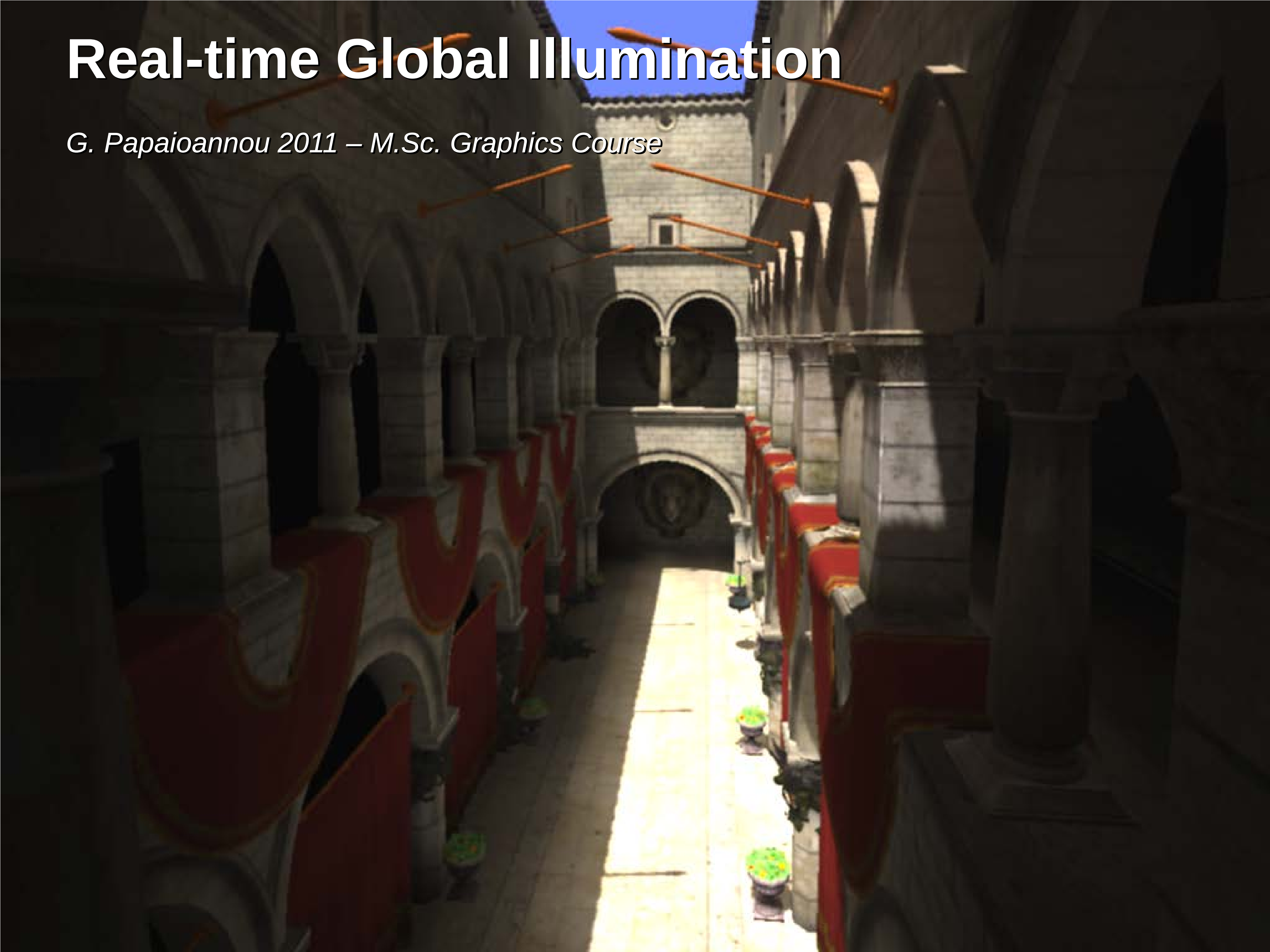
Real-time Global Illumination

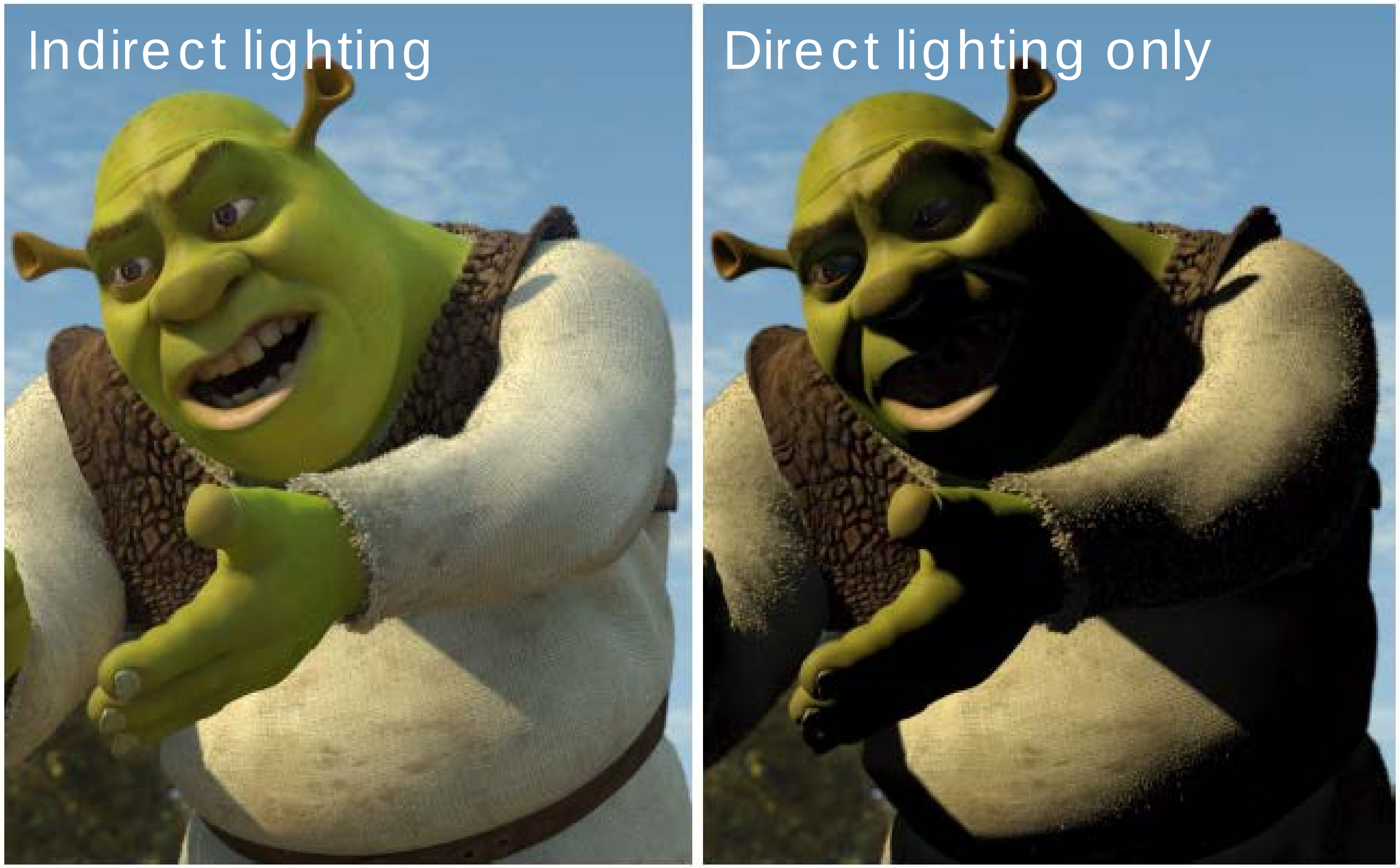
G. Papaioannou 2011 – M.Sc. Graphics Course



Contents

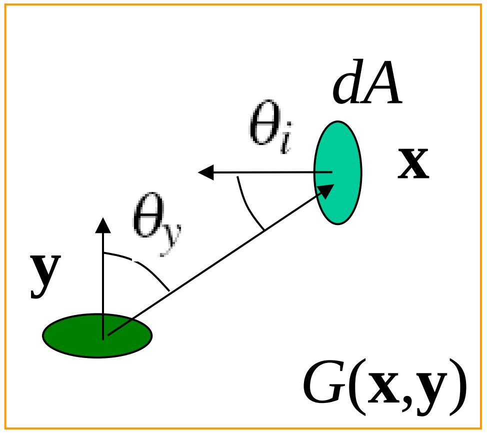
Why Use GI Algorithms?

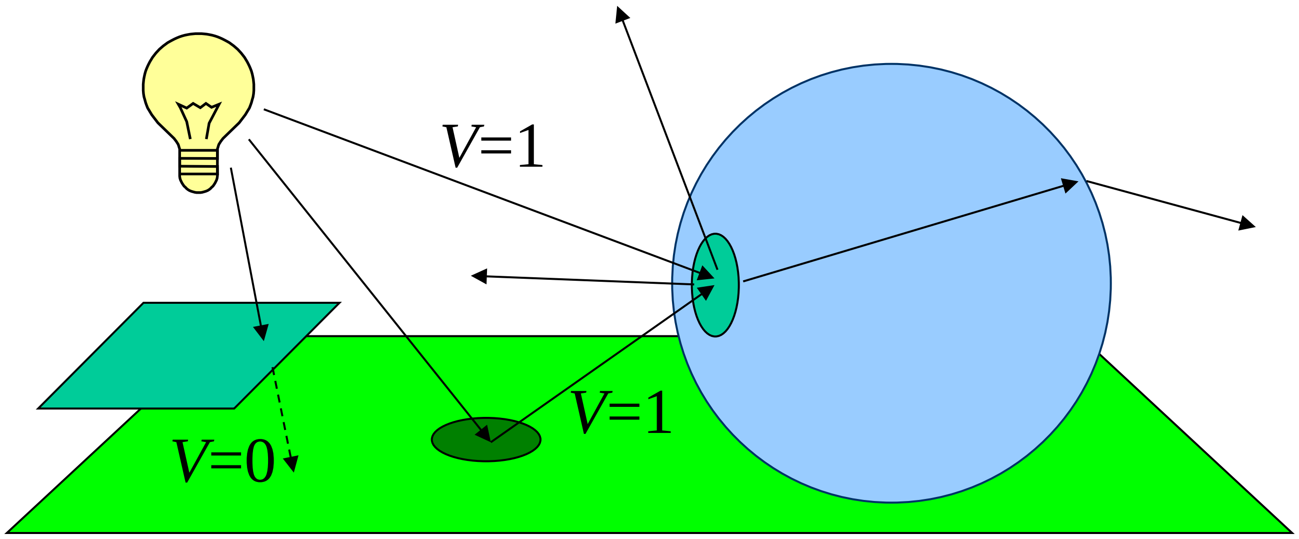
- Photorealistic simulation of illumination



The Rendering Equation

- Expresses the equilibrium of light distribution in a scene





$$L_r(\mathbf{x},\phi_r,\theta_r) = L_e(\mathbf{x},\phi_r,\theta_r) + \int_S L_r(\mathbf{y},\phi_y,\theta_y) f_r(\phi_r,\theta_r,\phi_i,\theta_i) G(\mathbf{x},\mathbf{y}) V(\mathbf{x},\mathbf{y}) dA$$

Non-real-time Approximations to GI

- The rendering equation must be solved simultaneously for all possible light paths in the environment
  - Unrealistic and non-feasible
  - Infinite light paths of uncertain importance
- Approximate solutions:
  - Discretize and sample space to generate a manageable set of light paths
  - Keep only paths that reach the image pixels
  - Rely on robust stochastic models to create unbiased results (Monte Carlo, Russian roulette, Metropolis) or
  - Use biased, light caching techniques (Photon maps)

Non-real-time GI Results

Bidirectional path tracing

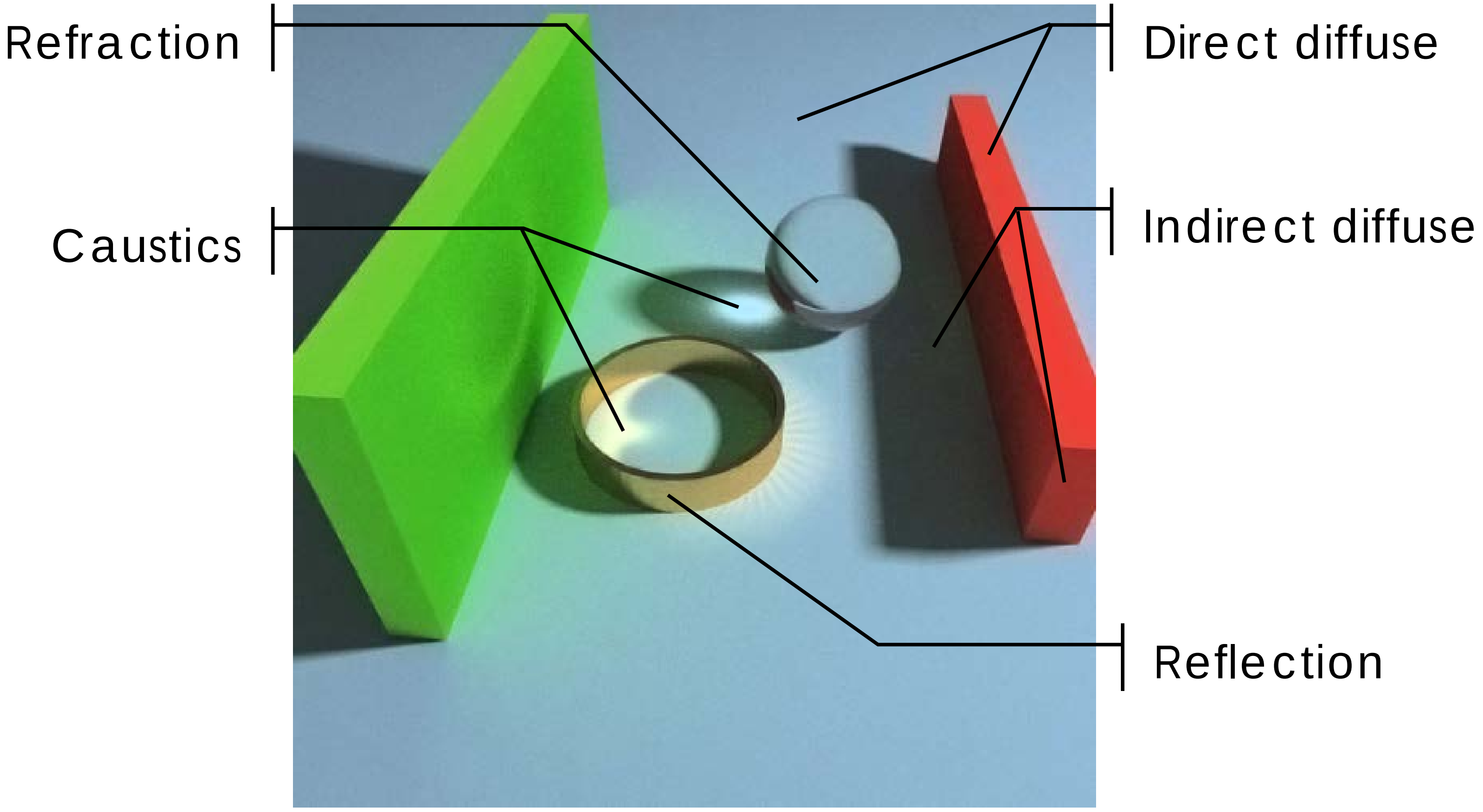
Photon mapping

Lighting components

- For computational efficiency and accuracy light paths are distinguished according to:
  - **Direct lighting**: unobstructed light from sources → Dense, directional sampling of visible portions of emitters
  - **Indirect diffuse**: Main scattering of light in environment (ambient light)
  - Specular **reflections** and **refracted** light
  - Specular-to-diffuse light bounces (e.g. **caustics**)
- Except from direct lighting, all other types of transmission are hard to tackle in real-time

Lighting components

- For computational efficiency and accuracy light paths are computed separately:



Part A: Real-time Rendering Techniques

Rendering to a 2D texture

Multiple render targets

Deferred rendering

Layer re-targeting

Rendering to a volume (3D) texture

Point injection

Multi-resolution rendering

Rendering to a 2D texture (1)

- Conventional direct rendering pipeline:
  - Output of fragment processing operations to the **frame memory buffer**
- Modern techniques require the output of the fragments into intermediate memory:
  - To post-process the results
  - To use the rendered image as input to the next rendering algorithm (as a texture, e.g. reflections, shadow maps etc).
  - To randomly access the stored values
  - To stream the output to another application

Rendering to a 2D texture (2)

- Modern graphics cards and APIs can redirect graphics output to custom frame buffers that write directly in textures (images)
- Steps:
  - Prepare (allocate) a 2D texture
  - Prepare a frame buffer object
  - Link the 2D texture with one of the frame buffer attachment attributes (color/depth)
  - Enable the frame buffer object as current graphics output

Rendering to a 2D texture (3)

- In OpenGL:

```
Gluint buffer, FBO;

glGenTextures(1,&buffer);
glBindTexture(GL_TEXTURE_2D, buffer);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, width, height, 0,
    GL_RGBA, GL_UNSIGNED_BYTE, NULL);

glGenFramebuffersEXT(1, &FBO);
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT,FBO);
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
    GL_COLOR_ATTACHMENT0_EXT, GL_TEXTURE_2D, buffer, 0);
```

Internal format

Attachment

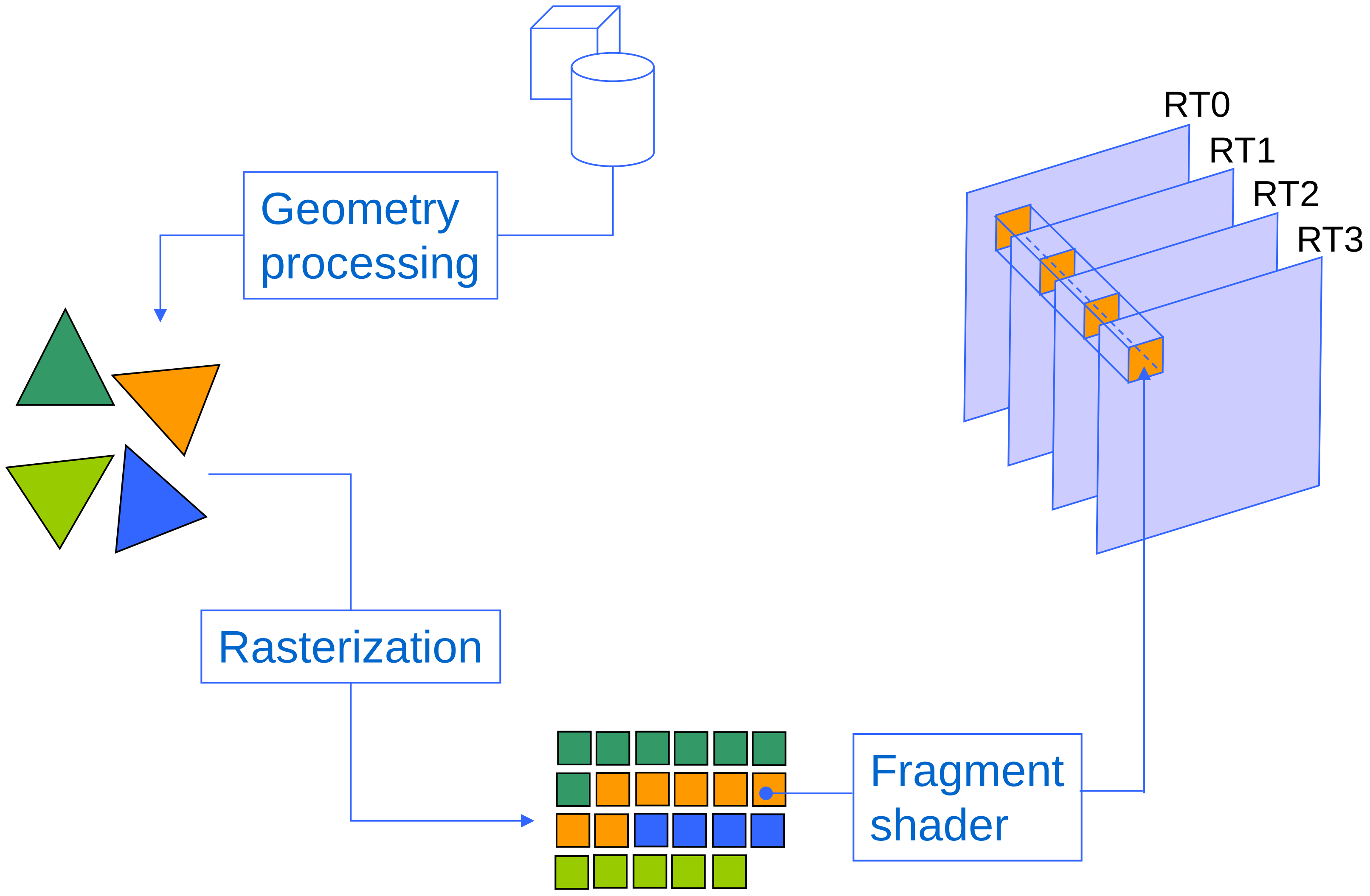
Multiple Render Targets (1)

- It is often useful to be able to write many fragment operation results to **multiple internal buffers, without re-rendering the geometry**
- Examples:
  - Cube map generation (6 buffers, 6 viewing transformations – also requires retargeting by a geometry shader)
  - Deferred rendering (3+ buffers, one viewing transformation)
  - Reflective shadow maps (ok, this is still deferred rendering!)

