Hardware Implementation

SIGGRAPH2008

Pascal Gautron

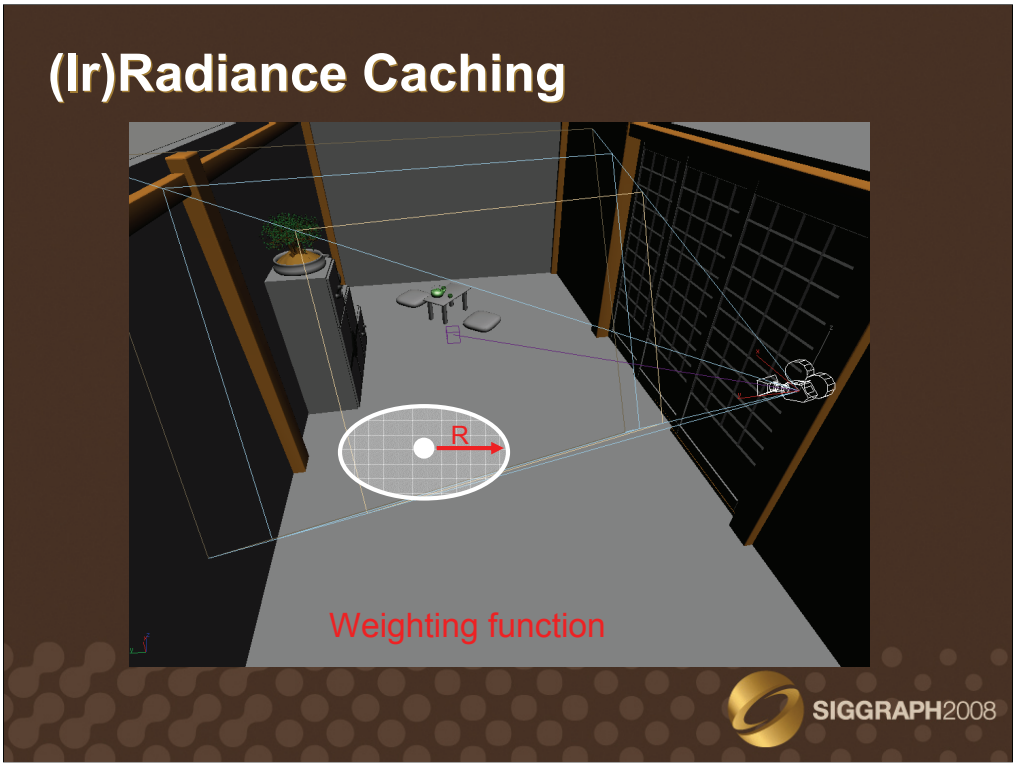
R&D Engineer

Thomson Corporate Research

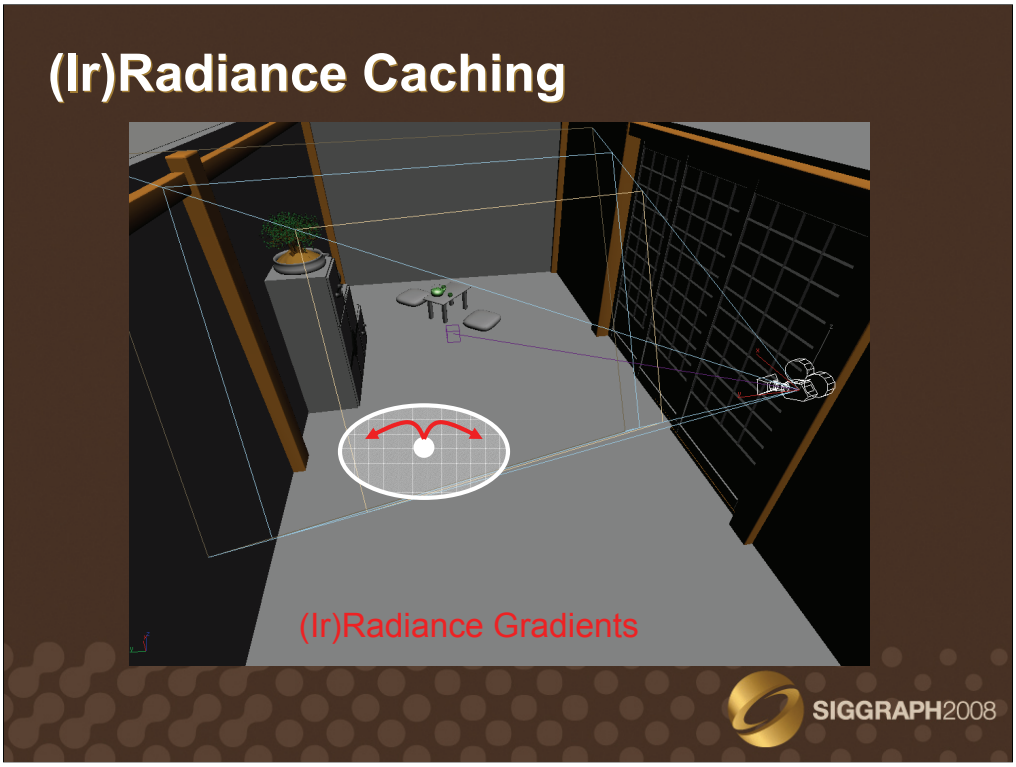
France

Course Outline

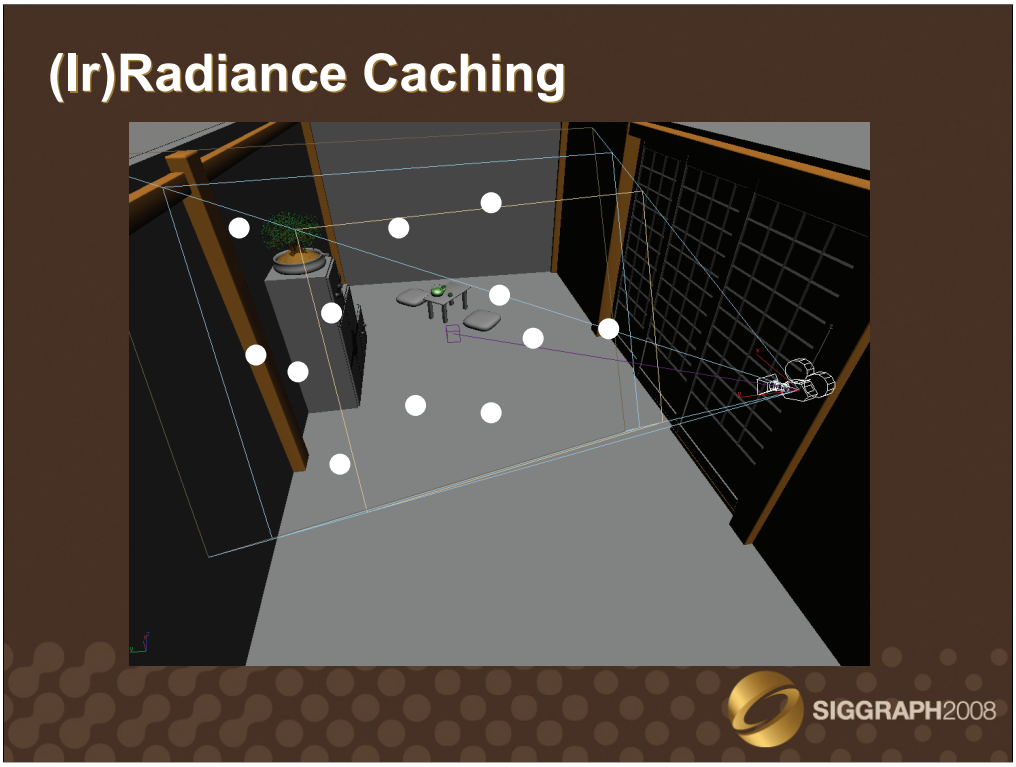
- Irradiance Caching: ray tracing and octrees
- A reformulation for hardware implementation
- Irradiance Cache Splatting
- GPU-Based hemisphere sampling



The irradiance caching algorithm is based on sparse sampling and interpolation of indirect diffuse lighting at visible points. Each irradiance record contributes to the indirect lighting of points within its zone of influence. The size of this zone is adapted according to the mean distance R to the surrounding objects using the irradiance weighting function [Ward88].



