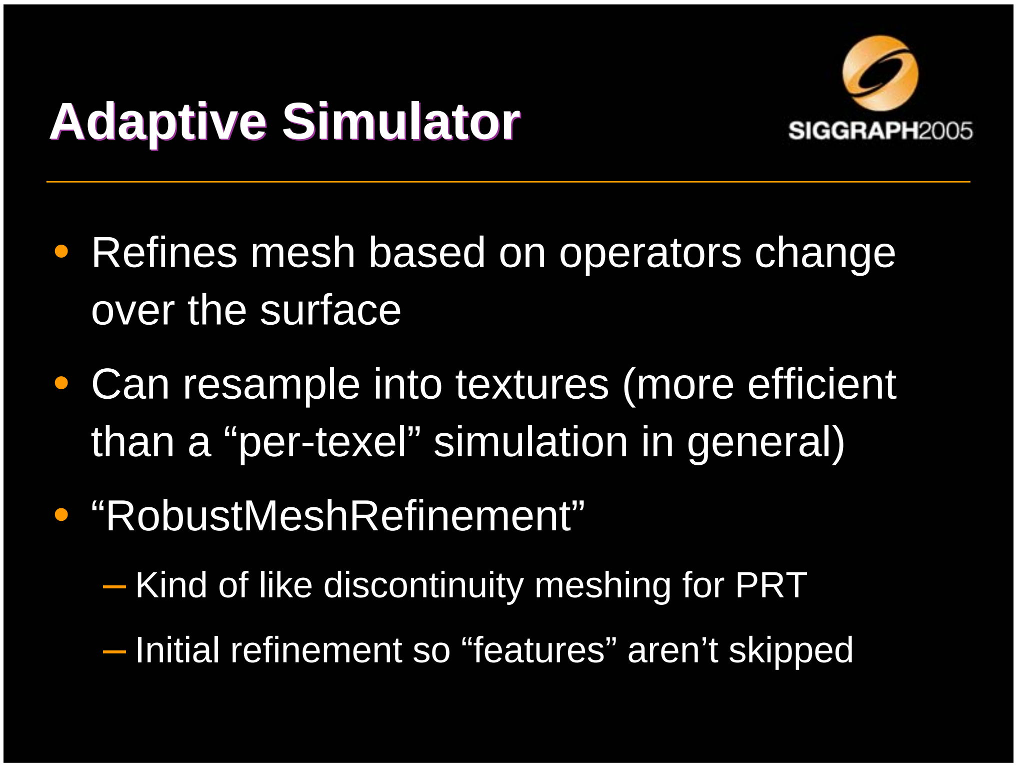
# SH and PRT in DirectX



## PRT and DirectX

SIGGRAPH2005

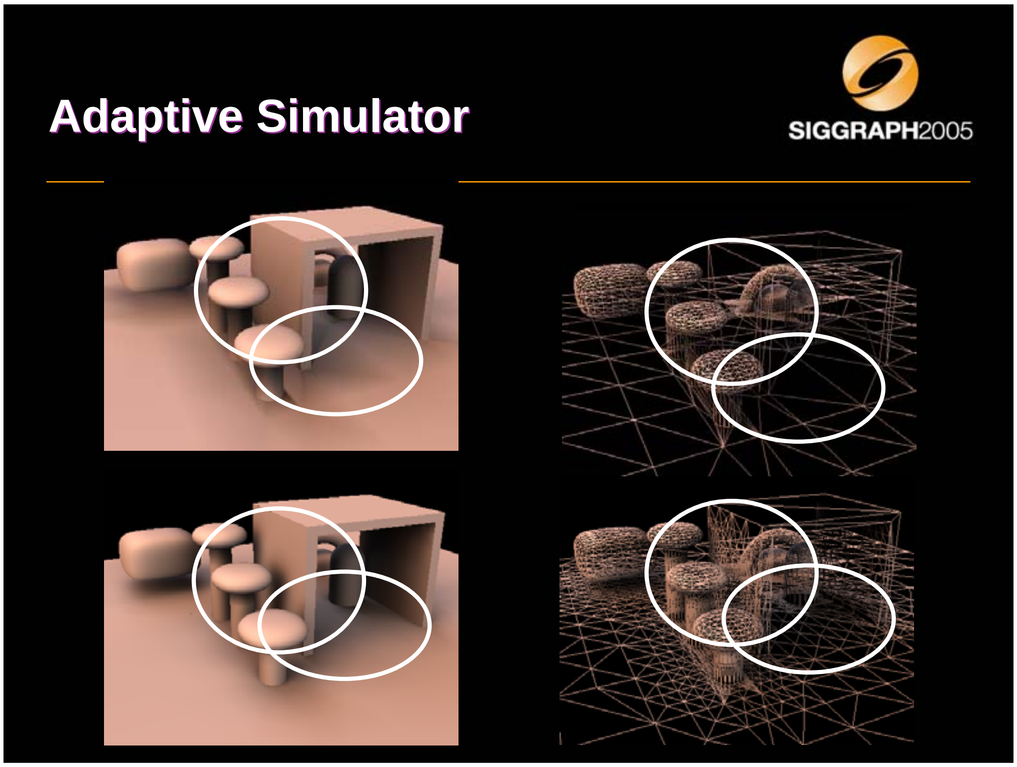
- CPU simulator
  - Per vertex or per texel
  - Direct for SH + any number of indirect for anything
  - Subsurface scattering
  - Transfer matrices computed at any point in space or on mesh (direct or bounced)
  - Albedo factored out is default
  - Per vertex or per texel albedo