Multiple Render Targets (2)

- This is enables via the Multiple Render Targets (MRT) mechanism:
  - The geometry is sent once for primitive generation
  - The pixel (fragment) shader writes results at the same location on multiple buffers
  - Different calculations and hence output values can be written to each buffer in the same pixel shader

Multiple Render Targets (3)



Multiple Render Targets (4)

- OpenGL initialization:

```
GLuint FBO, buffer[4]; // up to 8 for now.
glGenTextures(4,buffer);
glGenFramebuffersEXT(1, &FBO);
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT,FBO);

glFramebufferTexture2DEXT( GL_FRAMEBUFFER_EXT,
    GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, buffer[0],0 );
glFramebufferTexture2DEXT( GL_FRAMEBUFFER_EXT,
    GL_COLOR_ATTACHMENT1, GL_TEXTURE_2D, buffer[1],0 );
glFramebufferTexture2DEXT( GL_FRAMEBUFFER_EXT,
    GL_COLOR_ATTACHMENT2, GL_TEXTURE_2D, buffer[2],0 );
glFramebufferTexture2DEXT( GL_FRAMEBUFFER_EXT,
    GL_COLOR_ATTACHMENT3, GL_TEXTURE_2D, buffer[3],0 );
```

Multiple Render Targets (5)

- OpenGL usage:

```
GLenum targets[4] =
  { GL_COLOR_ATTACHMENT0_EXT, GL_COLOR_ATTACHMENT1_EXT,
    GL_COLOR_ATTACHMENT2_EXT, GL_COLOR_ATTACHMENT3_EXT };

glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, point_fbo);

If (glCheckFramebufferStatusEXT(GL_FRAMEBUFFER_EXT)
    !=GL_FRAMEBUFFER_COMPLETE_EXT)
    {
        // Failed to initialize the FBO. Handle the error here
    }

glDrawBuffers(4,targets);
```

Multiple Render Targets (6)

- And in the GLSL shader, you simply write the data to the appropriate buffer:

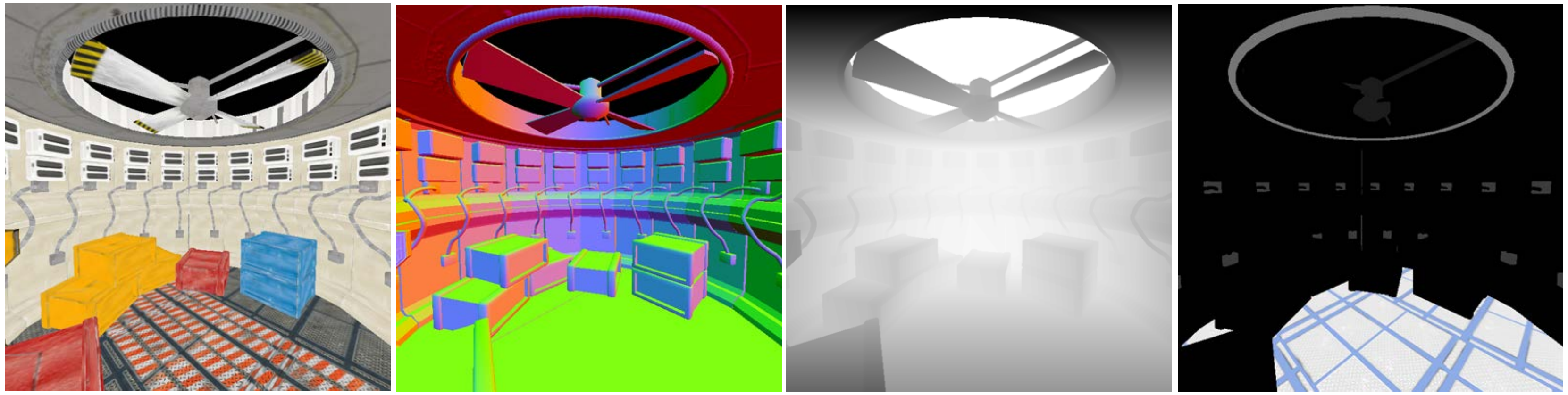
```
void main()
{
    ... // other fragment shader code
    gl_FragData[0] = vec4(...);
    gl_FragData[1] = vec4(...);
    gl_FragData[2] = vec4(...);
    gl_FragData[3] = vec4(...);
}
```

Deferred Rendering (1)

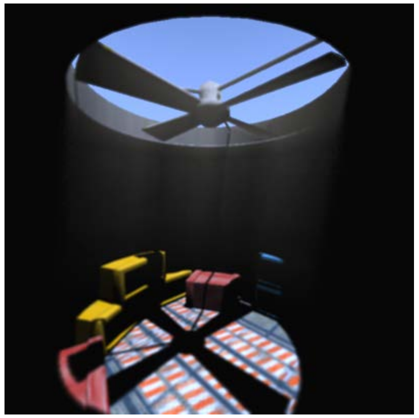
- In deferred rendering, the geometry is not immediately rendered but instead, it is used for the generation of intermediate data, which are later used for calculating the final image
- The intermediate data are generated through the MRT mechanism in one pass
- All shading calculations are postponed for the final (deferred) stage
- Why?
  - **Expensive shading calculations are performed once per pixel** (visible fragments only)

Deferred Rendering (2)

- Typically, the albedo, the normals, the depth and specular attributes are written in MRTs (G-buffer)



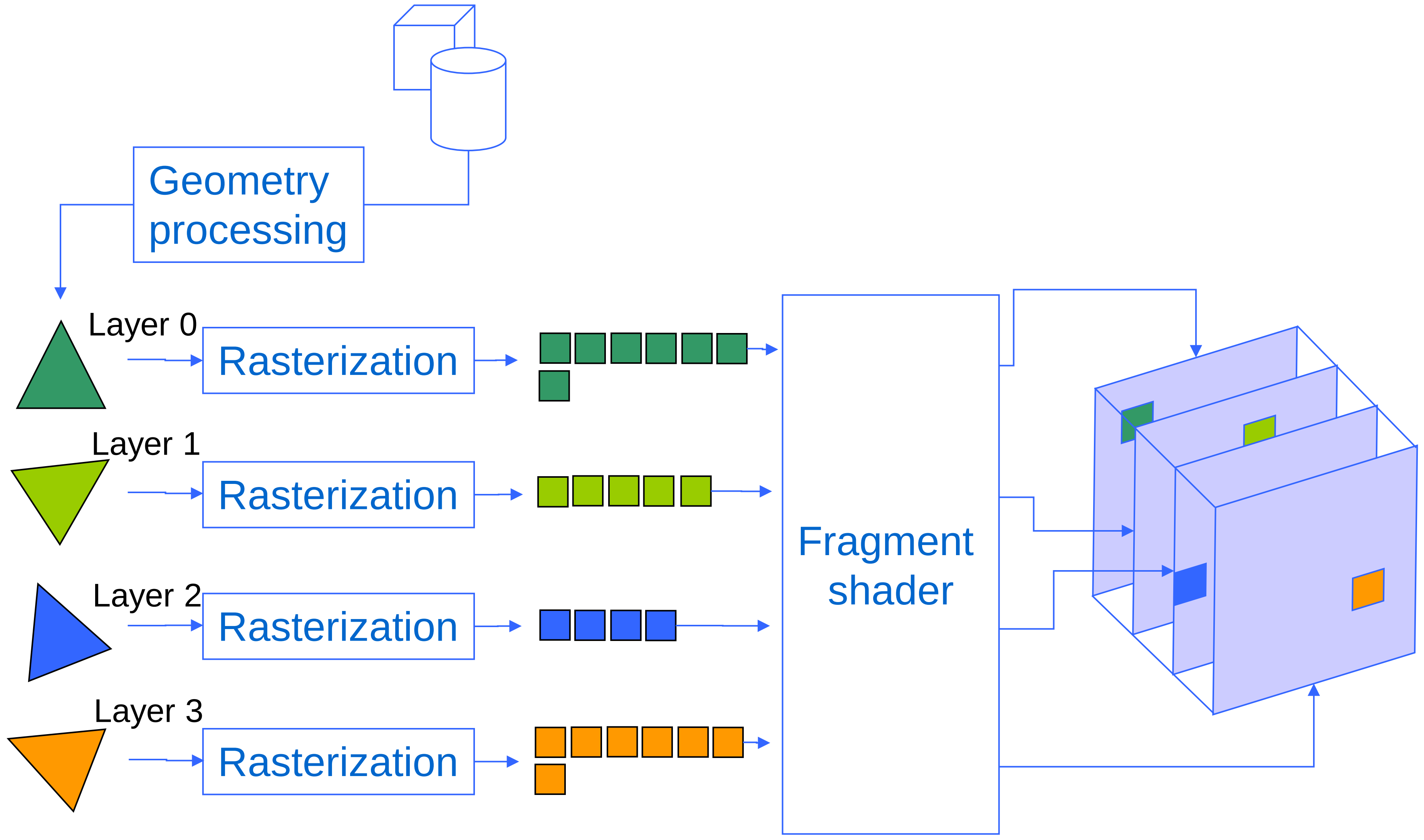
- The final shading uses the above buffers as textures to calculate illumination:



Texture Arrays and Rendering Layers

- We have seen that textures can be bound as frame buffers
- We can instruct the hardware to bind an array of textures as output of a single rendering target
- Each texture in the array is treated as a separate rendering layer
- The geometry shader can determine which layer to emit a primitive to
- This technique can be combined with MRT rendering

Rendering Layers



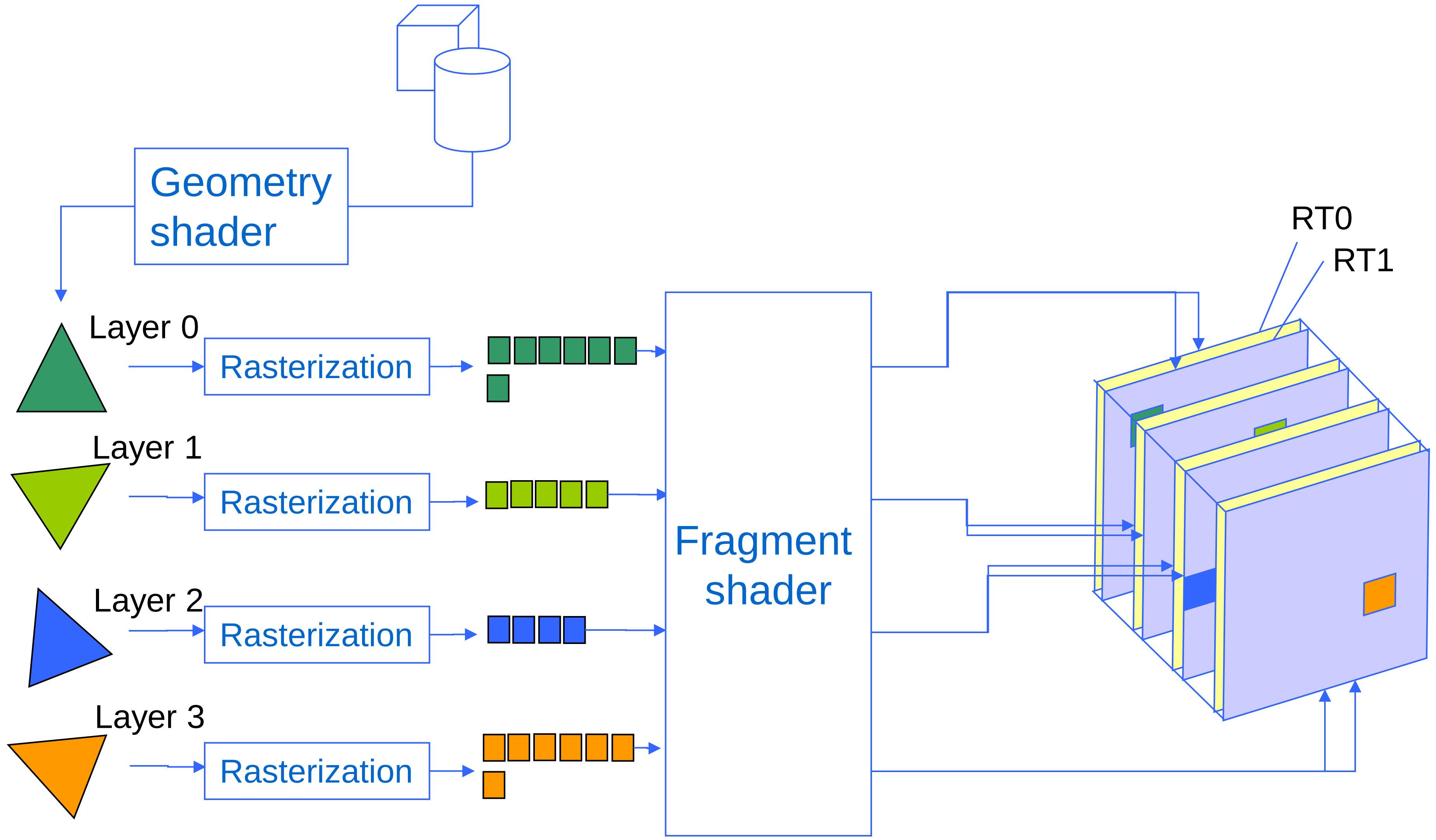
Layers vs MRTs

- Layers:
  - The geometry shader **selects** a buffer and emits a primitive for rasterization to it
  - A primitive can be generated and emitted to any number of layers
  - Each layer selection and primitive emission adds a new rasterization task to the primitve queue
  - Generated primitive fragments are unrelated across layers
- MRTs:
  - The fragment shader **simultaneously writes** data to all MRTs
  - Number of RTs is predetermined
  - Fragment coordinates (x,y) are identical to all RTs

Layers using MRTs (1)

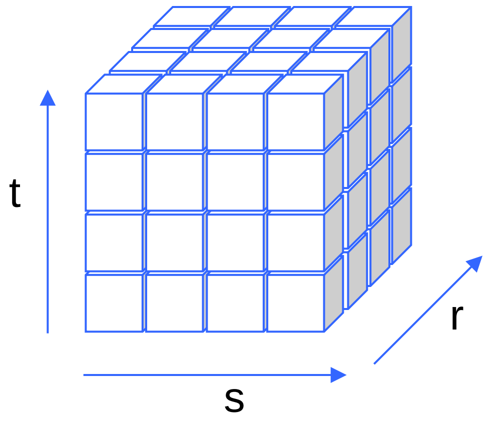
- Layers and MRTs can be combined!
- We can enable both. Essentially, we can have multiple layers, each one with multiple render targets
- You can think of the extra RTs as extra channels in a texel (multiples of base type, e.g. 4XRGBA)
- Each layer is a separate multichannel canvas
- We decide which primitive to submit for rendering to which canvas (and how).