The irradiance value at points within the zone of influence of a record can be extrapolated using irradiance gradients [Ward92, Krivanek05a, Krivanek05b].

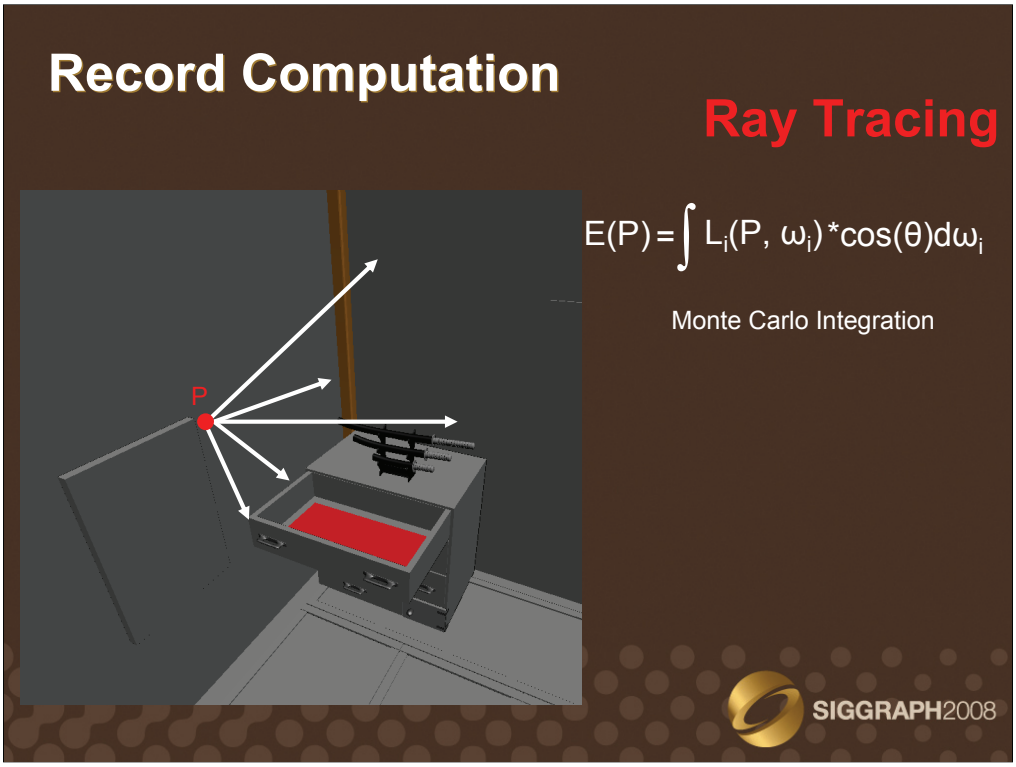


When the zones of influence of the records cover the entire zone visible from the viewpoint, the image representing the indirect lighting can be rendered. The irradiance caching algorithm is then divided into three steps:

-The computation of the records

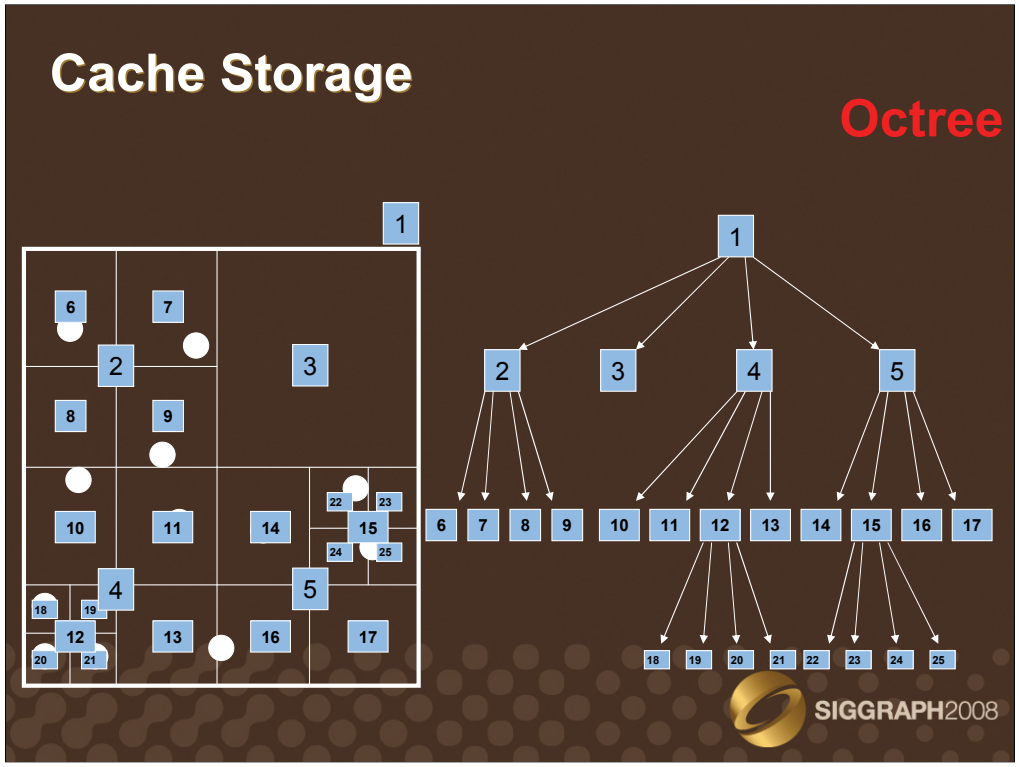
-The records storage

-The estimation of the indirect lighting using nearby records

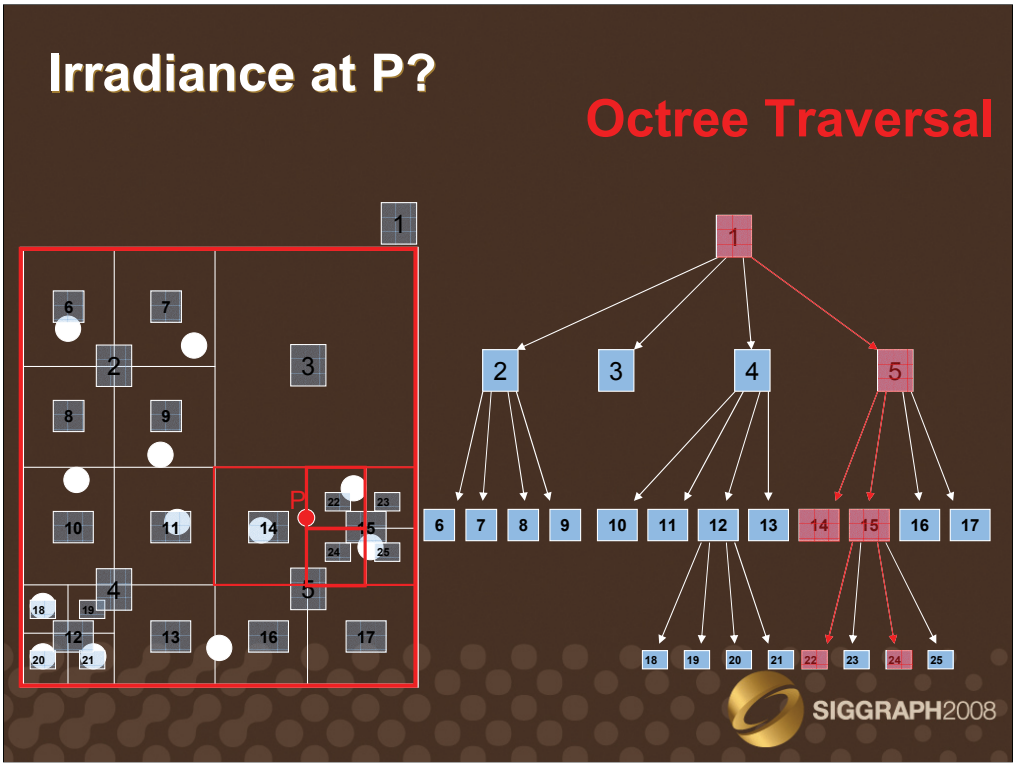


The computation of the irradiance value at a given point P requires the evaluation of the integral of the lighting over the surrounding hemisphere. This integral is typically estimated by Monte Carlo integration. Since the irradiance caching algorithm reuses the value of irradiance records over many pixels, the irradiance value of the record is computed with high precision, typically by tracing several hundreds to thousands rays.

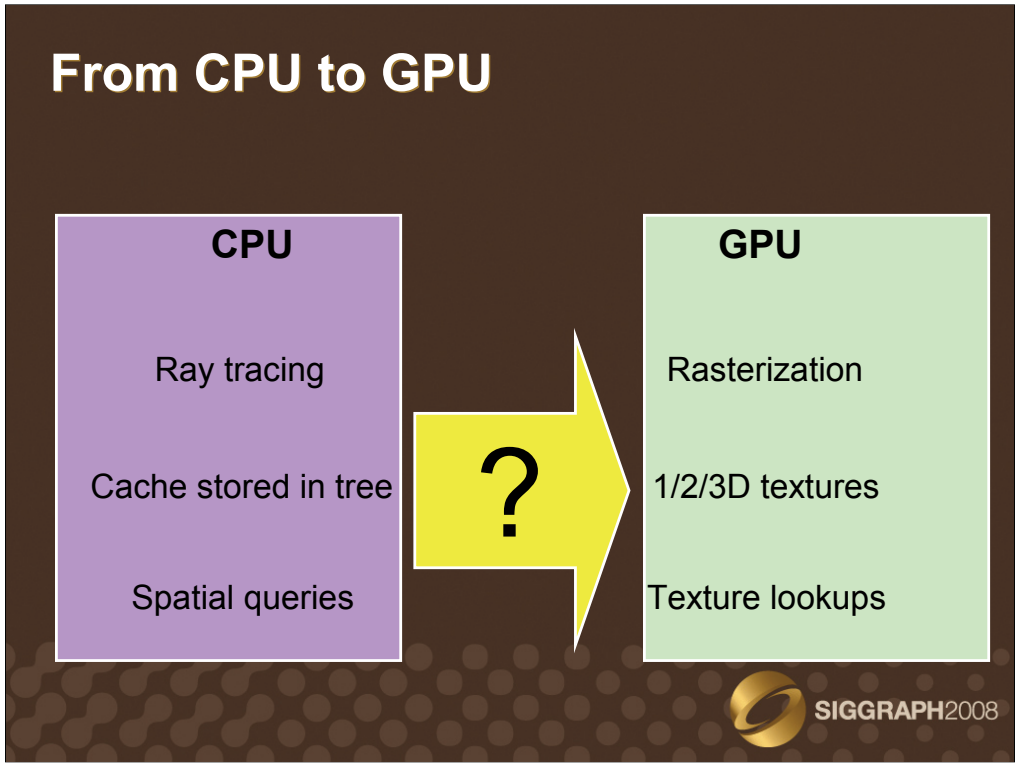
Once the irradiance value is computed, a record is created. This record contains the corresponding position, normal, irradiance, gradients, and mean distance to the surrouding objects.



The records are then stored in an octree, which allows for fast spatial queries. Note that the octree is a recursive data structure, which construction involves many conditional statements.



The irradiance at a point P can then be estimated efficiently by querying the octree for nearby records. However, this involves a traversal of the structure, which also involves many conditional statements.



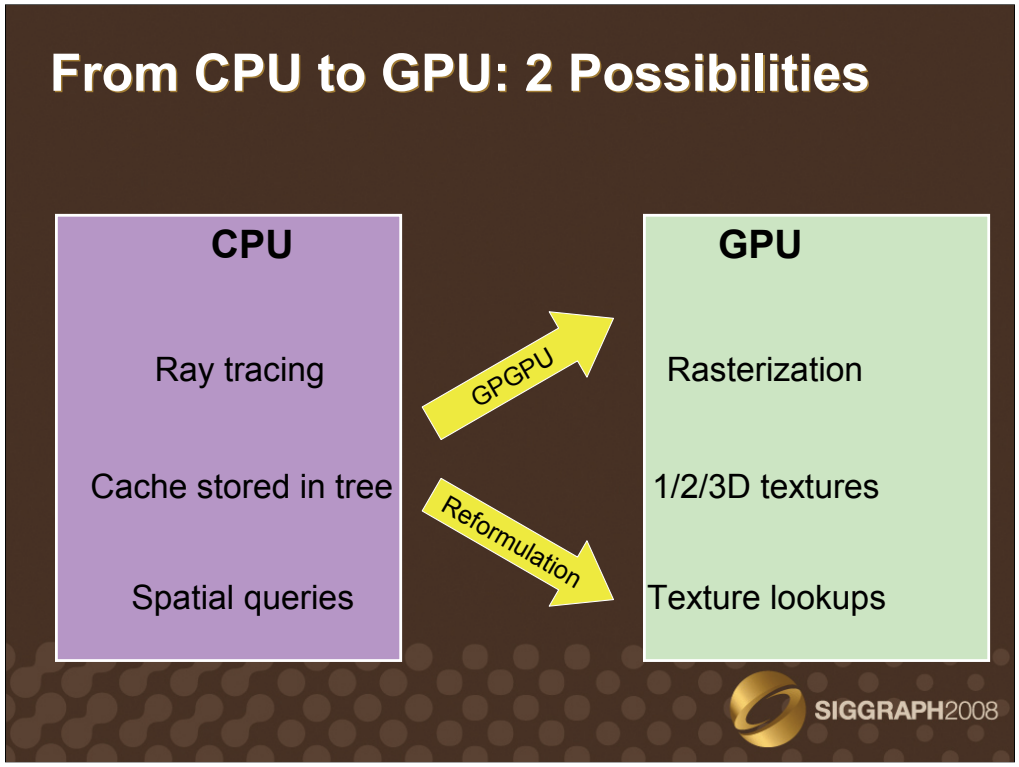
To summarize, the classical irradiance caching algorithm is implemented on the CPU, and is based on:

-Ray tracing

-Octrees

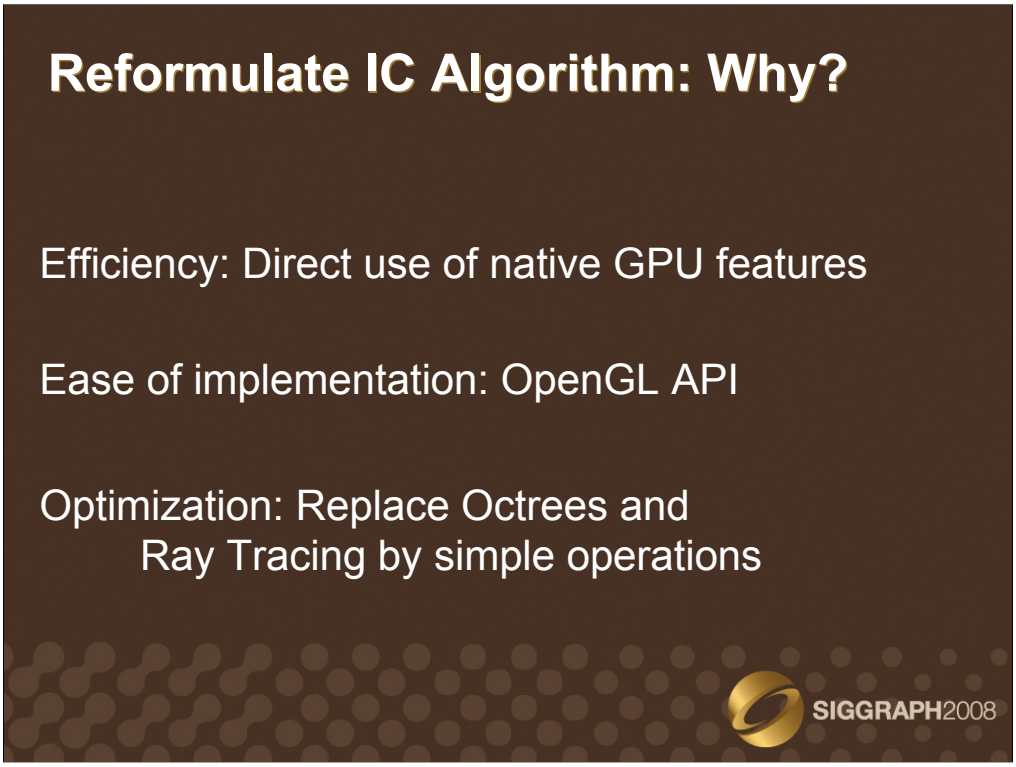
-Spatial queries in the octree

However, the GPU does not natively implement those operations: the visibility tests are performed using rasterization and Z-Buffering, and the only data structures available are 1, 2, and 3D textures. Particularly, the GPUs do not support pointers, which are the most common way of implementing recursive data structures.



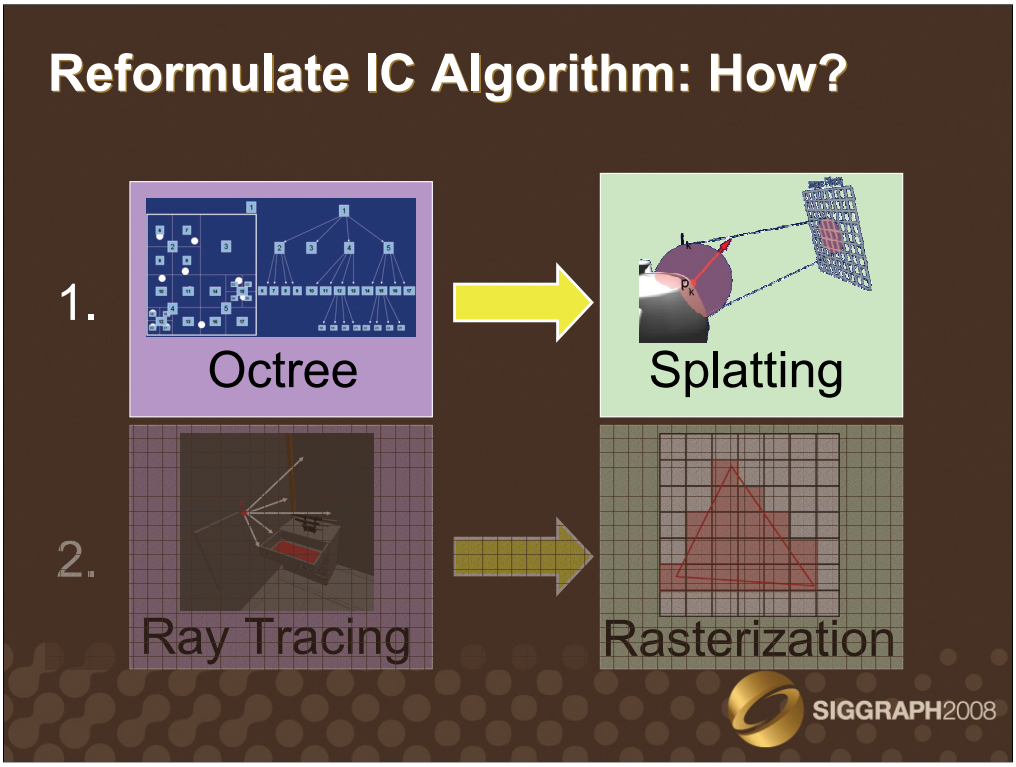
The irradiance caching algorithm cannot be directly mapped to the GPU architecture. Two methods can be considered: first, the use of libraries for general purpose computations on the GPU (such as Brook for GPU, http://graphics.stanford.edu/projects/brookgpu/). An important amount of research work has been performed to achieve interactive ray tracing on GPUs (such as [Purcell02, Purcell03]). A kD-Tree implementation for GPUs has also been proposed [Foley05].

An other way of performing irradiance caching on graphics hardware is to reformulate the algorithm so that only the native features of the GPU are used. This would allow us to get the best performance out of the graphics processors.

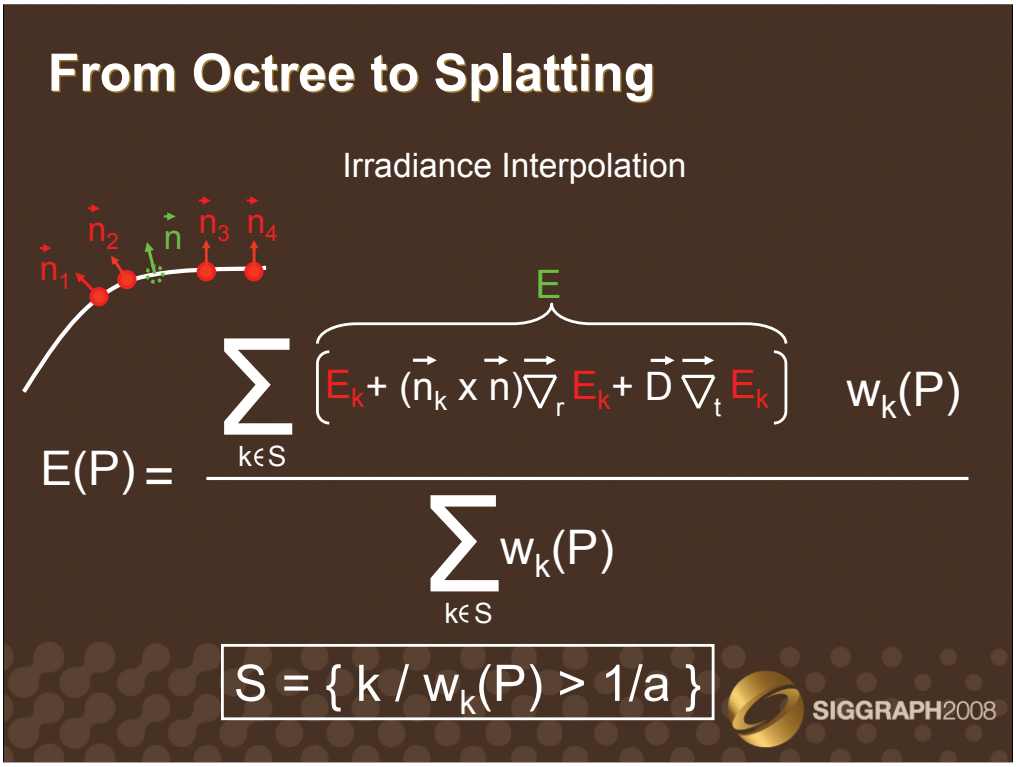


More precisely, we chose to reformulate the irradiance caching algorithm for three reasons. First, the direct use of native GPU features allow us to use each part of the GPU at its best, improving the performance. Second, the reformulated algorithm can be implemented directly using at 3D graphics API such as OpenGL or DirectX.