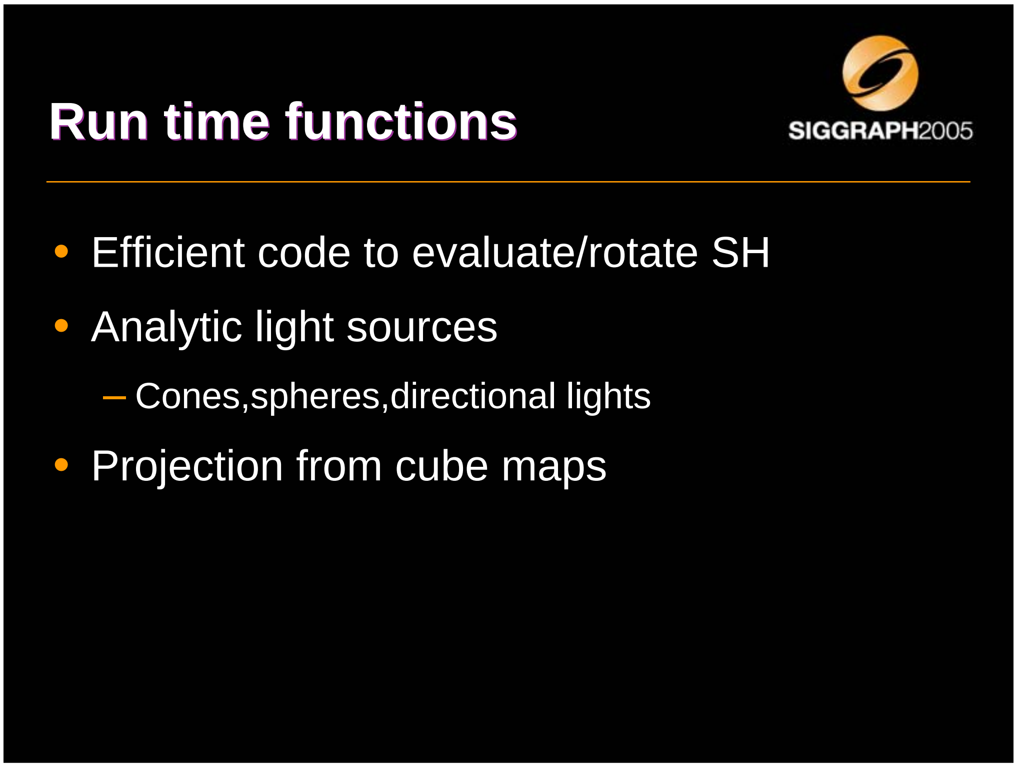
## Adaptive Simulator

SIGGRAPH2005

- Refines mesh based on operators change over the surface
- Can resample into textures (more efficient than a “per-texel” simulation in general)
- “RobustMeshRefinement”
  - Kind of like discontinuity meshing for PRT
  - Initial refinement so “features” aren’t skipped