Layers using MRTs (2)



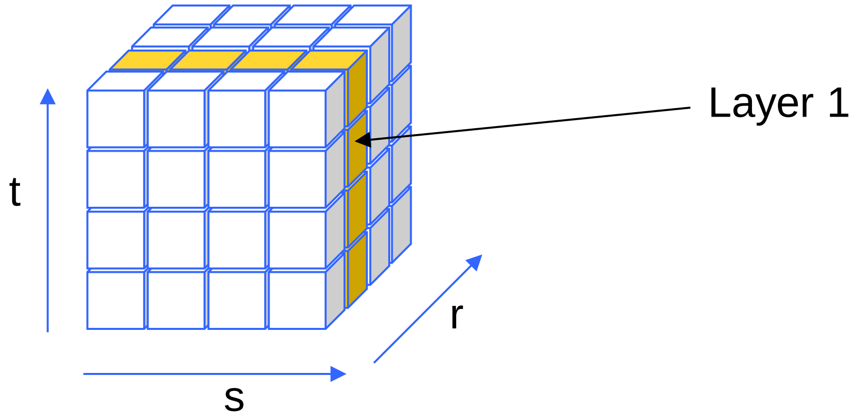
Volume Textures

- Volume textures are packed arrays of equally sized 2D textures, slice by slice
- They are different from 2D texture arrays:
  - They are indexed by 3 normalized params (s,t,r)
  - They can be trilinearly filtered. Texture arrays are not interpolated across different slices
  - They are also accessible from fixed graphics pipeline



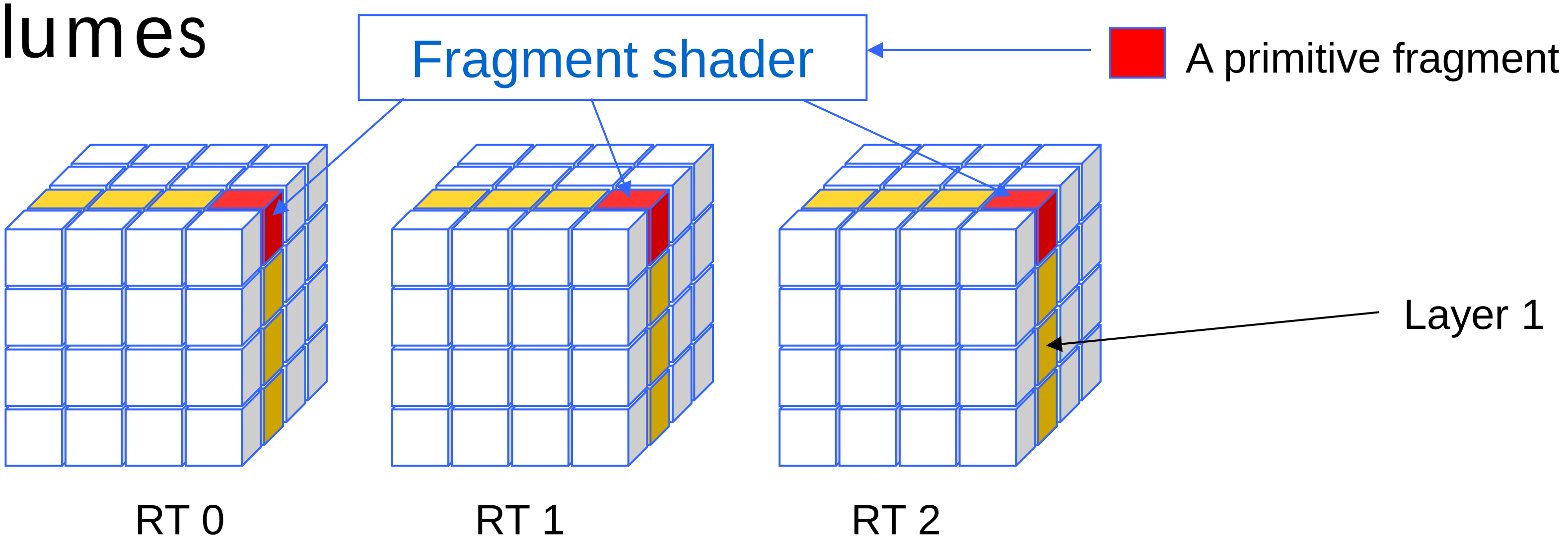
Rendering into 3D Textures (1)

- To directly render into a 3D texture, we can bind a frame buffer object to it
- Each slice of the 3D texture is treated as a frame buffer attachment and indexed as a separate layer
- The geometry shader redirects output of a primitive to one or more depth layers



Rendering into 3D Textures (2)

- 3D textures can be also used as MRTs
- Each (identical) 3D texture can be bound to a different FBO attachment
- Each primitive is submitted for rendering into a specific layer, where its fragments update the corresponding pixels of the same layer in all MRT volumes



Rendering into 3D Textures (3)

- OpenGL Initialization:

```
GLuint fbo, buffers[4];
glGenFramebuffersEXT(1, &fbo);
glGenTextures(4, buffers);
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT,fbo);

glBindTexture(GL_TEXTURE_3D, buffers[0]);
glTexImage3D(GL_TEXTURE_3D, 0, GL_RGBA16F, resx,resy,resz, 0,
    GL_RGBA, GL_HALF_FLOAT, NULL);
glFramebufferTexture3DEXT( GL_FRAMEBUFFER_EXT,
    GL_COLOR_ATTACHMENT0, GL_TEXTURE_3D, buffers[0], 0, 0 );
... // do the same for other textures as well
GLenum targets[4] =
  { GL_COLOR_ATTACHMENT0_EXT, GL_COLOR_ATTACHMENT1_EXT,
    GL_COLOR_ATTACHMENT2_EXT, GL_COLOR_ATTACHMENT3_EXT };
glDrawBuffers(4,targets);
```

Rendering into 3D Textures (4)

- GLSL geometry shader:

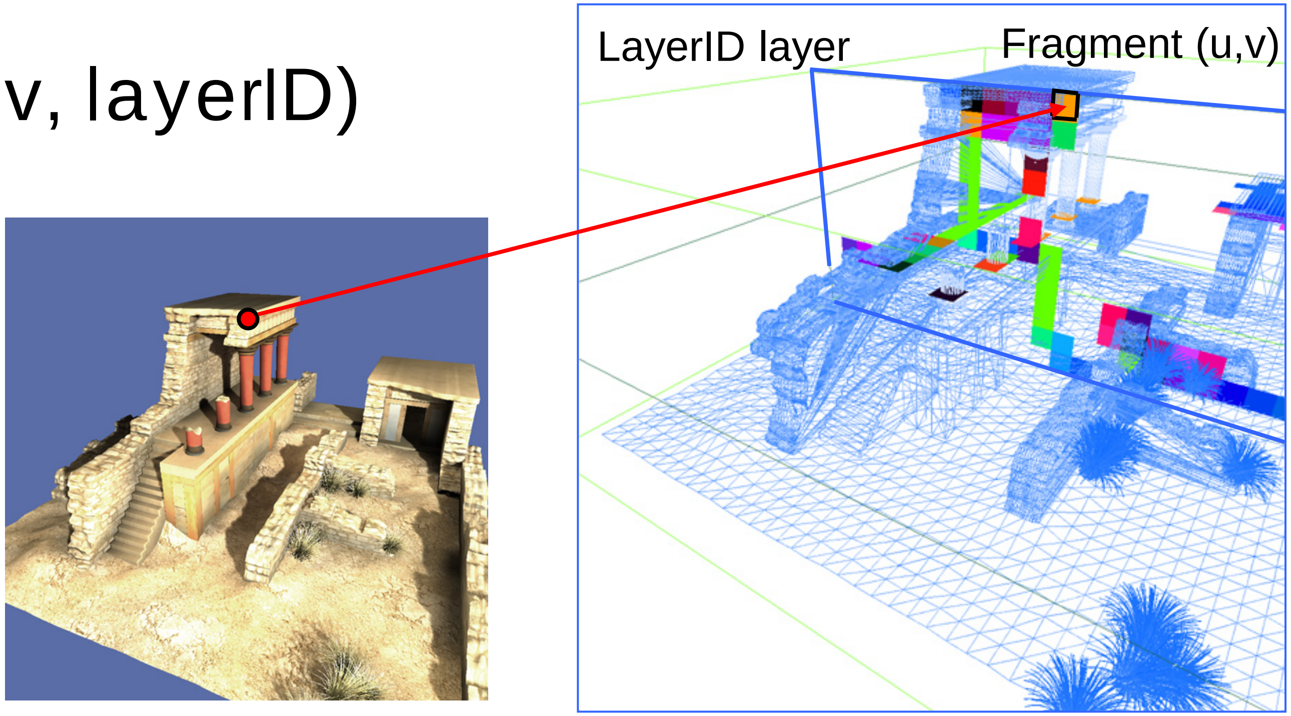
```
// Example:
//  Point rendering. Incoming points are redirected for rendering
//  to a 3D volume slice according to relative z-value in (minz,maxz)
uniform vec3 pmin, pmax;
void main()
{
    int layer = 32*floor((gl_PositionIn[0].z-pmin.z)/(pmax.z-pmin.z));
    gl_Position = gl_PositionIn[0];
    gl_Layer = layer;
    EmitVertex();
}";
```

Point Rendering

- Point rendering is the drawing of a (dense) cloud of points to substitute surface geometry
- "Points" may occupy more than one fragment (depending on the point size)
- Dense point clouds can effectively replace complex geometry at a moderate cost
- Sparse point clouds can be used in algorithms that require only a general spatial "geometry distribution" in the scene.
- Many modern GI algorithms depend on point injection (rendering) in volume textures.

Point Injection

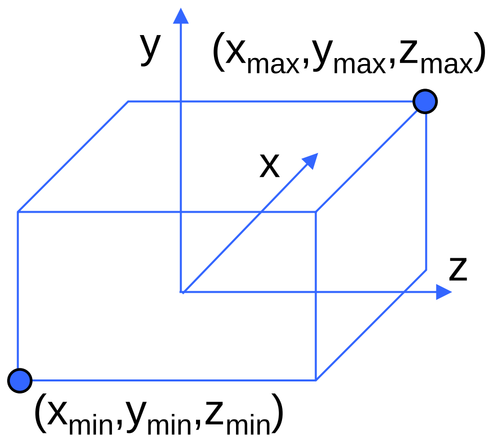
- Is the process of placing point samples inside a volume that represents the spatial extents of a 3D scene
- It is implemented via the volume layer mechanism:
- P=(x,y,z)→(u,v, layerID)



Point Injection – The Volume

- Usually, a grid represented as a 3D texture is defined covering the bounding box (extents) of the scene
- Then a scale and translation transform the coordinates inside the bounding box to the normalized 3D texture coordinates:

$$\mathbf{M}_{Vol} = S_{\frac{1}{x_{\max}-x_{\min}},\frac{1}{y_{\max}-y_{\min}},\frac{1}{z_{\max}-z_{\min}}} T_{-x_{\min},-y_{\min},-z_{\min}}$$



- Finding the slice is easy:

$$p = (x, y, z). \qquad p_{Vol} = (u, v, w) = \mathbf{M}_{vol} \cdot p$$

$$layer = \left\lfloor w \cdot Volsize_z + \frac{1}{2} \right\rfloor$$

Point Injection - Implementation

- The volume point injection can be easily implemented in the geometry shader:
  - A grid of points or the vertices of the geometry as transformed according to $\mathbf{M}_{Vol}$
  - The layer is selected where the points will be emitted for rasterization
- Point coordinates are mostly derived from:
  - Raw (WCS) triangle vertices
  - Stored geometry images (textures encoding x,y,z coordinates as RGB data)
  - Un-projected points in a depth or shadow map
- Additional transformations may need to be applied before the injection procedure

Part B: Illumination Functions Compression

- Projection and reconstruction of signals
- Frequency analysis of light field
- Light and visibility as functions over the sphere
- Spherical harmonics
- Spherical radial basis functions
- Low-frequency illumination storage

Orthonormal Basis Functions

- A basis function b_n is an element of a particular basis for a function space
- Every continuous function in the function space can be represented as a linear combination of basis functions:

$$f(x) = \sum_{n \in N} a_n b_n(x)$$

- Check similarity with vector spaces
- An orthonormal basis additionally satisfies the property:

$$\int b_i b_j = \delta(i-j) \qquad \forall i, j \in \mathrm{N}$$

Signal Projection on Orthonormal Bases

- The projection of an arbitrary continuous function on a set of basis functions results in the definition of the blending coefficients a_n
- It can be proven that for orthonormal function bases, the best least squares fitting of a function f over a predefined set of basis functions b_n results in:

$$a_n = \int f(x) b_n(x) \mathrm{d}x$$

- (Again, relate this with the dot product projection in orthonormal bases for vector spaces)

Signal Reconstruction

- The number of basis (blending) functions may be infinite or too large and therefore we must choose a finite subset of them that converges “reasonably” to the desired result
- The reconstructed function (signal) is derived from the linear combination of the (truncated series) of basis functions:

$$\tilde{f}(x) = \sum_{n=1}^{N} a_n b_n(x)$$

Spherical Harmonics (1)

- Spherical Harmonics define an orthonormal basis over the sphere **S**.
- A point s on the sphere is parameterized as:

$$s = (x, y, z) = (\sin\theta\cos\varphi, \sin\theta\cos\varphi, \cos\theta)$$

- They are harmonic functions and more specifically they constitute the angular part of the solution of the Laplace's equation on the unit sphere:

$$\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} + \frac{\partial^2 f}{\partial z^2} = 0$$

Spherical Harmonics (2)

- The (complex) basis functions are defined as:

$$Y_l^m(\theta,\varphi) = K_l^m e^{im\varphi} P_l^{|m|}(\cos\theta), l \in \mathbf{N}, -l \leq m \leq l$$

where P_l^m are the associated Legendre polynomials and K_l^m are the following normalization factors:

$$K_l^m = \sqrt{\frac{(2l+1)(l-|m|)!}{4\pi(l+|m|)!}}$$

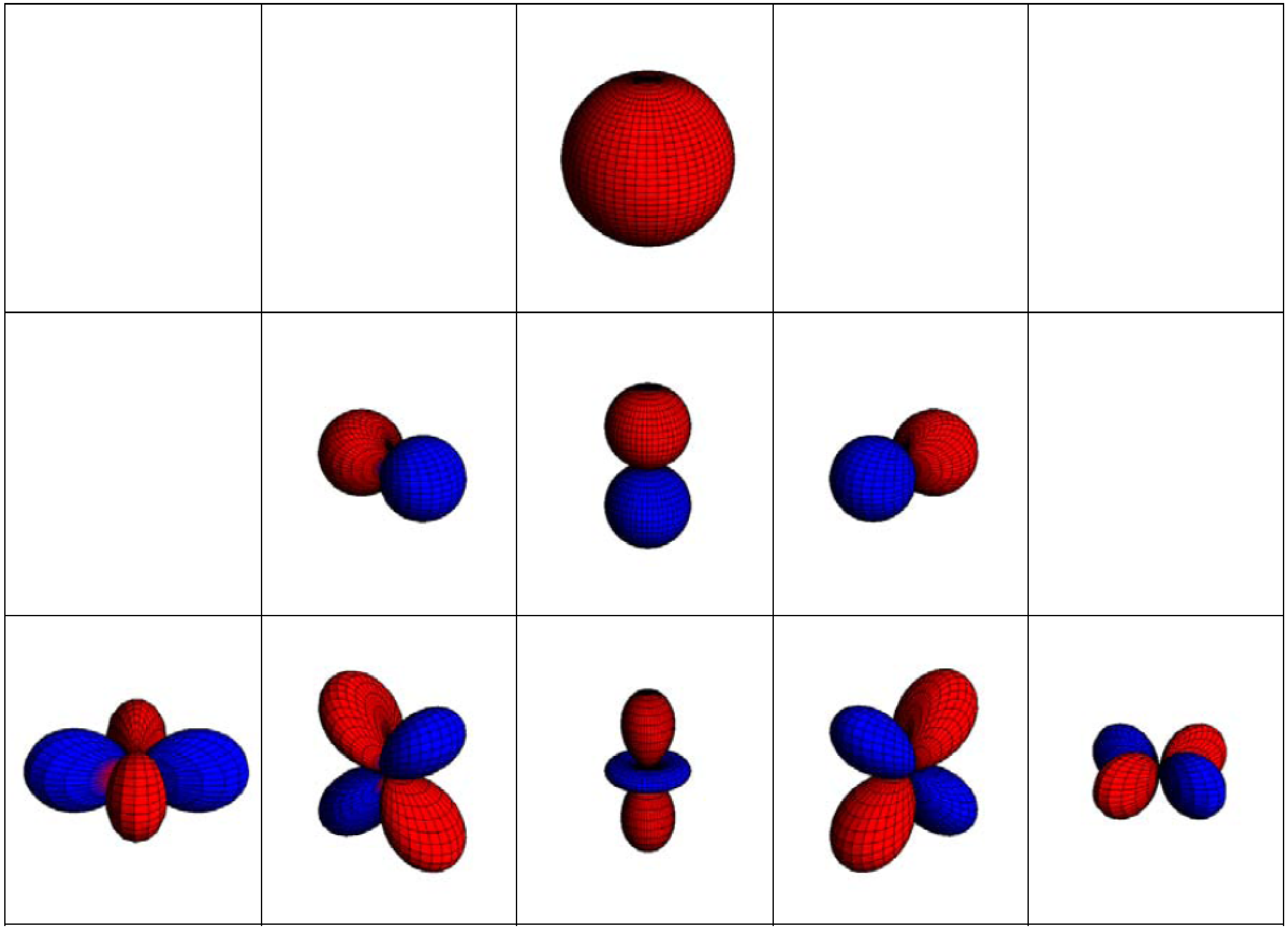
Spherical Harmonics (3)

- Real versions of the SH basis functions can be obtained from the transformation:

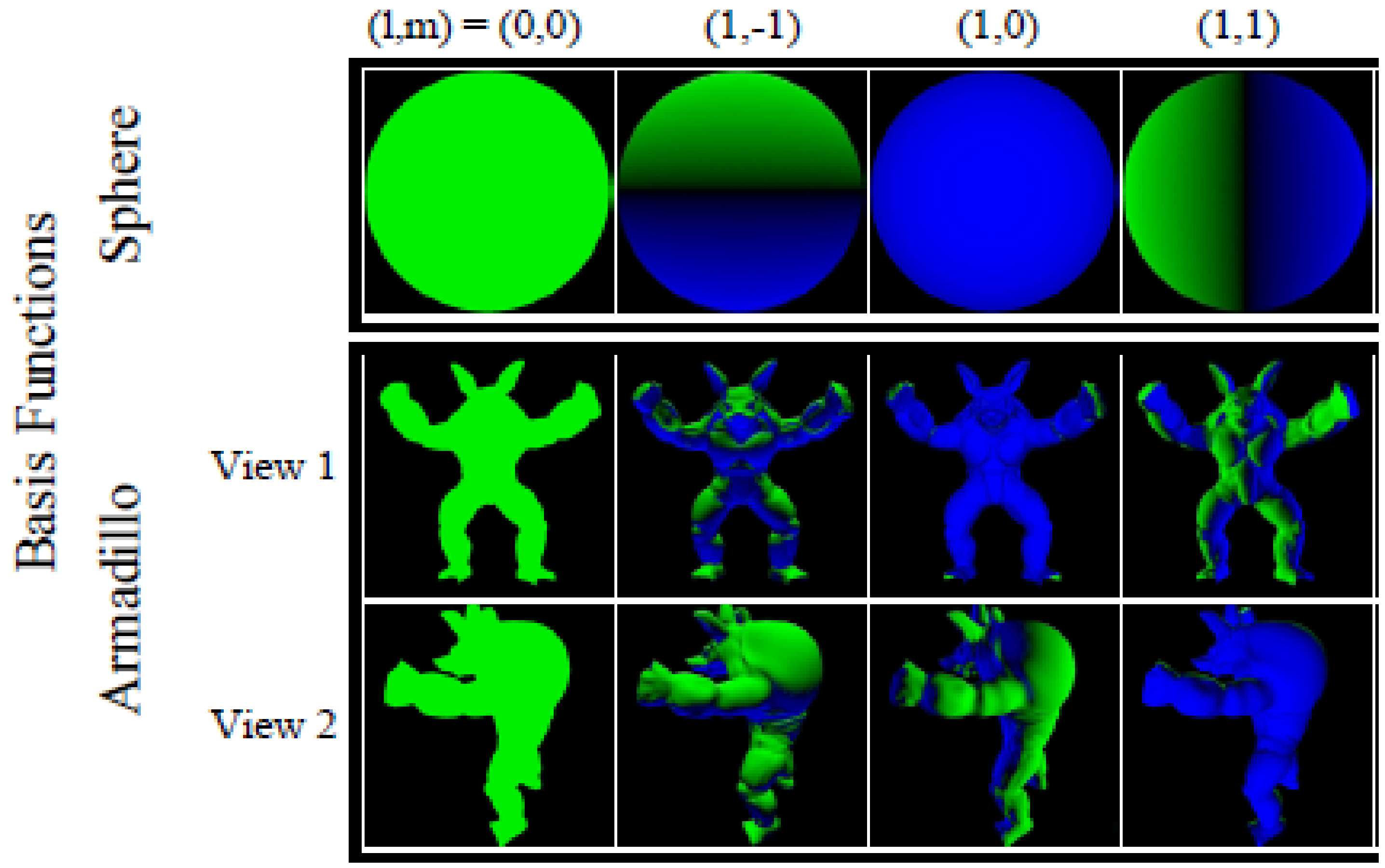
$$y_l^m = \begin{cases} \sqrt{2}\,\mathrm{Re}(Y_l^m) & m > 0 \\ \sqrt{2}\,\mathrm{Im}(Y_l^m) & m < 0 \\ Y_l^0 & m = 0 \end{cases} = \begin{cases} \sqrt{2}K_l^m \cos m\varphi \; P_l^m(\cos\theta) & m > 0 \\ \sqrt{2}K_l^m \sin|m|\varphi \; P_l^{|m|}(\cos\theta) & m < 0 \\ K_l^0 P_l^0(\cos\theta) & m = 0 \end{cases}$$