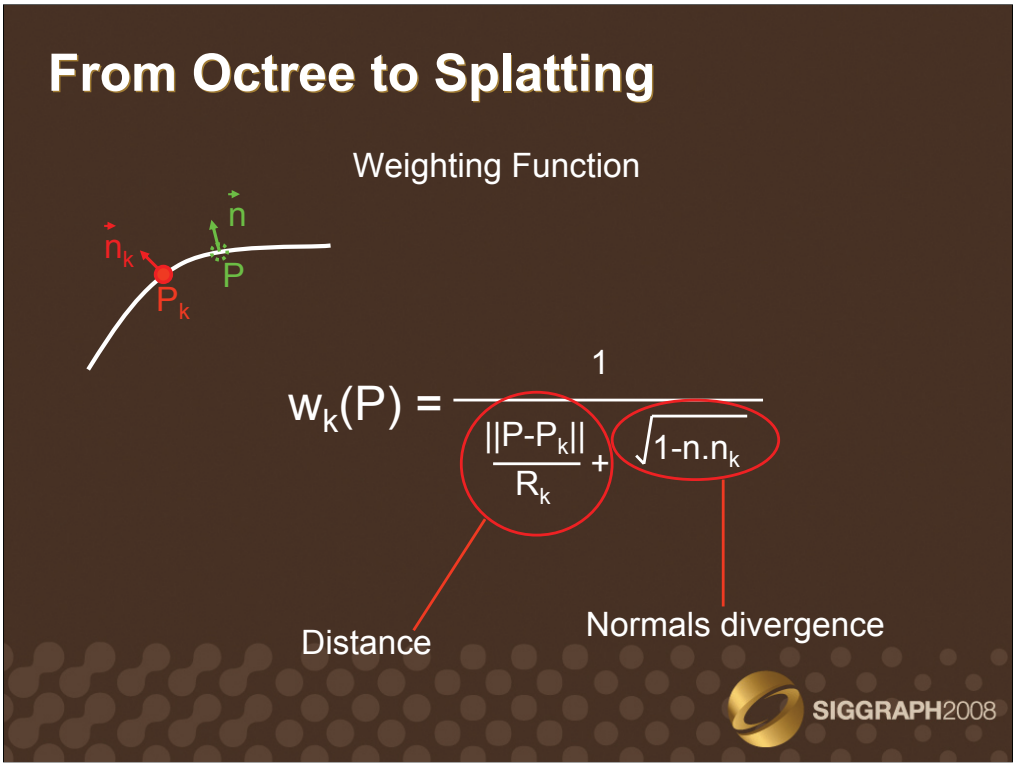
Third, this reformulation gives us the occasion of attacking two costly aspects of the irradiance caching algorithm: the hierarchical data structure, and the irradiance computation using ray tracing. Replacing those by more simple operations on the GPU increases the performance, yielding interactive frame rates in simple scenes.



The reformulation is divided into two tasks: the replacement of the octree by a splatting operation, and the use of rasterization in place of ray tracing.



The reformulation is based on the irradiance interpolation equation [Ward88, Ward92] presented before: the irradiance estimate at a point P is the weighted average of the contributions of the surrounding records. The contribution of each record is computed using irradiance gradients for translation and rotation. The set S of contributing records is defined as the set of records for which the weigthing function evaluated at P is above a user-defined threshold a.

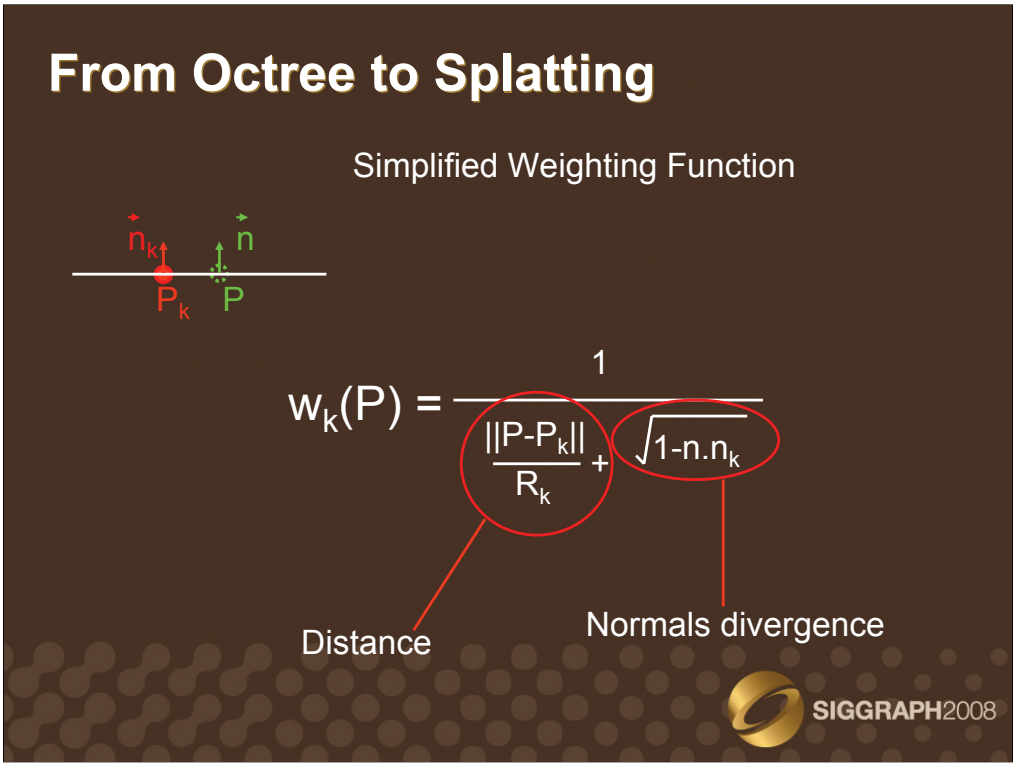


The weighting function is very simple, and depends on:

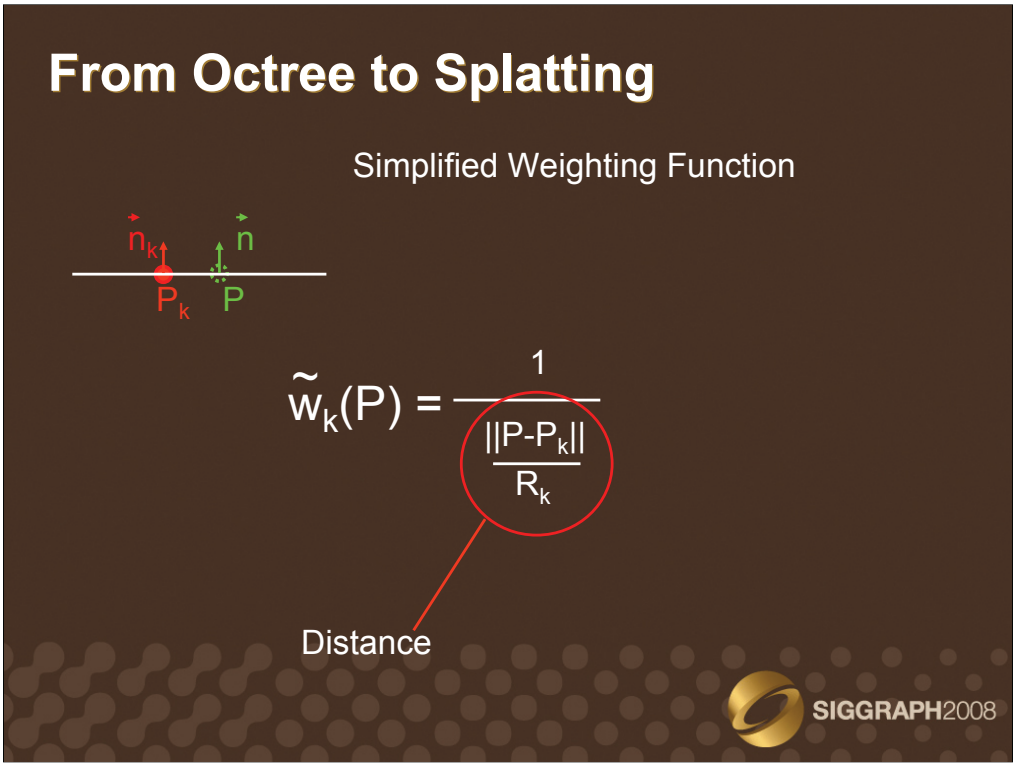
-The distance between the record location and the point P