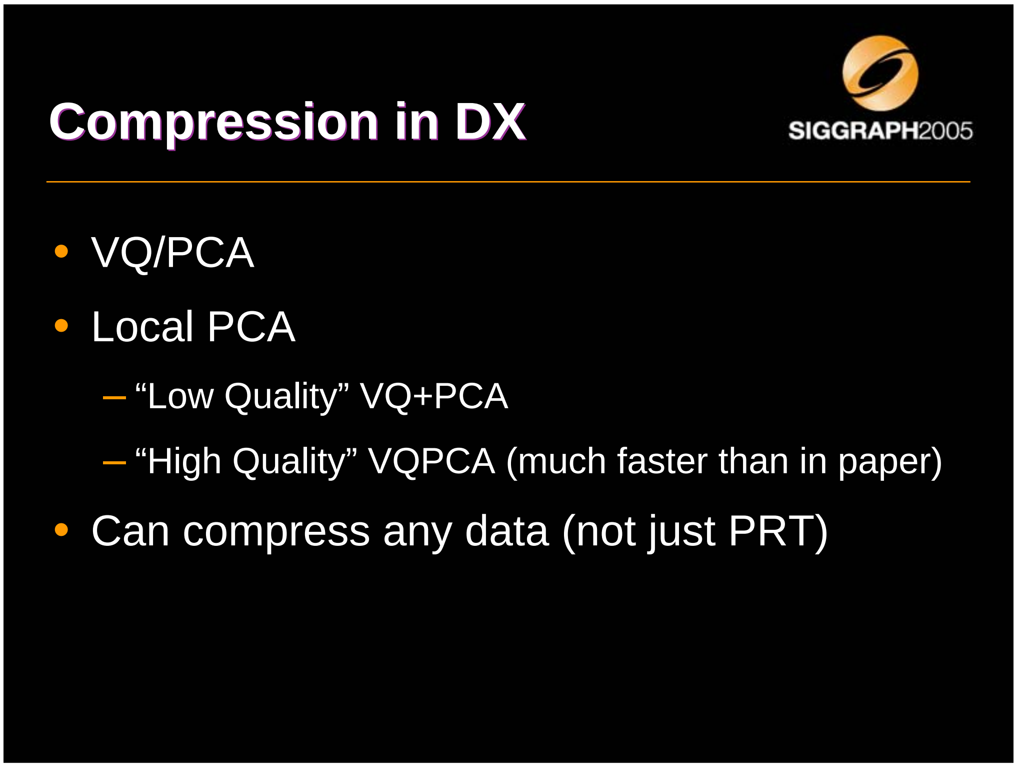
## Adaptive Simulator



## Run time functions

SIGGRAPH2005

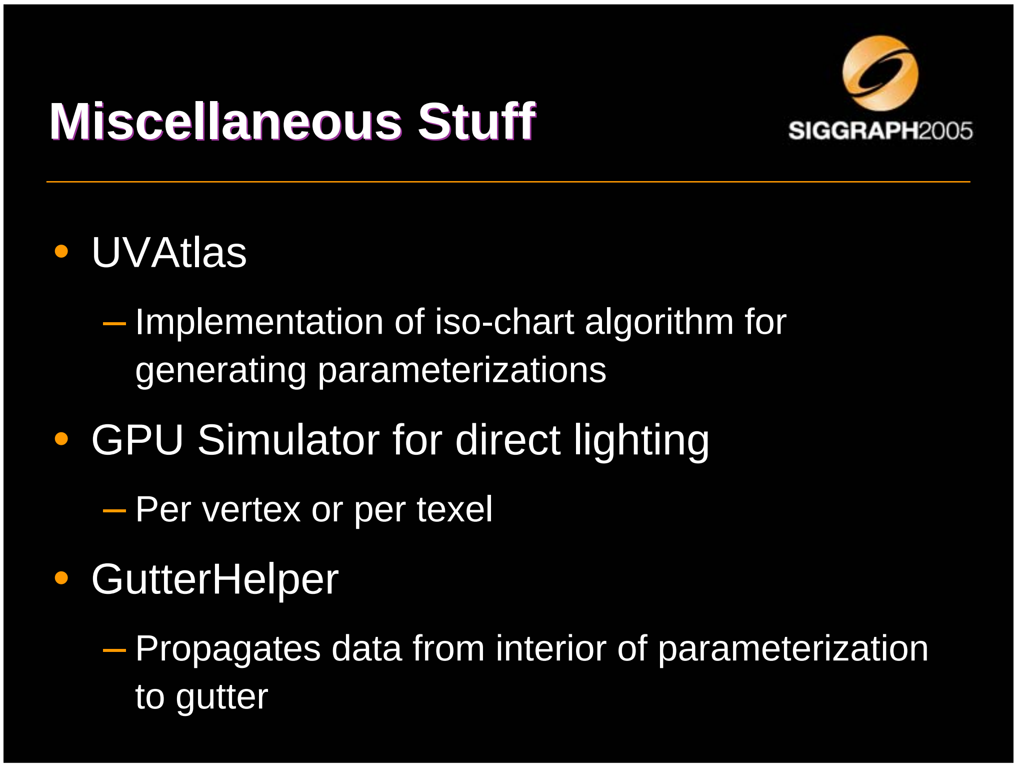
- Efficient code to evaluate/rotate SH
- Analytic light sources
  - Cones,spheres,directional lights
- Projection from cube maps