- l represents the band of the SH functions
- Each band has $2l+1$ SH basis functions
- Each band corresponds to an increasing angular frequency

Spherical Harmonics (4)



Spherical Harmonics (5)



Spherical Harmonics (6)

- Being an orthonormal set of basis functions:

$$f_l^m = \int f(s) y_l^m(s)\, ds$$

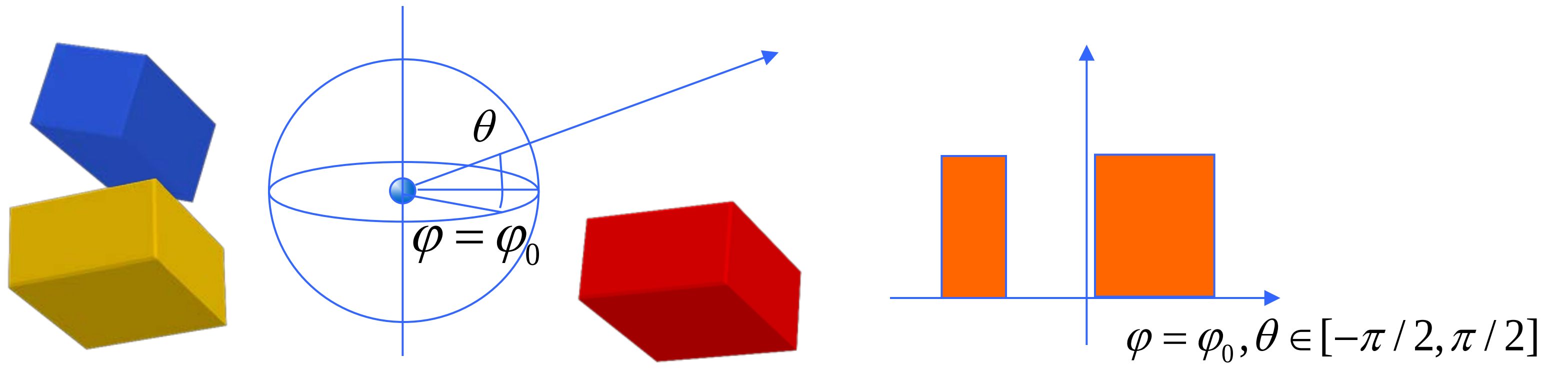
- The reconstruction of the signal can use up to any order of SH bands, truncating the infinite series of coefficients and respective basis functions
- Similarly, the encoded (projected) signal has to be band limited and encoded in a finite set of SH coefficients
- How many bands should we use?

Radiance Field

- In broad terms, radiance is the light power transmitted over a path connecting two points in space (see Advanced Shading Models presentation for a detailed definition)
- Incident or emitted radiance is parameterized as function of space and direction (5 DoF)
- Therefore, in its more general form, it can be represented as a 5D field
- What are the spectral characteristics of this field?

Visibility Field

- Similar to radiance, we can encode visibility as a 5D field:
  - What is the visibility (how open is the environment) at a point (x,y,z) in space in a direction (θ,φ)?
  - Encodes the ability of the specific point to receive light from an incident direction (θ,φ)

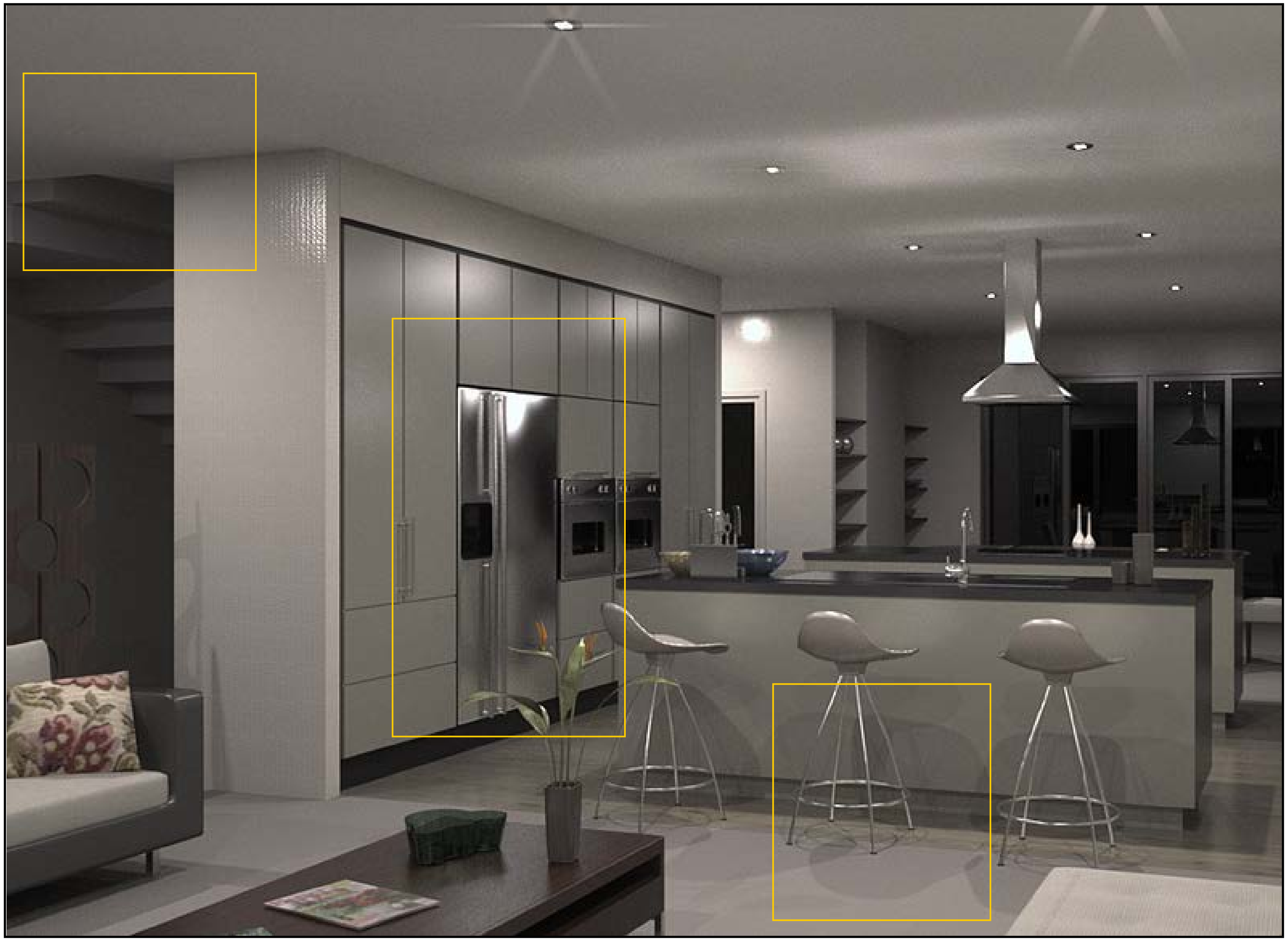


- What are the spectral characteristics of this field?

Frequency Analysis of Illumination (1)

- Global illumination effects have distinctively different spectral characteristics
- As a principle:
  - Diffuse inter-reflections produce low frequency directional radiance
  - The same holds for most cases involving occlusion in diffuse light bounces
  - Direct illumination with occlusion (shadows) contains high frequencies in general (discontinuities)
  - Specular transmission usually contains high frequencies

Frequency Analysis of Illumination (2)



Encoding the Radiance/Visibility Field (1)

- Why?
  - Direct illumination is cheap to calculate at every point on the geometry
  - Indirect illumination is not (see presentation about GI)
- Solution:
  - Precalculate on surfaces/cache points OR
  - Calculate at sparse locations at run time
- What:
  - Visibility AND/OR
  - Radiance field of indirect lighting

Encoding the Radiance/Visibility Field (2)

- Calculating and storing the radiance/visibility field once or per frame:
  - Disassociates its utilization from the geometry
  - Enables the easy evaluation of GI in real-time graphics (direct rendering techniques)

Encoding Visibility (Distant Illumination) (1)

- From the rendering equation:

$$L_r(\phi_r, \theta_r) = L_e(\phi_r, \theta_r) + \int_{\Omega_i} L_i(\phi_i, \theta_i) f_r(\phi_r, \theta_r, \phi_i, \theta_i) \cos(\theta_i) d\omega_i$$

- If we assume only a “distant” environment emitting the radiance (e.g. sky, sun, distant light sources etc), then:

$$L_r(\phi_r, \theta_r) = \int_{\Omega_i} \underbrace{\boxed{L(\phi_i, \theta_i)}}_{\text{radiance}} \underbrace{\boxed{V(\phi_i, \theta_i) f_r(\phi_r, \theta_r, \phi_i, \theta_i) \cos\theta_i}}_{\text{transfer function}} d\omega_i$$

Encoding Visibility (Distant Illumination) (2)

- For diffuse surfaces this is simplified to:

$$L_r(\phi_r,\theta_r)=\frac{\rho}{\pi}\int_{\Omega_i} L(\phi_i,\theta_i)\underbrace{V(\phi_i,\theta_i)\cos\theta_i}_{T(\phi_i,\theta_i)}d\omega_i$$

- The hemisphere is aligned with the surface normal at every point
- The transfer function characterizes the specific point but for diffuse inter-reflection can be considered a slow varying quantity (thus sparsely evaluated).

Encoding Visibility (Distant Illumination) (3)

- We can encode both the transfer function and the incident radiance using a set of basis functions
- Orthonormal bases (such as SH) are ideal as they provide the useful property:

$$\int \tilde{f}(s)\tilde{g}(s)ds = \sum_{i=1}^{k} f_k g_k$$

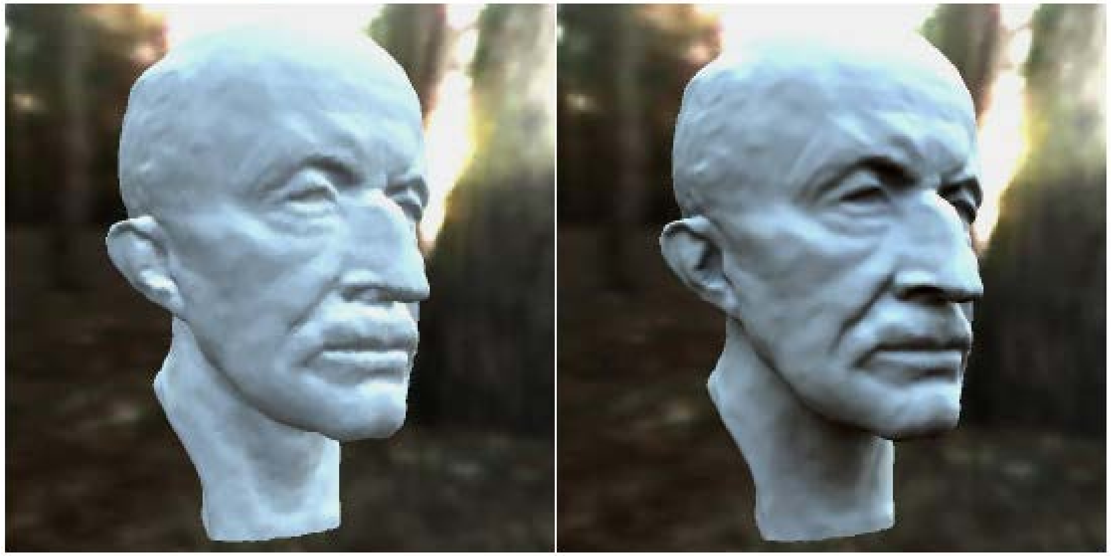
- i.e.: The integral of two band limited functions equals the dot product of their coefficients when projected to the orthonormal basis

Precomputed Radiance Transfer (1)

- The transfer (visibility over the hemisphere) function T can be precomputed and encoded in compact form
- When using Spherical Harmonics, 9 or 16 coefficients can effectively encode both *T* and L_i for diffuse light transfer
- The coefficients for T can be sparsely (pre-) evaluated, stored to and evaluated from:
  - A sparse lattice
  - A texture atlas

Precomputed Radiance Transfer (2)

$$L_r(\phi_r,\theta_r)=\frac{\rho}{\pi}\int_{\Omega_i}L(\phi_i,\theta_i)\cos\theta_i d\omega_i$$



$$L_r(\phi_r,\theta_r)=\frac{\rho}{\pi}\int_{\Omega_i}L(\phi_i,\theta_i)V(\phi_i,\theta_i)\cos\theta_i d\omega_i$$

Encoding the radiance field for diffuse GI (1)

- If $L(\mathbf{x},\omega)$ is the incident radiance field at point **p** from direction ω, then the diffusely reflected light at **p** is:

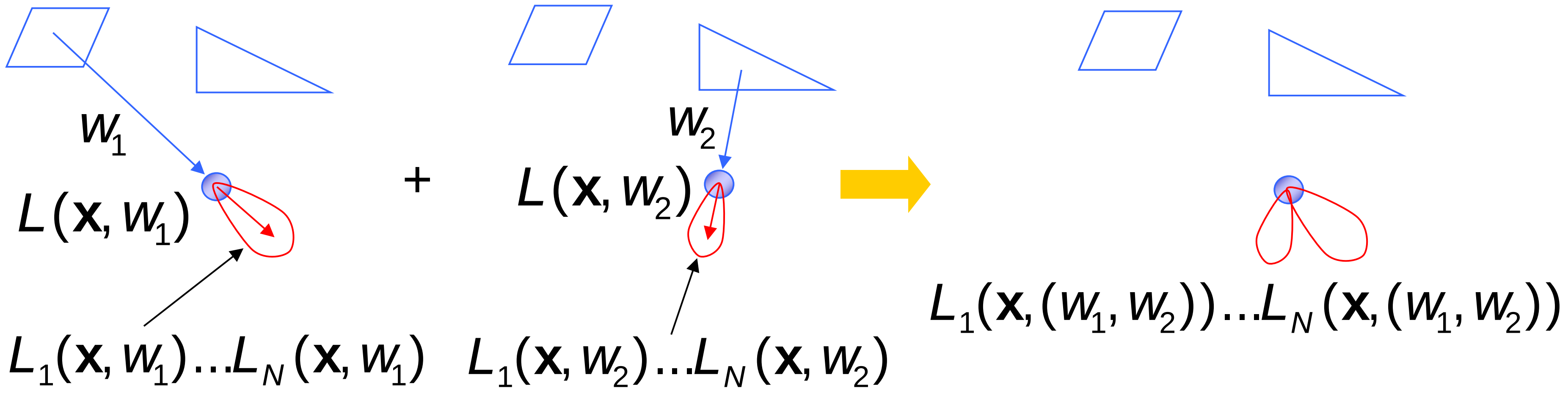
$$L_r(\mathbf{x}) = \frac{\rho}{\pi}\int_{\Omega_i} L(\mathbf{x},\omega_i)\cos\theta_i d\omega_i = \frac{\rho}{\pi}\int_{\Omega_i} L(\mathbf{x},\omega_i)H_i(\mathbf{n},\omega_i)d\omega_i$$

- Diffuse light is band limited, so using a projection to an orthonormal basis:
  - reflected radiance can be obtained from the N low order coefficients of the two functions:

$$L_r(\mathbf{x}) \;;\; \frac{r}{p}\mathop{\boldsymbol{\varepsilon}}\limits_{k=1}^{N} L_k(\mathbf{x})H_k(\mathbf{n})$$

Encoding the radiance field for diffuse GI (2)

- $L_k(\mathbf{x})$ are computed and interpolated at sparse locations (radiance field caching)
- $H_k(\mathbf{n})$ are computed at each evaluation point (closed form)
- $L_k(\mathbf{x})$ can be superimposed:

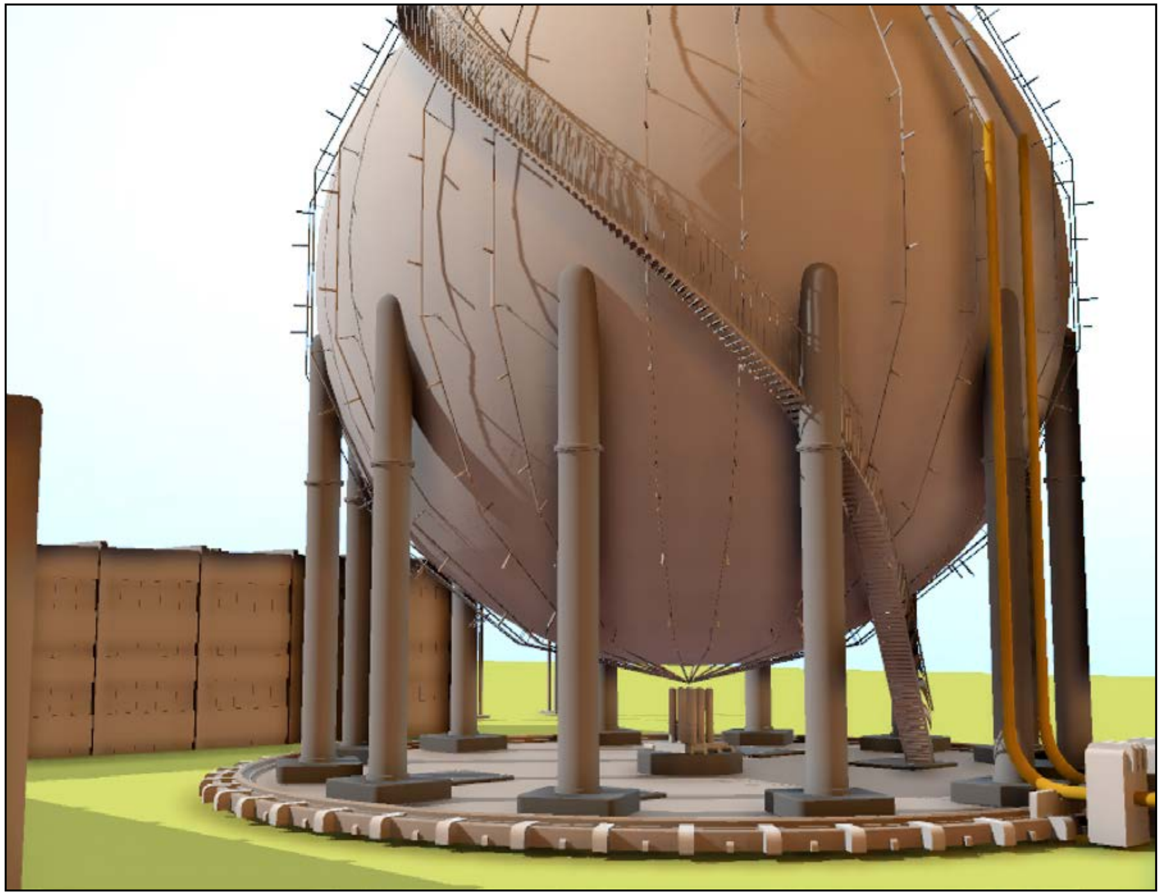

w_1
$L(\mathbf{x}, w_1)$
$L_1(\mathbf{x}, w_1) \ldots L_N(\mathbf{x}, w_1)$
+
w_2
$L(\mathbf{x}, w_2)$
$L_1(\mathbf{x}, w_2) \ldots L_N(\mathbf{x}, w_2)$
$L_1(\mathbf{x}, (w_1, w_2)) \ldots L_N(\mathbf{x}, (w_1, w_2))$

Part C: Real-time GI Methods

Techniques for completely dynamic scenes: no pre-computation

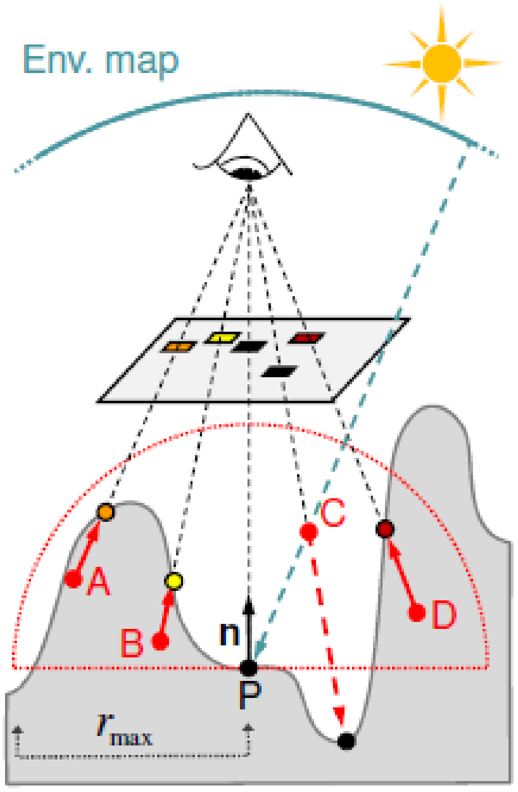
- Screen-space near field GI
- Instant radiosity
- Reflective shadow maps
- Radiance caching
- Volume-based global illumination
- Light propagation volumes
- Cascaded volume techniques

Screen Space Near Field GI (1)



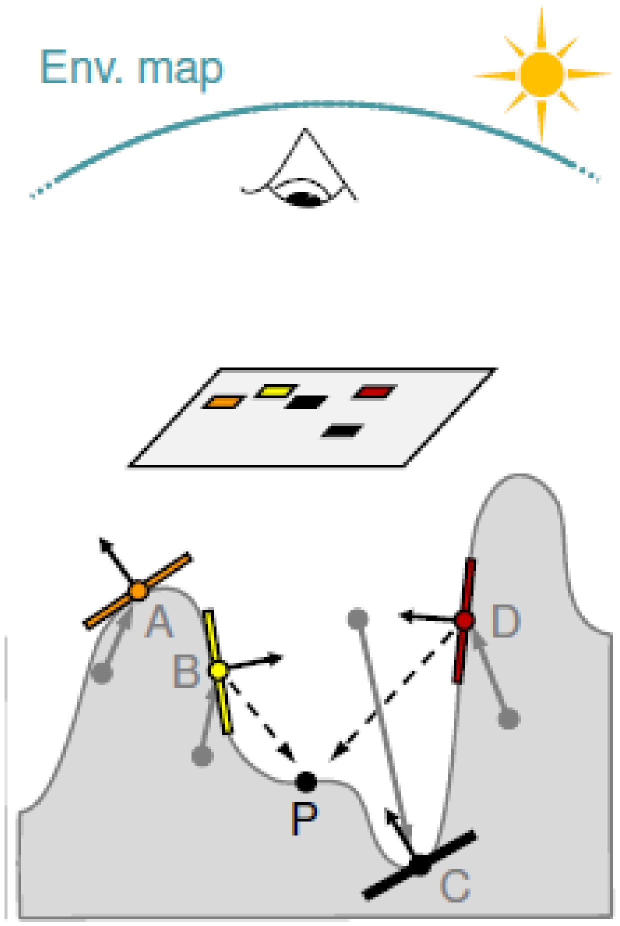
Screen Space Near Field GI (2)

- Distributes sample locations in hemisphere above a point **p** in screen space
- Check depth buffer for occlusion
- Directions to unoccluded points (C here) contribute to the direct lighting



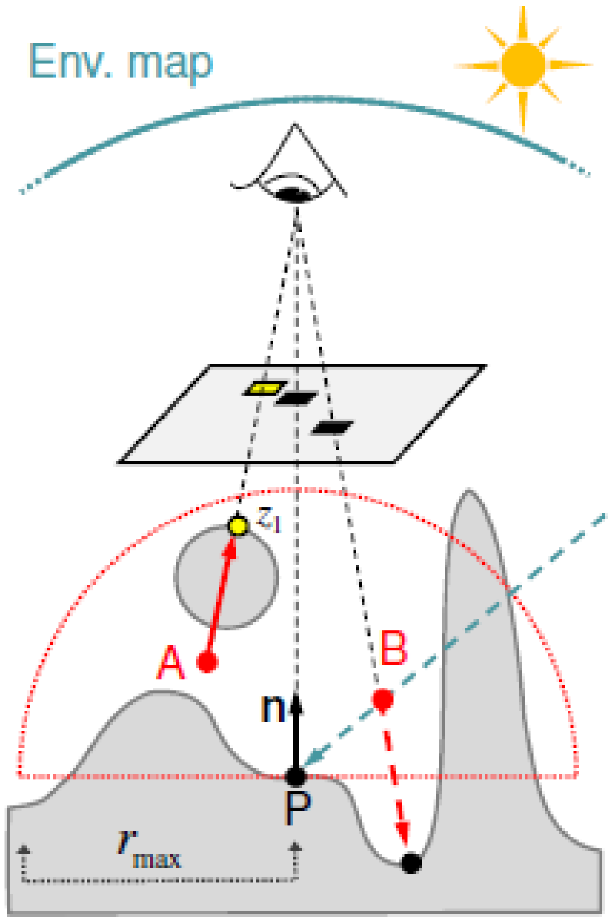
Screen Space Near Field GI (3)

- Occluded points are projected onto the depth map and their lighting and normal is measured
- Light is transferred to p according to the individual form factors calculated



Screen Space Near Field GI (4)

- Cons:
  - Very approximate solution
  - View dependency
  - Erroneous occlusion
- Pros:
  - Fast technique
  - Easy to implement

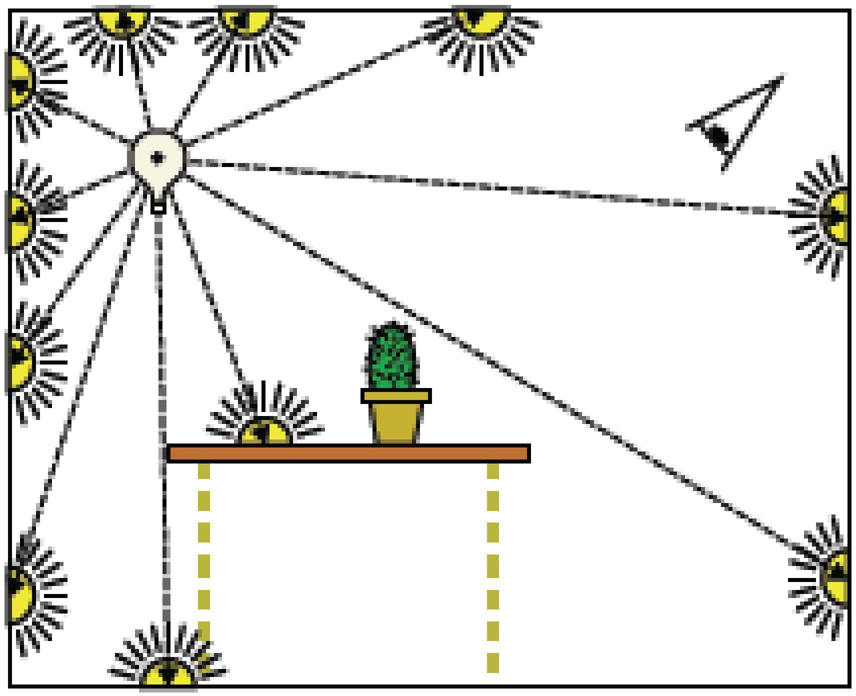


Instant Radiosity(1)

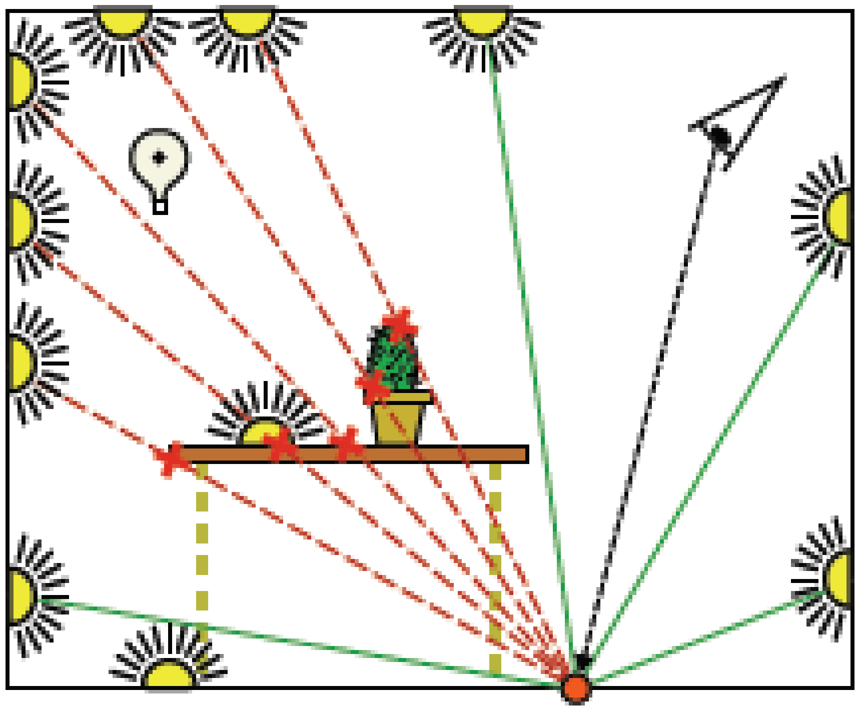
- Covers a wide range of methods, both interactive and off-line
- The concept is to replace indirect light bounces with direct illumination produced by virtual point lights (VPLs)
- VPLs (complete with visibility information) are placed at the intersection of photons from the light source with the geometry
- VPLs model the radiosity emitted from those intersection points
- VPLs are not limited to the first bounce only

Instant Radiosity(2)

VPL placement

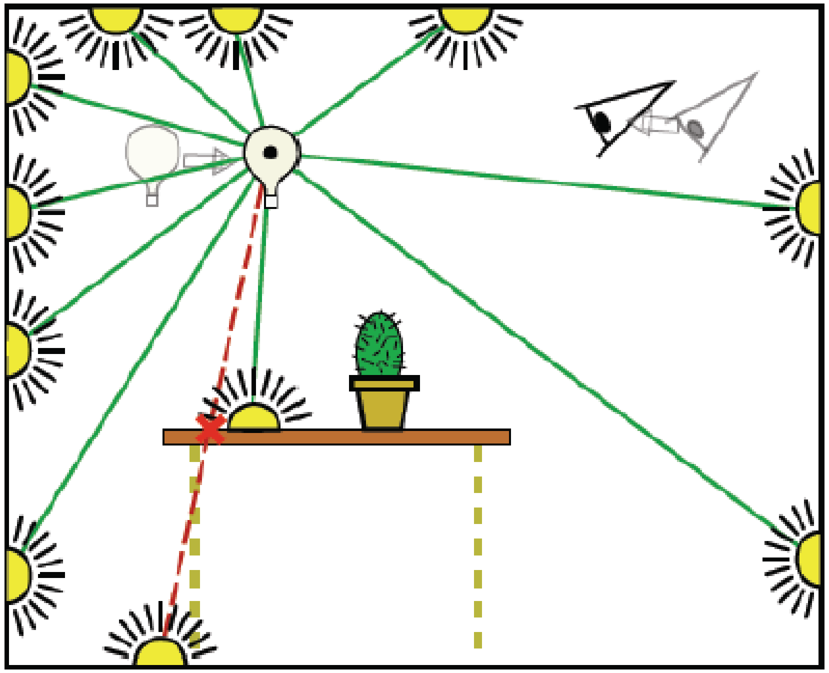


Indirect illumination from VPLs



Instant Radiosity – Dynamic VPL Update

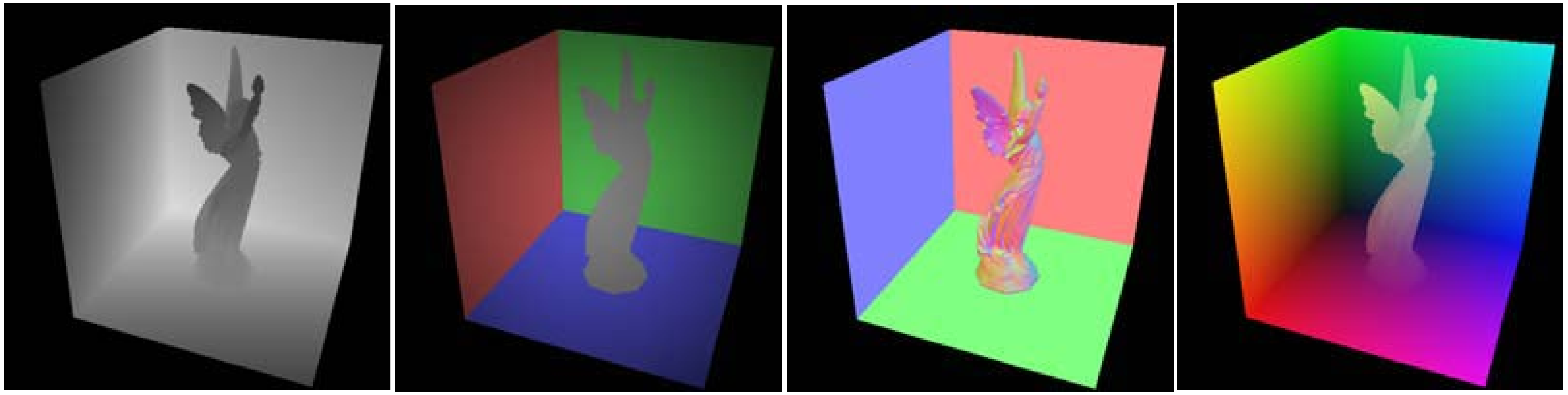
- Original CPU technique supported VPL updates
- When the scene changes, VPLs are updated:
  - Test VPL against shadow map
  - If invisible (beyond SM), discard VPL and add a new one



Reflective Shadow Maps(1)

Reflective Shadow Maps(1)

- Is a fast indirect lighting technique using:
- Shadow maps (depth maps) extended to also store VPL data:
  - Normals at visible points
  - Illumination (VPL power) at visible points
  - Optionally, location of VPLs and other data



Reflective Shadow Maps(2)

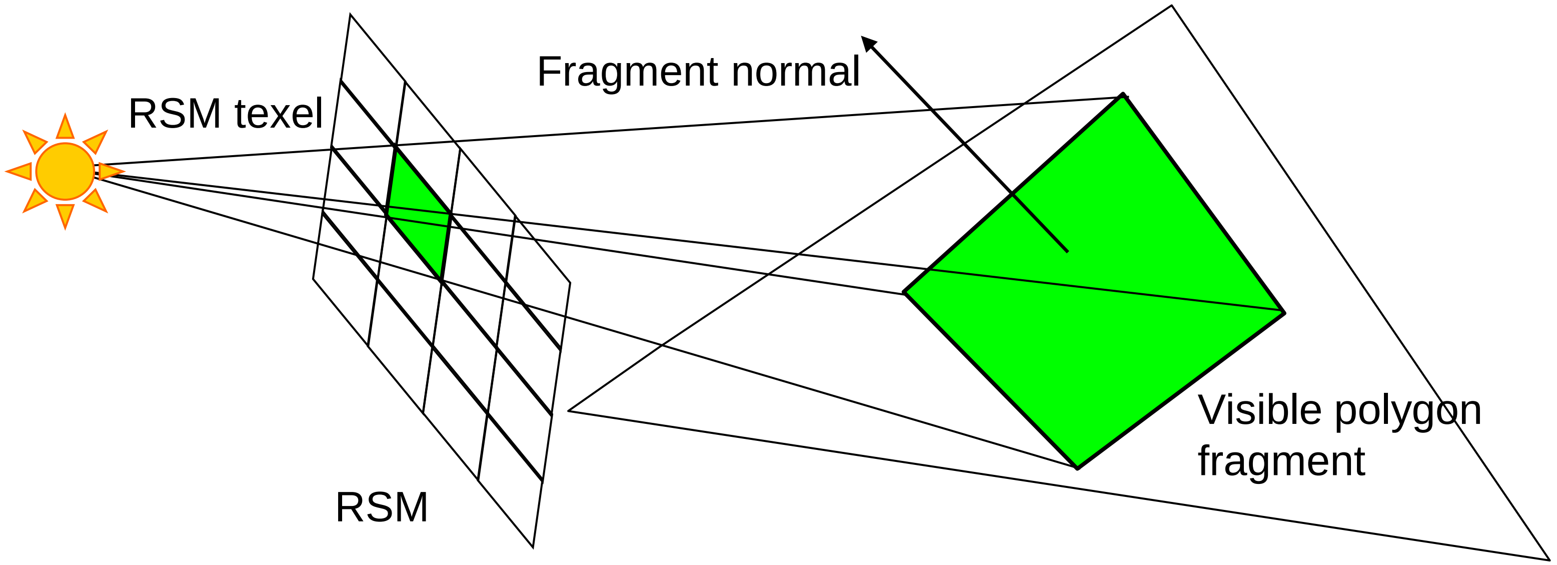
- Essentially, an RSM replaces the tracing of VPLs in the scene:
- Each SM texel is considered a VPL
- The shadow map contains the nearest scene points to the light source
- The extra data completely describe the power distribution of each VPL (shadow map texel)
- The extended SM storage is used by other GI techniques → RSM now also refers to the multi-channel shadow map storage.