-The mean distance R to the surrounding objects

-The divergence of the surface normals between the record location and the point P



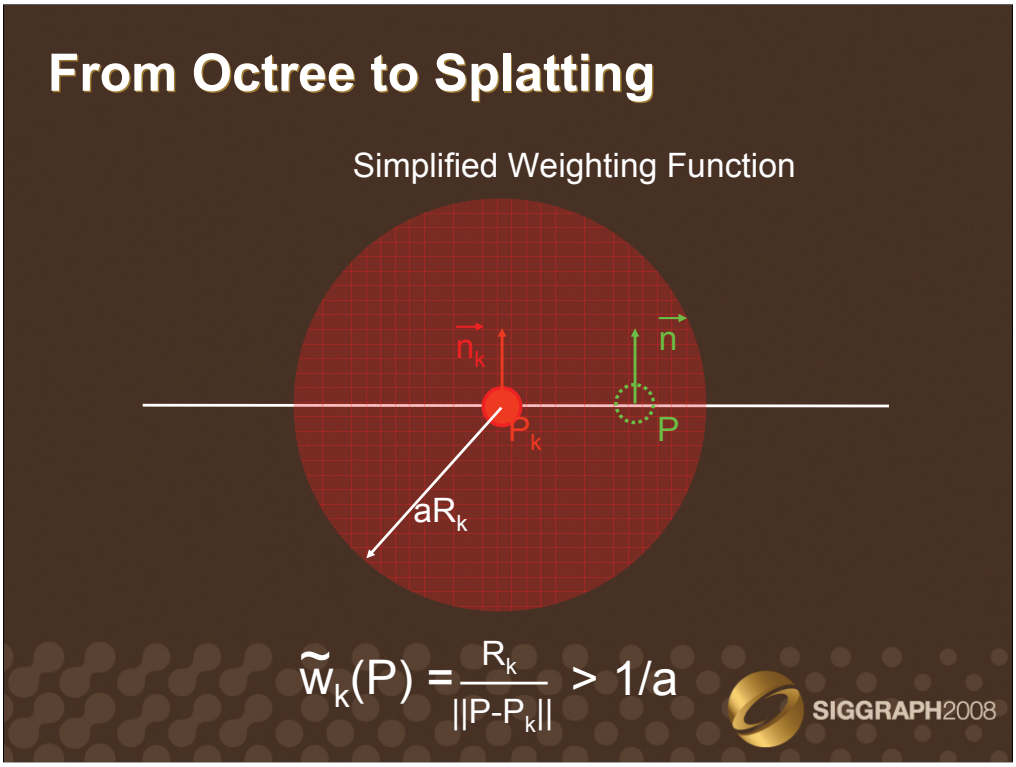
If we assume that the record location and the point P **always have the same normal**, the weighting function can be simplified by removing the dependence to the surface normals.



This yields a simplified weighting function, which depends only on two factors:

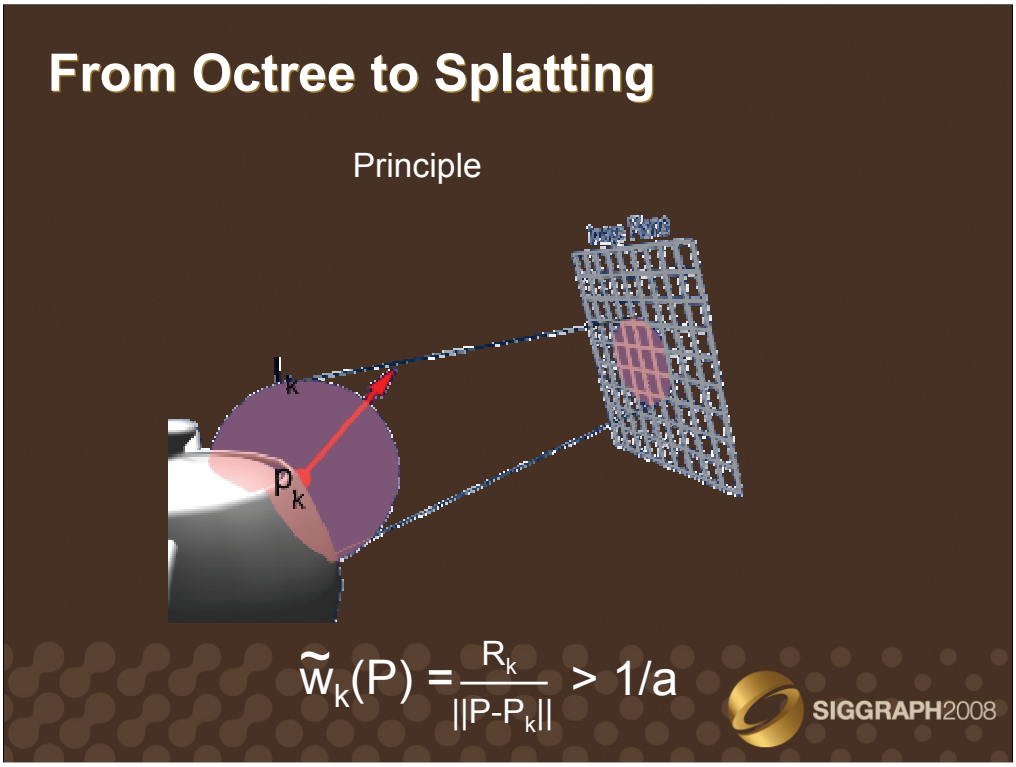
-The distance between the record location and the point P