## Compression in DX

SIGGRAPH2005

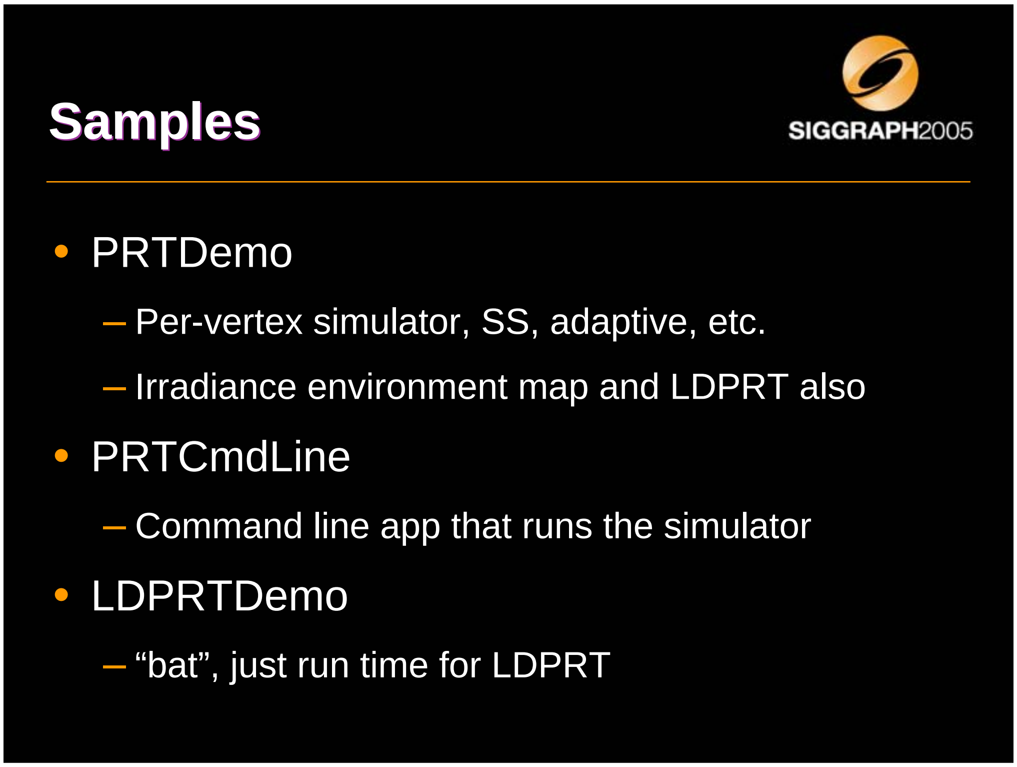
- VQ/PCA
- Local PCA
  - “Low Quality” VQ+PCA
  - “High Quality” VQPCA (much faster than in paper)
- Can compress any data (not just PRT)



## Miscellaneous Stuff

SIGGRAPH2005

- UVAtlas
  - Implementation of iso-chart algorithm for generating parameterizations
- GPU Simulator for direct lighting
  - Per vertex or per texel
- GutterHelper
  - Propagates data from interior of parameterization to gutter



## Samples

SIGGRAPH2005

- PRTDemo
  - Per-vertex simulator, SS, adaptive, etc.
  - Irradiance environment map and LDPRT also
- PRTCmdLine
  - Command line app that runs the simulator
- LDPRTDemo
  - “bat”, just run time for LDPRT

SIGGRAPH2005

demo

DX Demo

Run through the DX demos.

SIGGRAPH2005