Reflective Shadow Maps (2)

- What the RSM does NOT provide is visibility information for each VPL
- Therefore, the light from each VPL is considered unoccluded → no secondary bounce occlusion
- Also, RSM provides first-bounce GI only

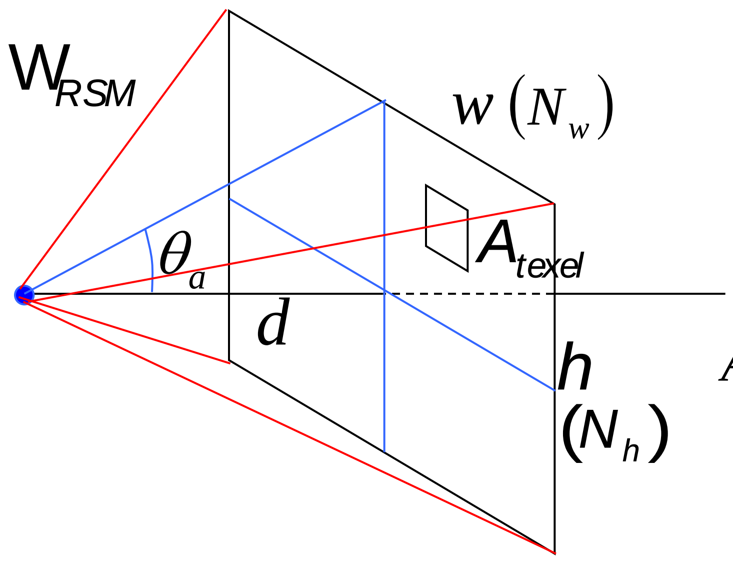
RSM – VPL Lighting Calculations (1)

- In the bibliography, the RSM illumination channel stores anything, from radiosity, intensity, to power
- Each texel (VPL) can be considered a cosine weighted point light but a more accurate modeling is a small (trapezoid) area light:



RSM – VPL Lighting Calculations (2)

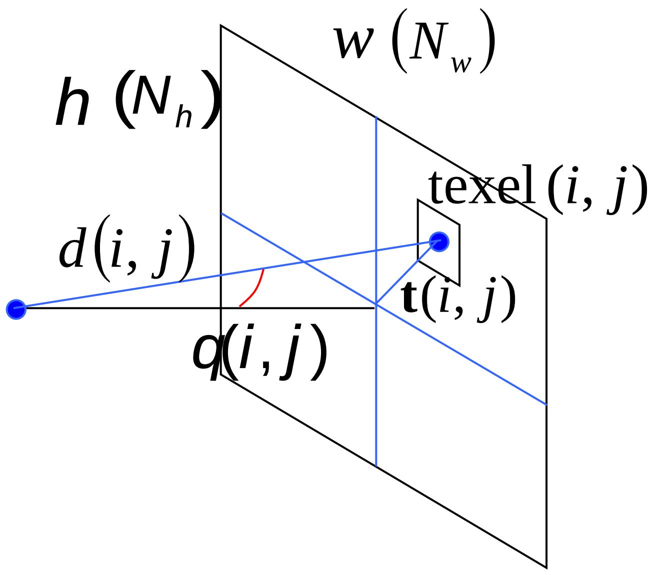
- Assume a directional light source with total flux Φ_{tot}, a shadow map with $N_w \times N_h$ square texels, distance d from the projection plane and vertical half aperture θ_a :



$$\Omega_{RSM} = 4\arcsin\left(\sin\left(\frac{w}{2}\right)\sin\left(\frac{h}{2}\right)\right)$$

$$A_{texel} = \frac{wh}{N_w N_h} = \frac{\left(\frac{N_w}{N_h} d \tan\theta_a\right) d \tan\theta_a}{N_w N_h} = \frac{d^2 \tan^2\theta_a}{N_h{}^2}$$

RSM – VPL Lighting Calculations (3)

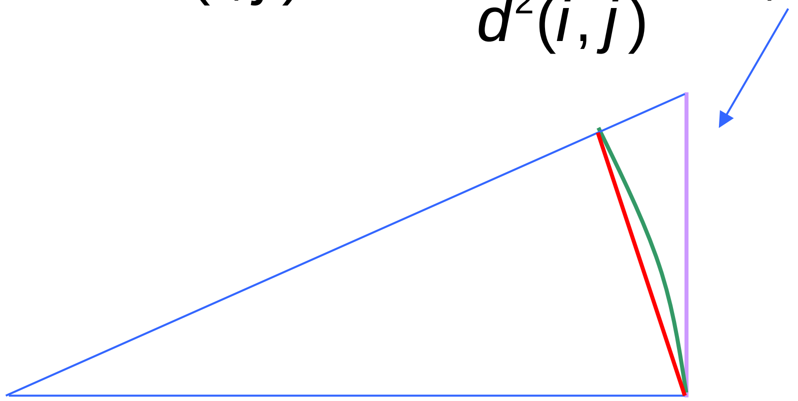


$$\mathbf{t}(i, j) = \left(-\frac{w}{2} + \left(\frac{1}{2} + i\right)\frac{w}{N_w}, \frac{h}{2} - \left(\frac{1}{2} + j\right)\frac{h}{N_h} \right)$$

$$\theta(i, j) = \arctan\left(\frac{|\mathbf{t}(i, j)|}{d}\right)$$

$$d(i, j) = d / \cos\theta(i, j)$$

$$w(i, j) = \frac{\boxed{A_{texel,PROJECTED}}}{d^2(i, j)} \; ; \; \boxed{\frac{\boxed{A_{texel}} \cos\theta(i, j)}{}}\!\!\frac{}{d^2(i, j)} = \frac{\tan^2\theta_a \cos^3\theta(i, j)}{{N_h}^2}$$



RSM – VPL Lighting Calculations (4)

- The power transmitted through RSM texel (i,j) that corresponds to the power of the (i,j) virtual area light is:

$$\Phi(i,j) = r(i,j)\frac{w(i,j)}{W_{RSM}}\Phi_{tot}$$

- Using the recorded RSM depth and normal at (i,j), we can also estimate the radiosity at any point on the virtual light:

$$A_{VL,PROJECTED} = \frac{depth^2(i,j)}{d^2(i,j)}A_{texel} \Rightarrow \boxed{A_{VL} = \frac{depth^2(i,j)A_{texel}}{d^2(i,j)<\mathbf{l}(i,j),\mathbf{n}(i,j)>_+} = \frac{depth^2(i,j)A_{texel}}{d^2(i,j)\mathbf{n}_z(i,j)}}$$

$$B(i,j) \approx \frac{\Phi(i,j)}{A_{VL}}$$

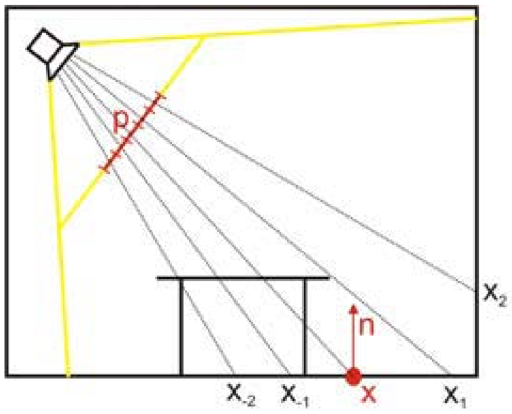
$\rightarrow$ Average radiosity over area light

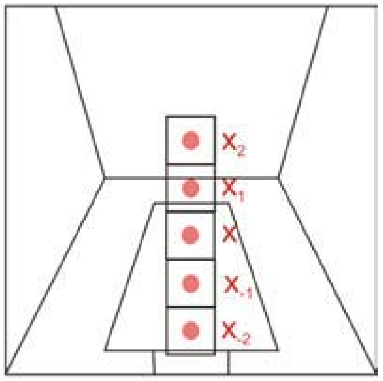
Using the RSM for Global Illumination (1)

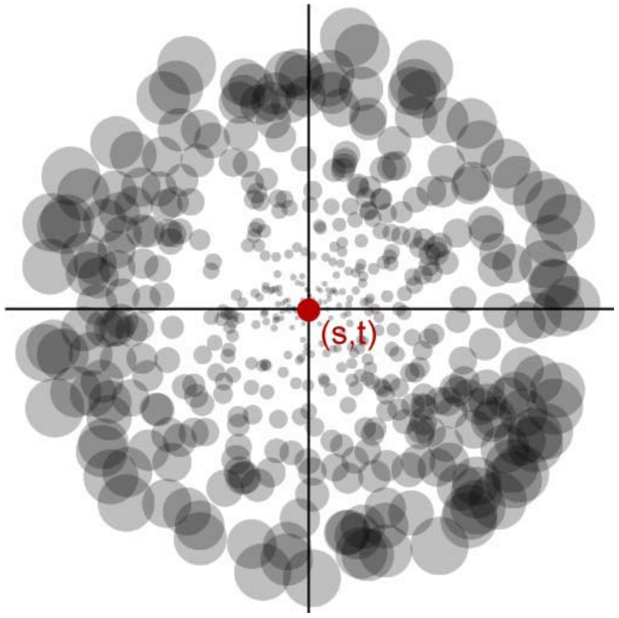
- RSM texels are sampled in the same manner as VPLs
- Light transfer can be estimated between each RSM virtual area light (or point light, depending on model) and the illuminated point
- Caution: Light transfer does not evaluate visibility between RSM samples and the receiving point

Using the RSM for Global Illumination (2)

- Practical RSM sampling:
  - Project receiving point on RSM
  - Determine an area around projected point in RSM parametric space to sample
  - Accumulate RSM sample contribution





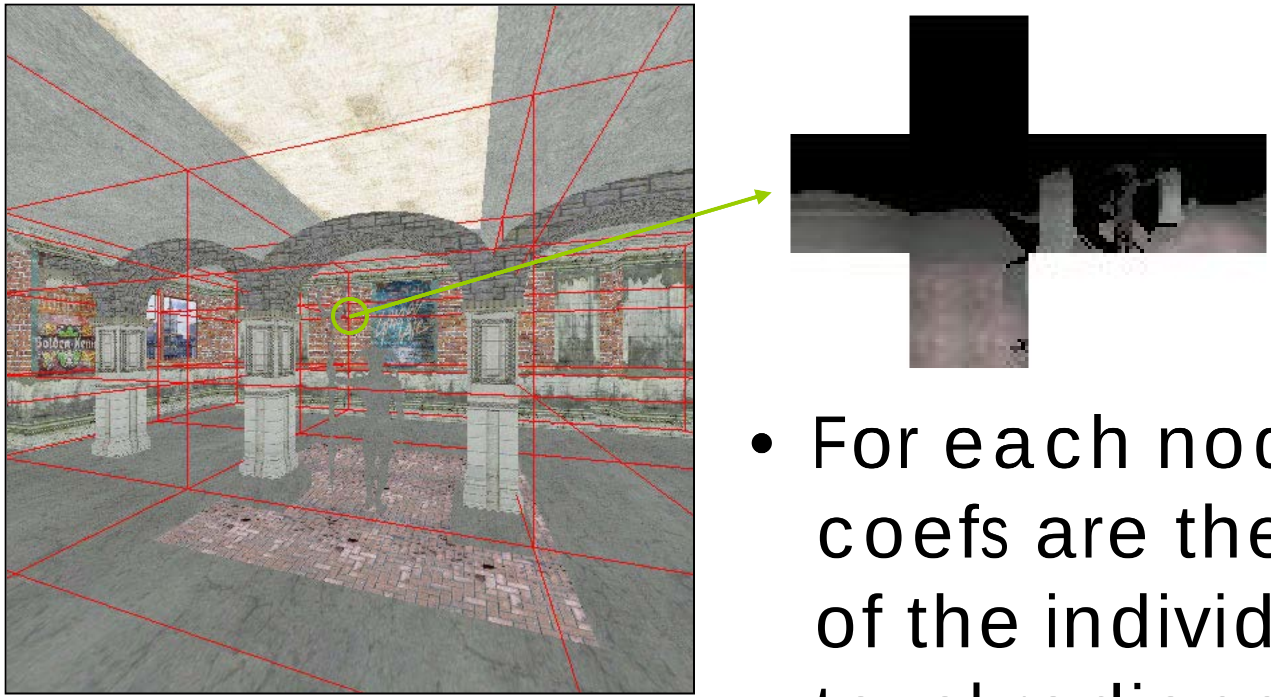


Radiance Field Caching (1)

Radiance Field Caching (2)

- Estimates the incident radiance field at the vertices of a uniform grid
- Radiance is captured by rendering the scene on a cubical environment map
- Compresses the radiance field using SH
- Evaluates the reflected radiance on surfaces by direct integration of the radiance field with the BRDF at each point in SH space
- SHs for points in between lattice vertices are interpolated.

Radiance Field Caching (3)



- For each node, the SH coefs are the superposition of the individual cubemap texel radiance projection:

$$L_l^m \approx \sum_{face=1}^{6} \sum_{i=1}^{size} \sum_{j=l}^{size} L_{face}(i,j) Y_l^m(\omega) A(\omega)$$

$$A(\omega) = \int_{pixel_{ij}} d\omega$$

Radiance Field Caching (4)

- Reflected radiance can be directly evaluated from the radiance field SH coefficients and the SH coefs of the transfer function (oriented BRDF):

$$
\begin{aligned}
L_{indirect}(\omega_o) &= \mathbf{T}L \\
&= \int_{\omega_i \in \Omega_N} L(\omega)\,\rho(\omega_i,\omega_o)\ \cos\theta_i\, d\omega_i \\
&= \int_{\omega_i \in \Omega_N} L(\omega)\,\widehat{\rho}(\omega_i,\omega_o)\ d\omega_i \\
&= \sum_l \sum_{m=-l}^{l} L_l^m \int_{\omega_i \in \Omega_N} Y_l^m(\omega)\ \widehat{\rho}(\omega_i,\omega_o)\ d\omega_i \\
&= \sum_l \sum_{m=-l}^{l} L_l^m\, T_l^m(\omega_o)
\end{aligned}
$$

Radiance Field Caching (5)

- For Lambertian surfaces (diffuse reflection):

$$L_{indirect}(\mathbf{p}) = \frac{\rho(\mathbf{p})}{\pi} \underbrace{\sum_{l} \sum_{m=-l}^{l} L_l^m(\mathbf{p})}_{\text{Radiance field SH coefs interpolated from 8 nearest lattice points}} \underbrace{H_l^m(\mathbf{n})}_{\text{Normal-aligned projected cosine-weighted hemisphere on SH basis}}$$

- Diffuse GI is well approximated with 2-3 order SH
- The transfer function can be generalized to Phong-like models (symmetric lobes) but require a significantly larger SH order (6+)→ impractical storage

Radiance Field Caching (6)

- Practical issues:
  - For truly dynamic scenes, cubemaps must be completely re-evaluated often
  - Secondary bounces may be handled by exchanging light among lattice points
  - The sparseness of the grid necessitates additional occlusion criteria when evaluating the radiance field:
    - Depth maps are also acquired per node
    - Instead of simply trilinearly interpolating the node radiance, a visibility check is performed against the node’s range in the direction of the sample

Volume-based Global Illumination

Volume-based GI (1)

- Uses an intermediate regular approximation of the geometry (voxel grid) to store lighting and geometry data →
- Rough discretization of the shaded environment
- Why volume-based GI?
  - Decouples local pixel calculations (GPU pipeline) from full-scene data
  - Provides access to full-scene data in the local-only context of a shaded pixel
  - GI calculations independent of scene complexity

Volume-based GI (2)

- The “lit” voxels represent virtual point lights
- Occupied voxels effectively block light transport
- What do we need to store for one-bounce GI (per voxel):
  - Direct lighting (VPLs) directionally encoded using the normal at the shaded fragments
  - Voxel coverage as occupancy (same storage – black voxels)
- What do we need for extra bounces?
  - Averaged (per voxel) surface normals
  - Average (per voxel) albedo

Volume-based GI (3)

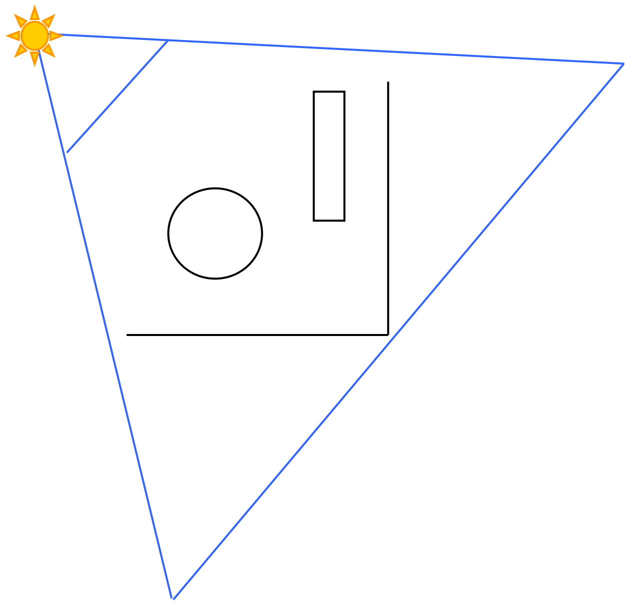
- All methods have two phases:
  - Volume data generation
  - GI estimation
- Volume generation:
  - Point injection
    - Geometry-based
    - Image-based
  - Multi-channel full-scene voxelization
- GI estimation:
  - Iterative radiance diffusion (light propagation volumes)
  - Ray marching