-The mean distance to the surrounding objects

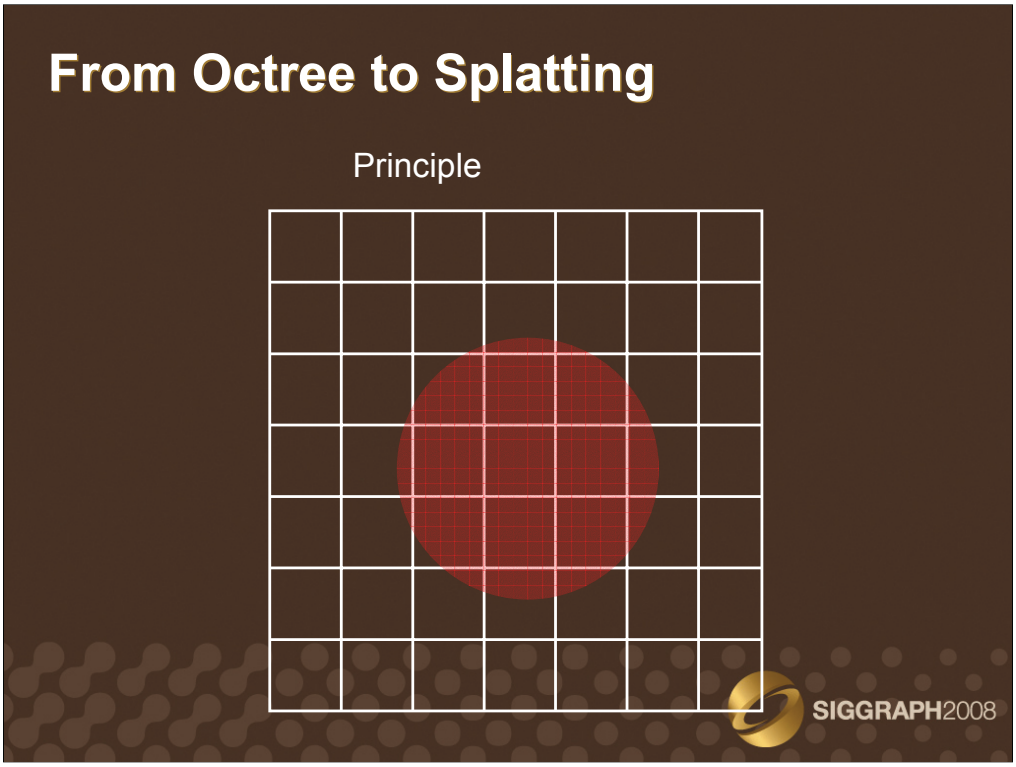


Using this simplified weighting function, a record k contributes to **all** points located within a sphere centered at the record location, with radius aRk. Hence this radius depends not only on the user-defined parameter a, but also on the mean distance to the surrounding objects.

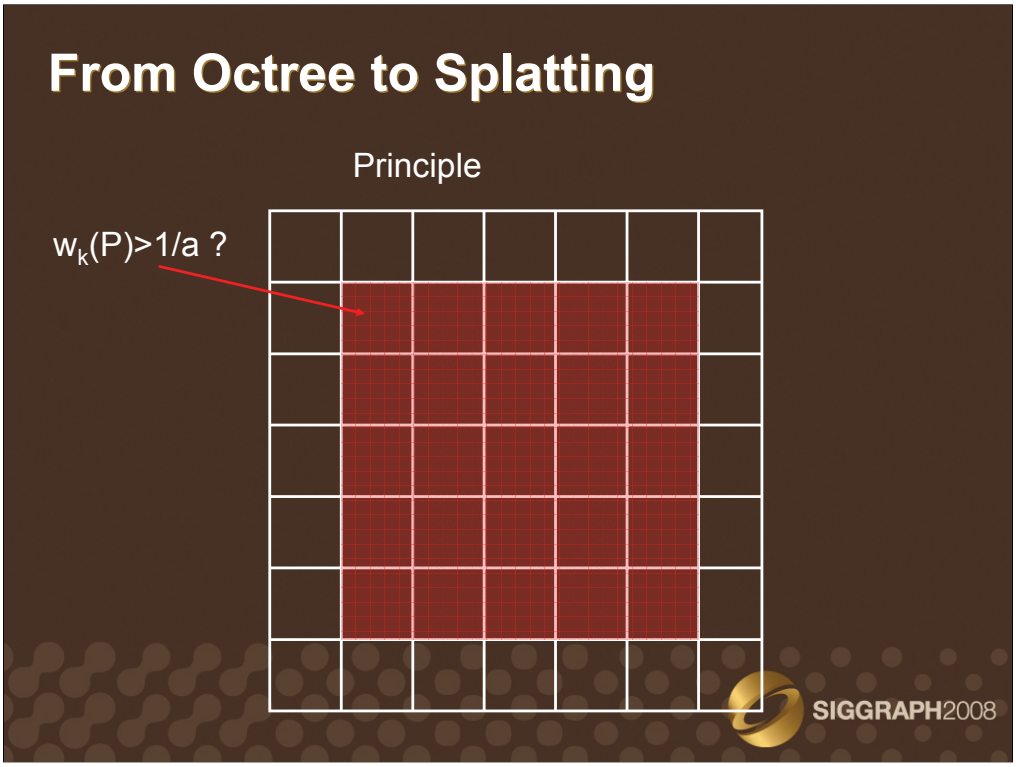
Note that the simplified weighting function removes the constraint on the surface normals. Therefore, the set of points at which the record actually contributes (with respect to the full weighting function) is a subset of the points located within the sphere.



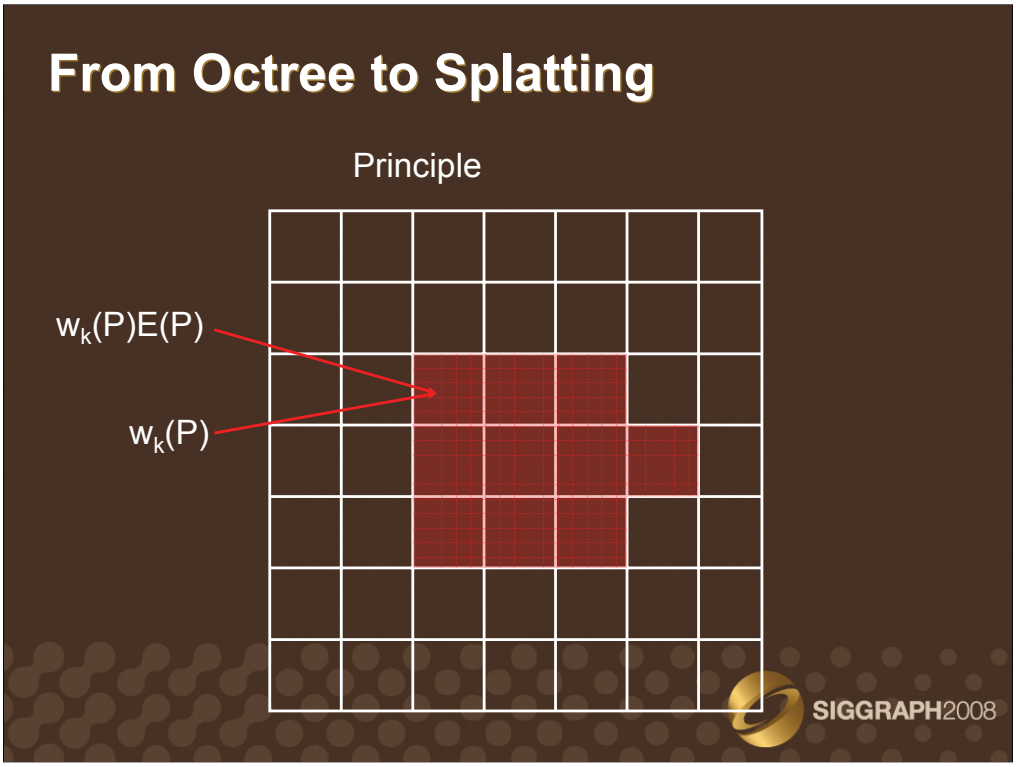
The sphere can be splatted onto the image plane. Hence the covered pixels correspond to the visible points at which the record may contribute.