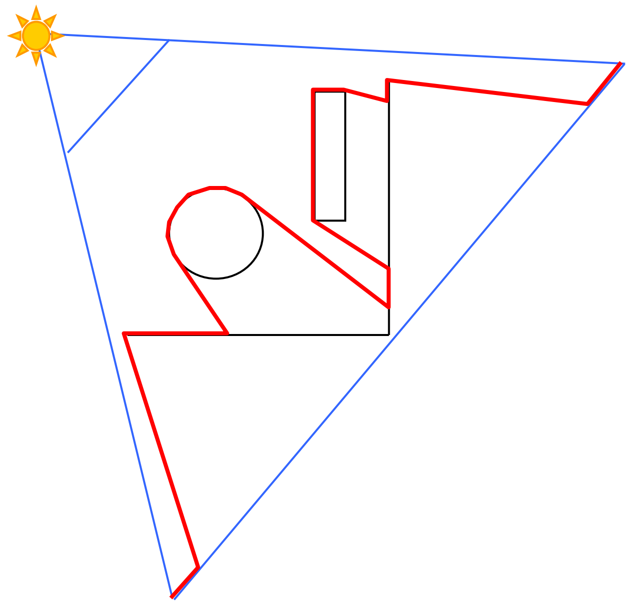
VBGI – Image-based Point Injection (1)

- Samples from the available frame buffers are injected into the volume using the technique discussed in part A
- Shadow maps (RSMs) hold a sampling of the surfaces lit by the particular light source → VPLs
- The camera buffer (MRT G-buffer) contributes additional occupancy-only points

VBGI – Image-based Point Injection (2)

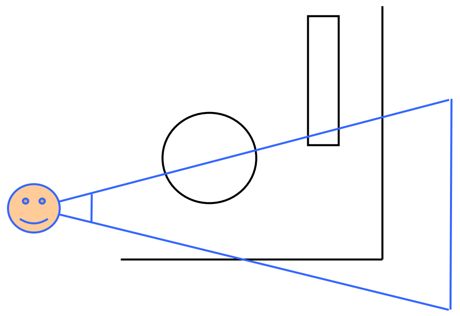
- How are the points injected?
  - Reflective shadow map acquisition:


Light setup

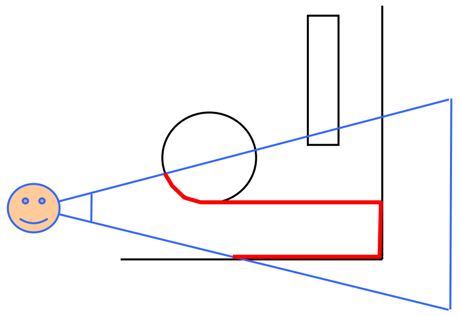

Shadow map points (WCS)

VBGI – Image-based Point Injection (3)

- How are the points injected (cont)?
  - Camera g-buffer acquisition (deferred rendering):



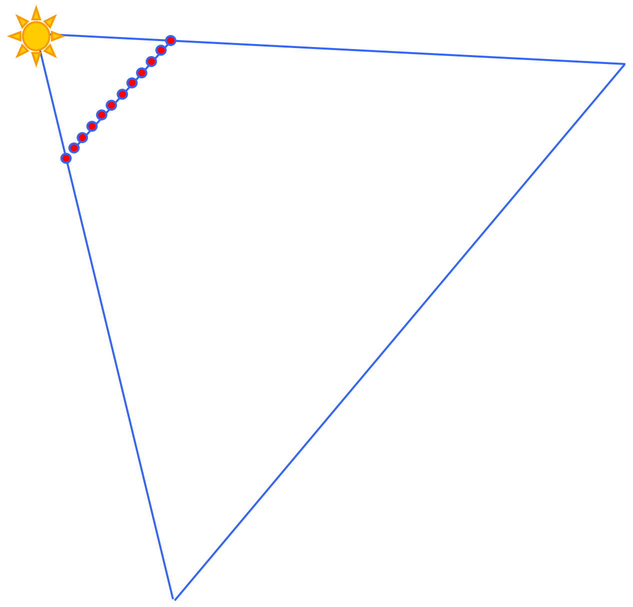
Camera setup

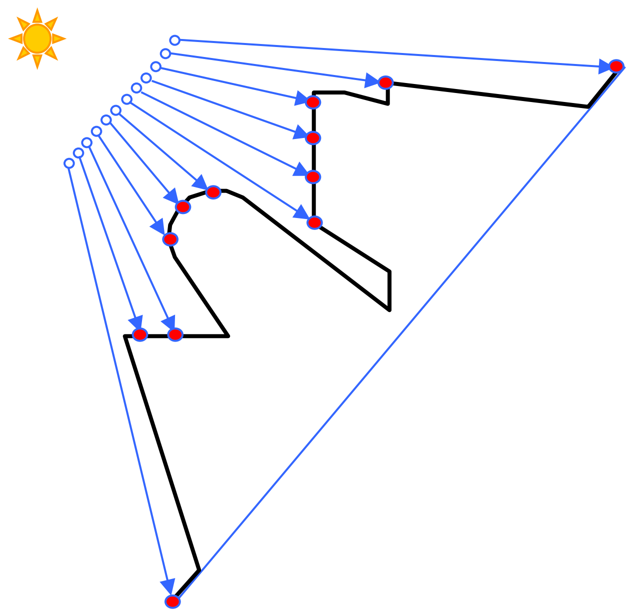


camera depth points (WCS)

VBGI – Image-based Point Injection (4)

- How are the points injected (cont)?
  - Geometry (points) generation:





- Render a planar grid of points.

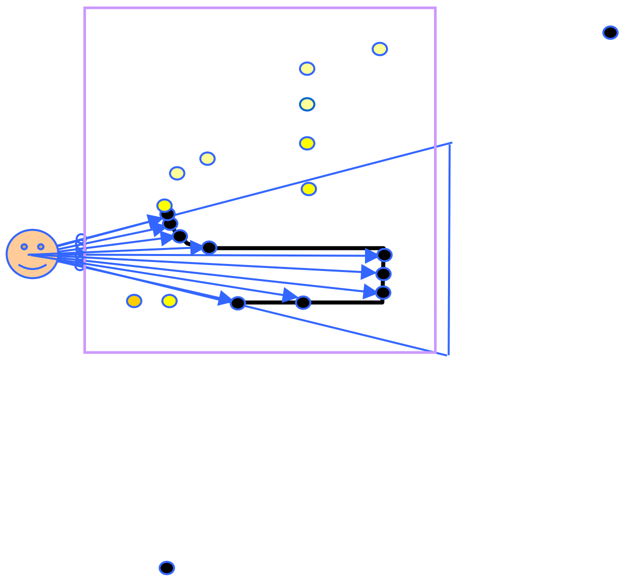
  For simplicity, arrange points in ([0,1],[0,1],0) interval

In a geometry shader:

- Lookup the (x,y) depth from the SM
- Transform (x,y,depth) to vol. coords
- Inject the transformed point in volume

VBGI – Image-based Point Injection (5)

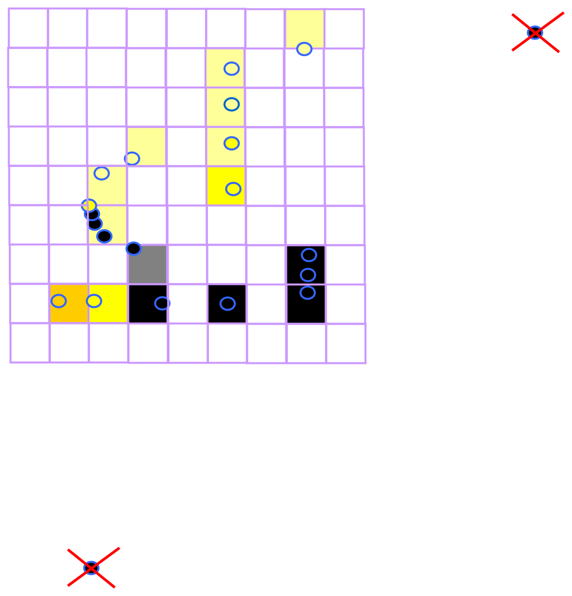
- How are the points injected (cont)?
  - Do the same for the camera buffer points:



- Additional camera points are unlit points
- We repeat the process for all available buffers (lights, reflection buffers, env. maps etc)

VBGI – Image-based Point Injection (6)

- The corresponding voxels now store the encoded lighting, occupancy and other data:

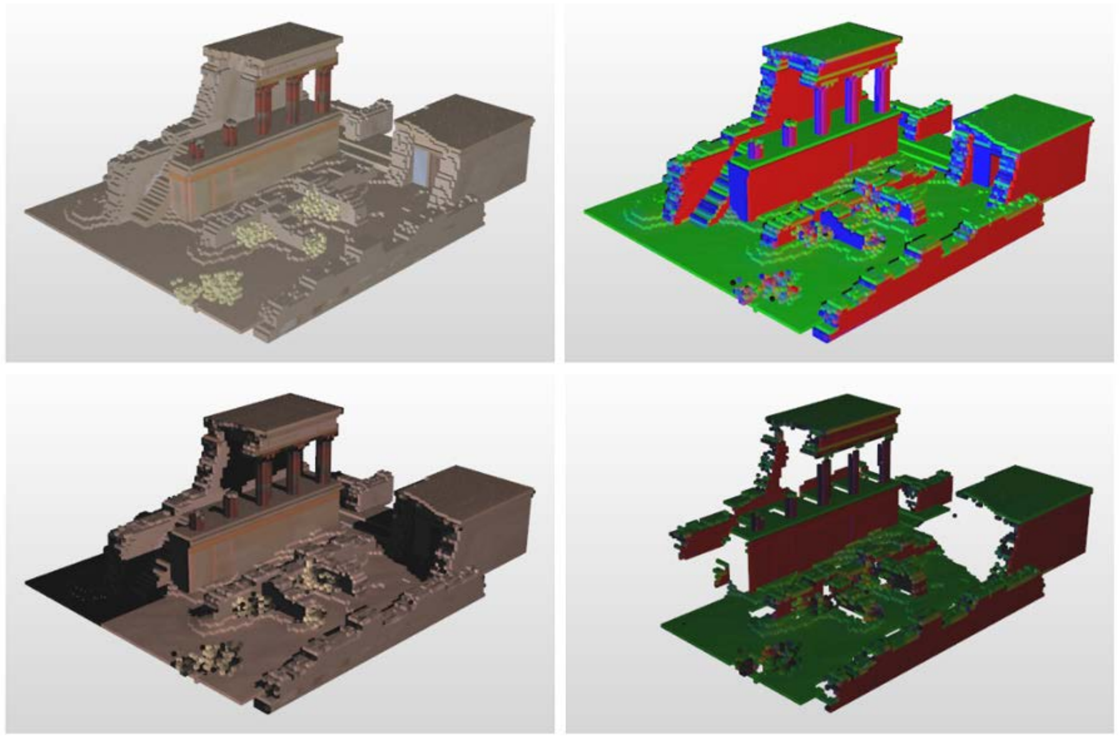


- The injected point contribution is not the same for all points! More on this later

VBGI – Full Scene Voxelization (1)

- Rasterizes the geometry into the volume buffer directly from the geometric data
- Imprints a complete occlusion information, regardless of visibility to buffers
- Voxelization → 3D Rasterization:
  - Voxel shaders compute and encode direct lighting, normals, albedo and occupancy
  - 2-5 volume textures required
- Many ways to perform it
- All methods slice the geometry into volume layers

VBGI – Full Scene Voxelization (2)

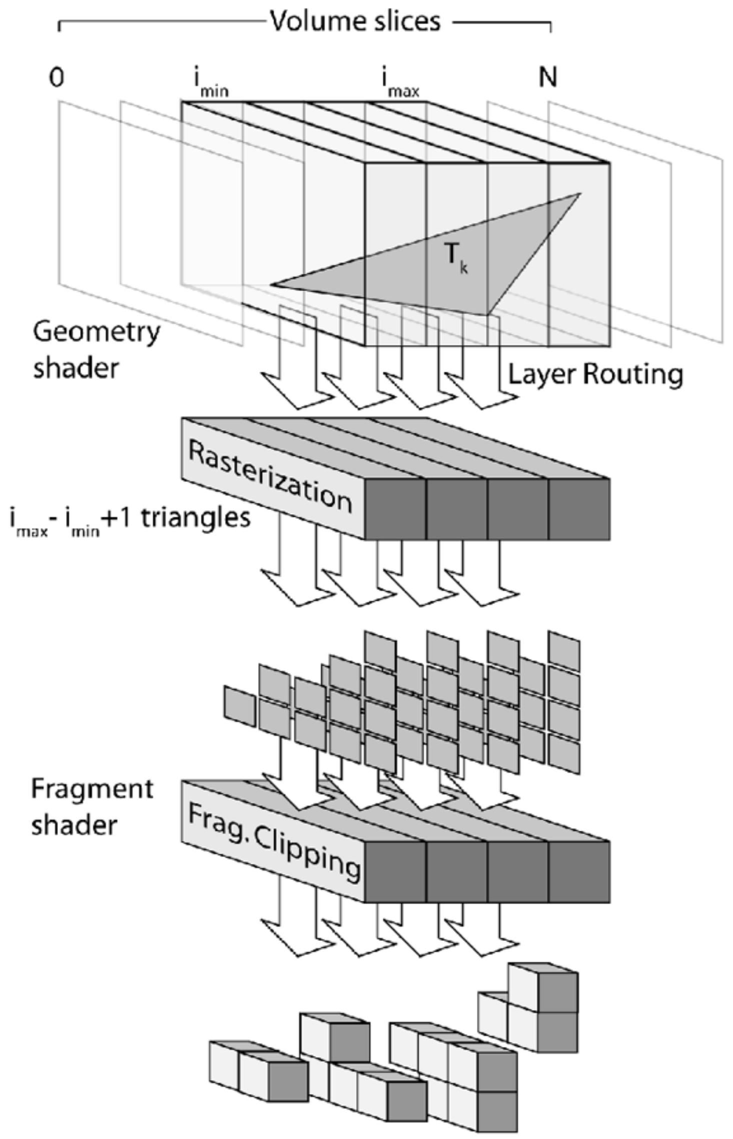


	texture channels				
Render targets (volumes)	R	G	B	A	
0	n_x	n_y	n_z	o	Luminance only
1	s_{00}	s_{1-1}	s_{10}	s_{11}	

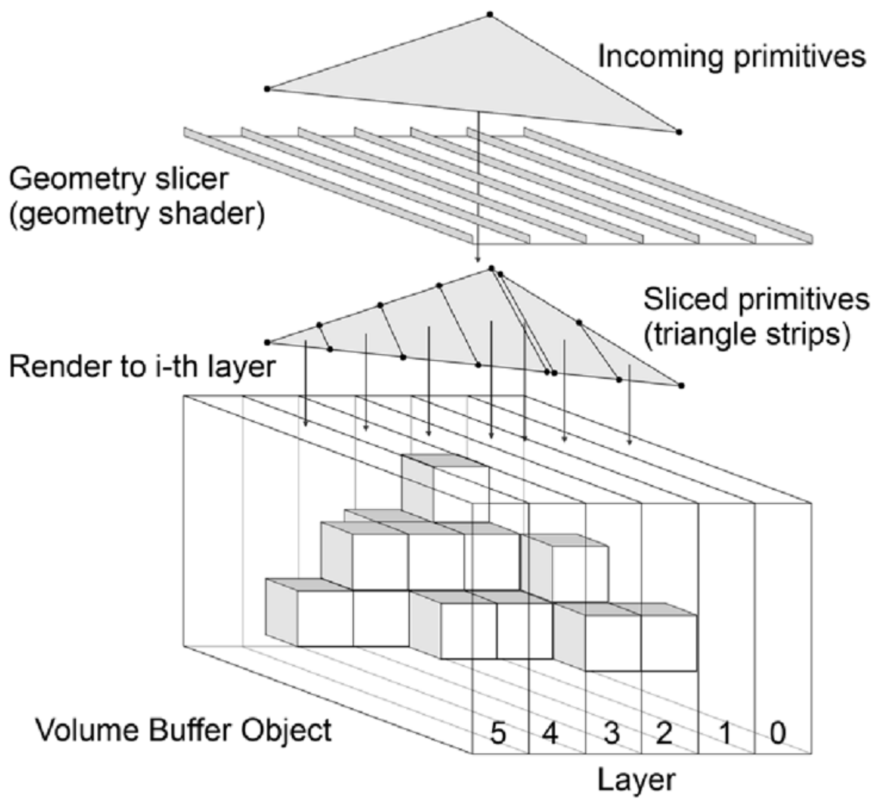
	texture channels				
Render targets (volumes)	R	G	B	A	
0	n_x	n_y	n_z	o	Full color GI
1	sr_{00}	sr_{1-1}	sr_{10}	sr_{11}	
2	sg_{00}	sg_{1-1}	sg_{10}	sg_{11}	
3	sb_{00}	sb_{1-1}	sb_{10}	sb_{11}	
4	c_r	c_g	c_b	c_a	+ color bleeding

VBGI – Full Scene Voxelization (3)

Fragment shader clipping

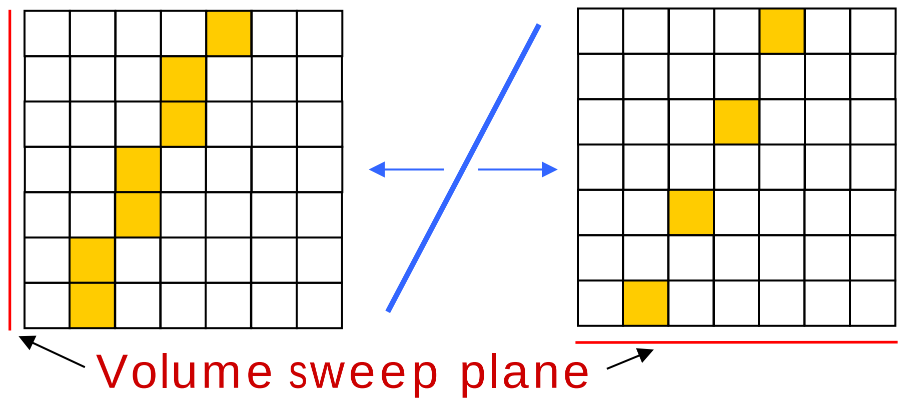


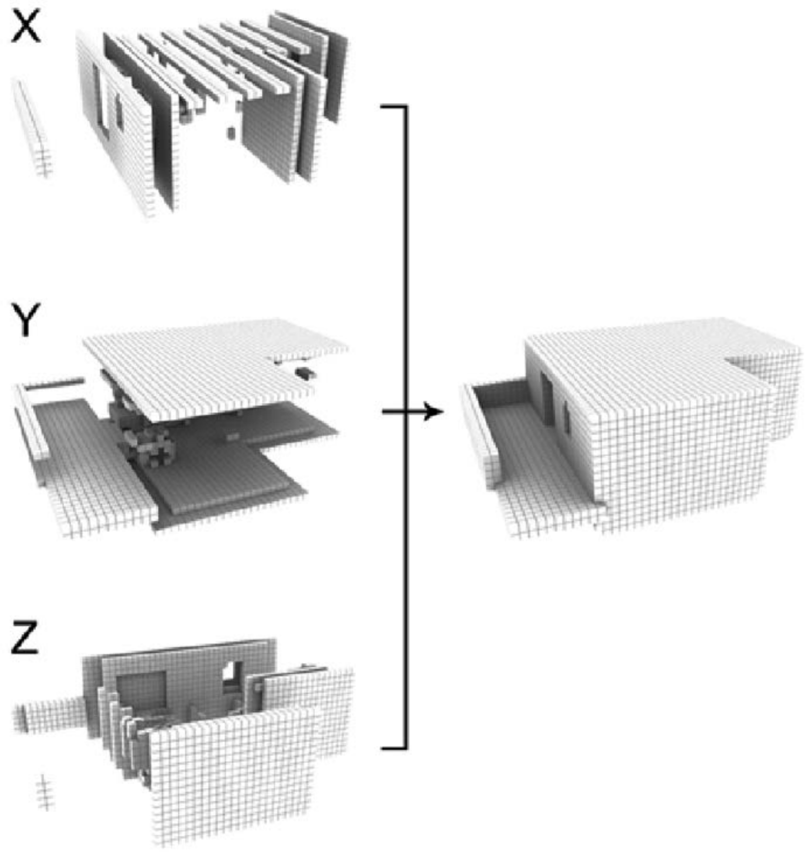
Geometry shader clipping



VBGI – Full Scene Voxelization (4)

- Polygons are rasterized to the volume sweep of maximum projection
- This ensures dense, coherent sampling





Binary data: OR op. Scalar data: MAX op.

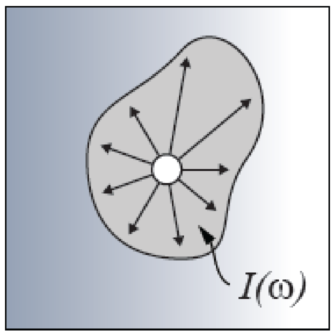
Blocking – Geometry Orientation/Coverage

- As volume textures are quite crude (e.g. 32^3), voxels should not be either on or off
- Regardless of volume generation method, volumes should store:
  - Occupancy proportional to voxel coverage and alpha → This is easier in full voxelization
  - Directional data (SHs) for each injected fragment →
    - Multiple surfaces with different orientations cross the voxel

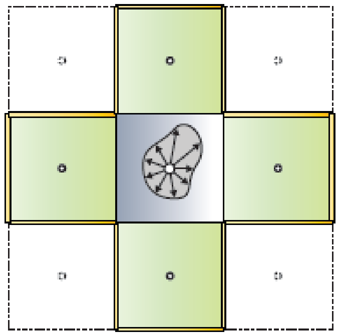
Light Propagation Volumes

Light Propagation Volumes (1)

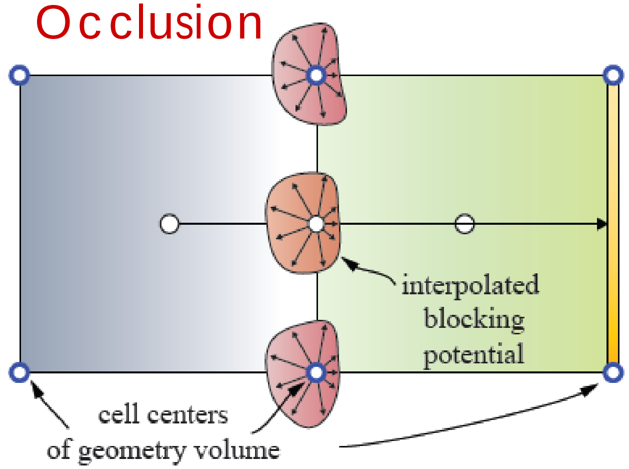
- Iteratively propagates flux from each cell to the next
- Blocks (attenuates) light according to occupancy data

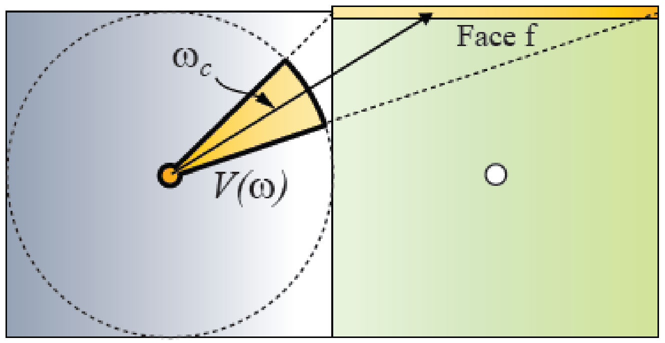


source cell

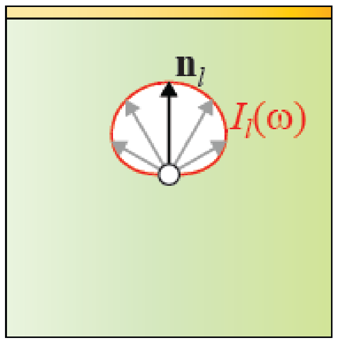


propagation along axial directions


Occlusion
interpolated blocking potential
cell centers of geometry volume



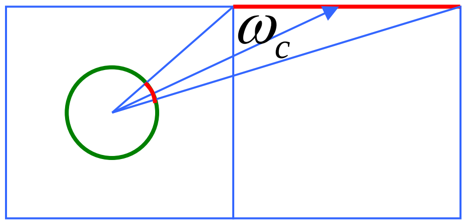
source cell destination cell



reprojection of the flux into a point light

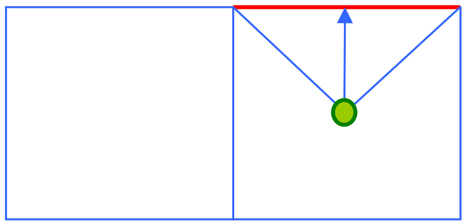
Light Propagation Volumes (2)

- The flux incident to each one of the faces of the neighboring cell is difficult to approximate as an integral using low-order SHs
- A rough empirical approximation is suggested:
  - Estimate the intensity in direction ω_c to the cone V(ω) center
  - Scale by the ratio of the solid angle subtended by the face against 4π (spherical solid angle)



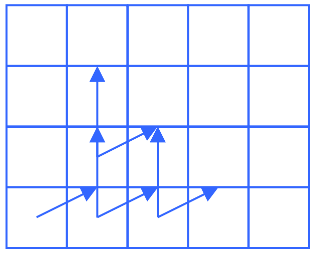
Light Propagation Volumes (3)

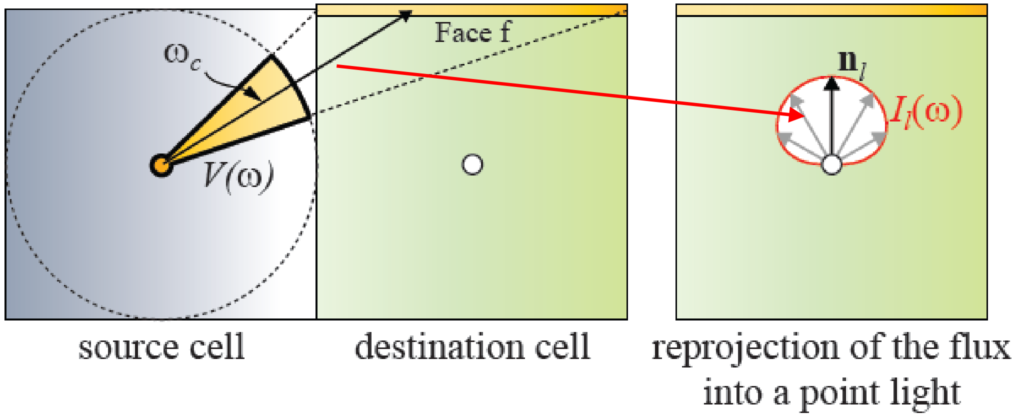
- Then a new VPL is generated at the neighboring cell with intensity matching the total flux of the face
- The VPL is encoded as SH and added to the cells intensity distribution



Light Propagation Volumes (4)

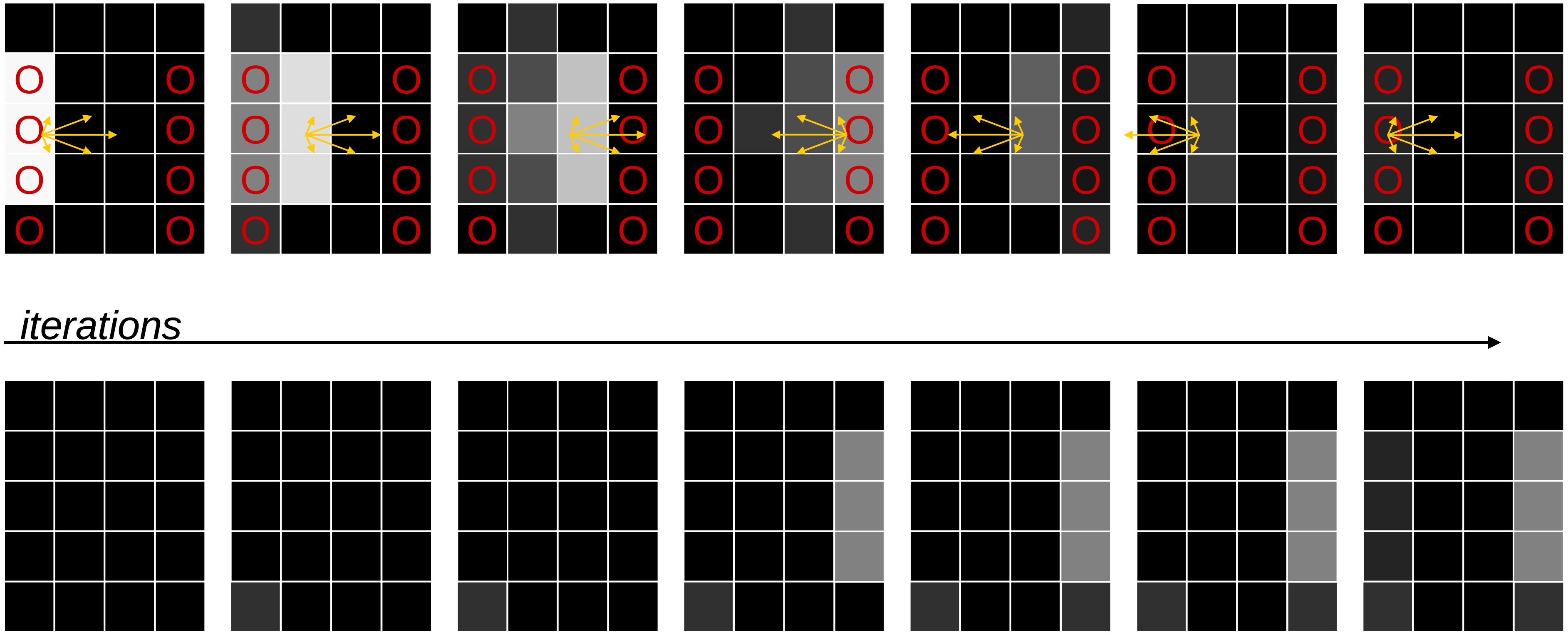
- Not a physically correct solution:
- Although flux balance is maintained,
- Flux is assumed to get diffused on "translucent walls" due to the change in propagation direction





Light Propagation Volumes - Bounces

Spherical harmonic buffer (pair – swapped for reading/writing)



GI accumulation buffer (flux sampled from decoded SH)

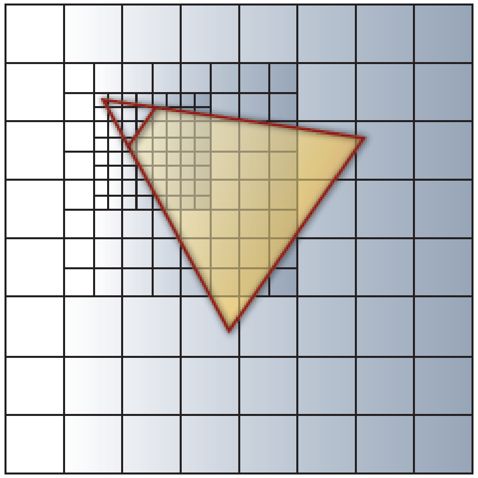
- Some leaking still occurs due to low SH order (series truncation) and approximate blocking

Light Propagation Volumes - Requirements

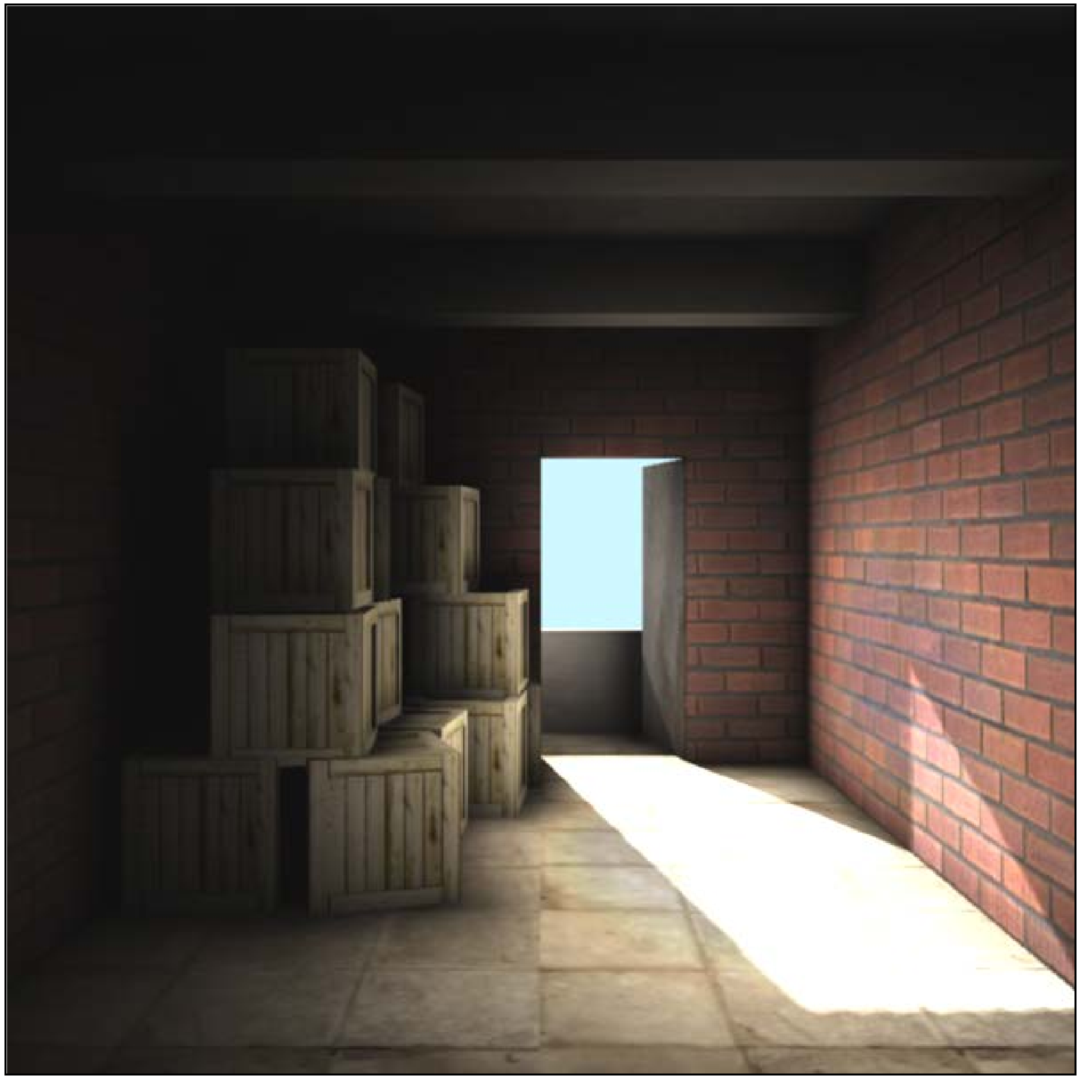
- Geometry (occlusion) volumes: 2nd order SH ($l=1$: 4 coefs) to encode directionality
- RGB Flux volumes: 3 X 2nd order SH: 12 coefs
- Need to duplicate flux volumes for ping pong rendering (iterations)

Cascaded LPVs

- Why?
  - Scenes are large to be covered by a single low-res volume (large volumes are slow and costly)
  - We need many iterations to transport flux across the scene
- Solution: Cascades
  - Overlapped volumes of same resolution but different size
  - Denser sampling near camera



VBGI - Ray Marching



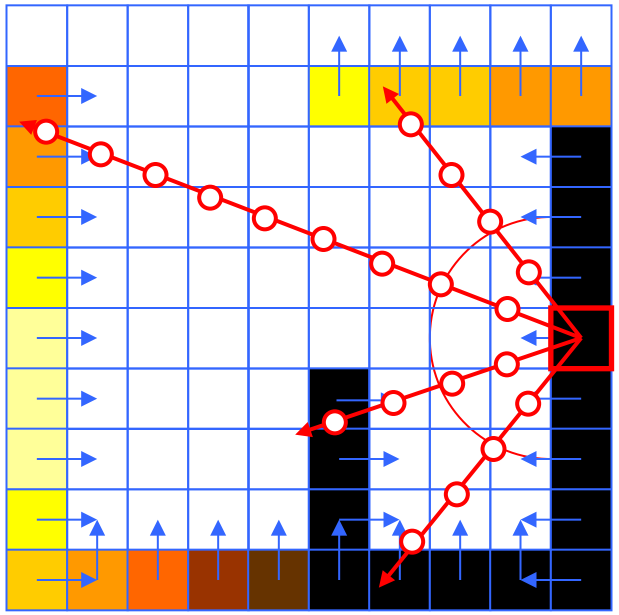
VBGI - Ray Marching (1)

- We can approximate a gathering operation (Monte Carlo integration) by marching rays in the volume instead of intersecting them with the scene
- We can march rays either from the shaded fragments or from the GI volume voxels (faster but cruder)

VBGI - Ray Marching (2)

- Ray marching:
  - Iteratively sample the volume along a line until a fully blocked voxel is reached
  - Gather light along the line from occupied voxels, according to orientation stored in them
  - Perform integration with the BRDF at the shaded point → Simple SH dot product for diffuse reflection

VBGI - Ray Marching (3)



```
Generate N random rays
L_gi = 0;
for each ray dir:
    s = ds;
    while s < r_max
        v = p + s*dir;
        if Occ(v)>0.5
            break;
        s += ds;
    if s >= r_max
        continue;
    F = clamp(dot(-Normal(v),dir),0,1);
    F *= clamp(dot(Normal(p),dir),0,1);
    L_gi += F*L(v);
L(p) += Color(p)*L_gi/N;
```

VBGI - Comparison

- Light propagation volumes:
  - Is fast
  - Not physically correct
  - Cannot guarantee that light reaches opposite surface
  - View dependent →
    - incomplete occlusion
    - Temporal aliasing (popping artifacts)
- Full voxelization GI:
  - More accurate
  - Stable
  - Slower