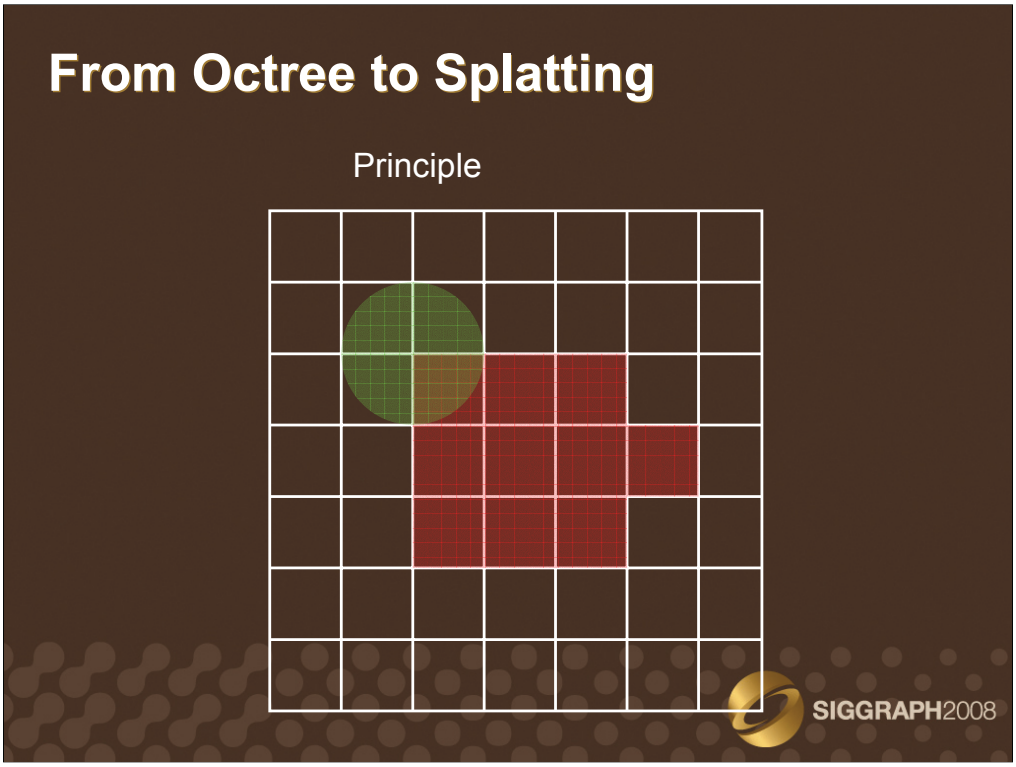
Let us consider the image plane, on which the sphere has been splatted. The splatted sphere encloses **all** the visible points at which the considered record may contribute (with respect to the full weighting function). Our goal is now to select the points for which the condition on the **full** weighting function is satisfied, that is $w_k(P) > 1/a$.



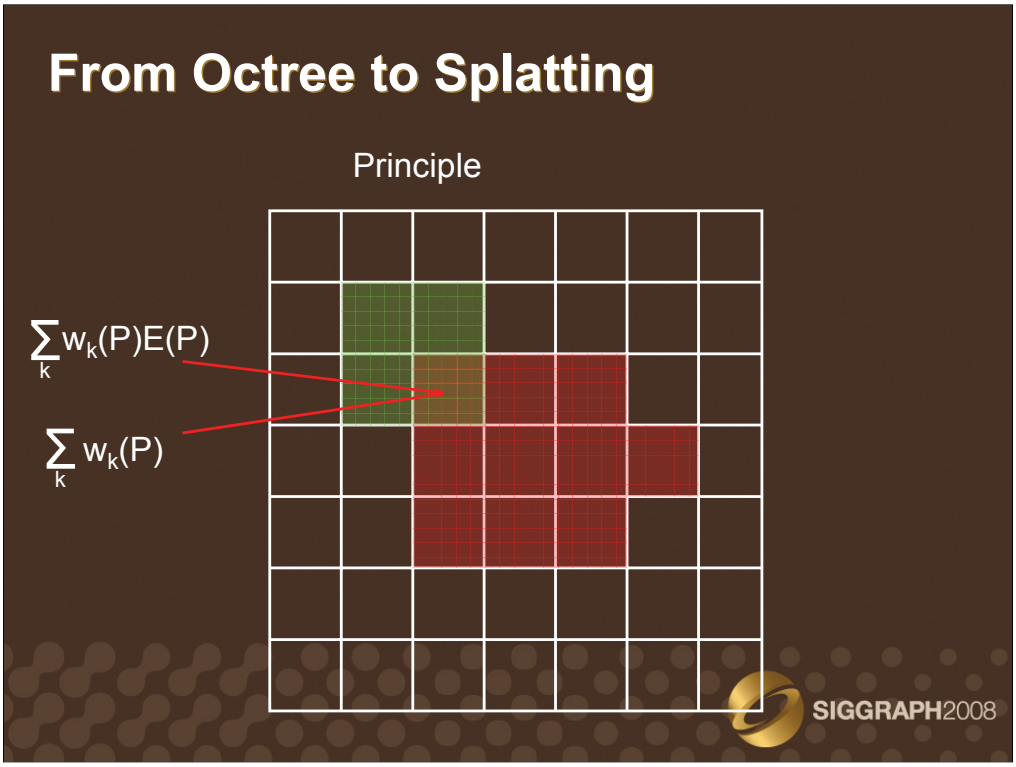
For the convenience of implementation, we use a quadrilateral tightly enclosing the splatted sphere. For each point visible through the pixels of the quadrilateral, the full weighting function is evaluated, and tested against the user-defined threshold. If the condition is not satisfied, the pixel is discarded.



This yields a set of pixels, corresponding to the set of visible points at which the record actually contributes. At those points, we compute separately the weighted contribution of the record (with respect to the full weighting function and to the irradiance gradients) and the weight of the contribution. This information can be easily stored within floating point RGBA pixels, using RGB for the weighted contribution, and the alpha channel for the weight value.

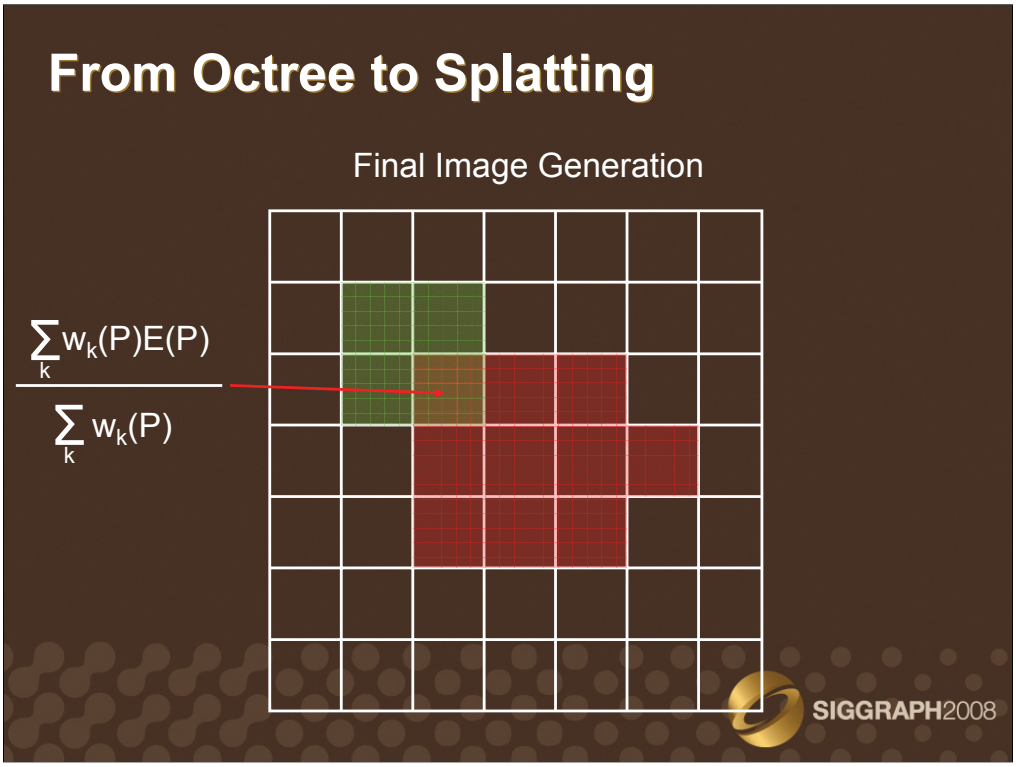


When splatting an other record, the same operations are to be done. However, the zones of influence of the two records overlap.



In this case, we first compute the weighted contribution and weight of the second record as described before. Then, the built-in alpha blending of graphics processors adds the contributions and weights together in the overlapping area.

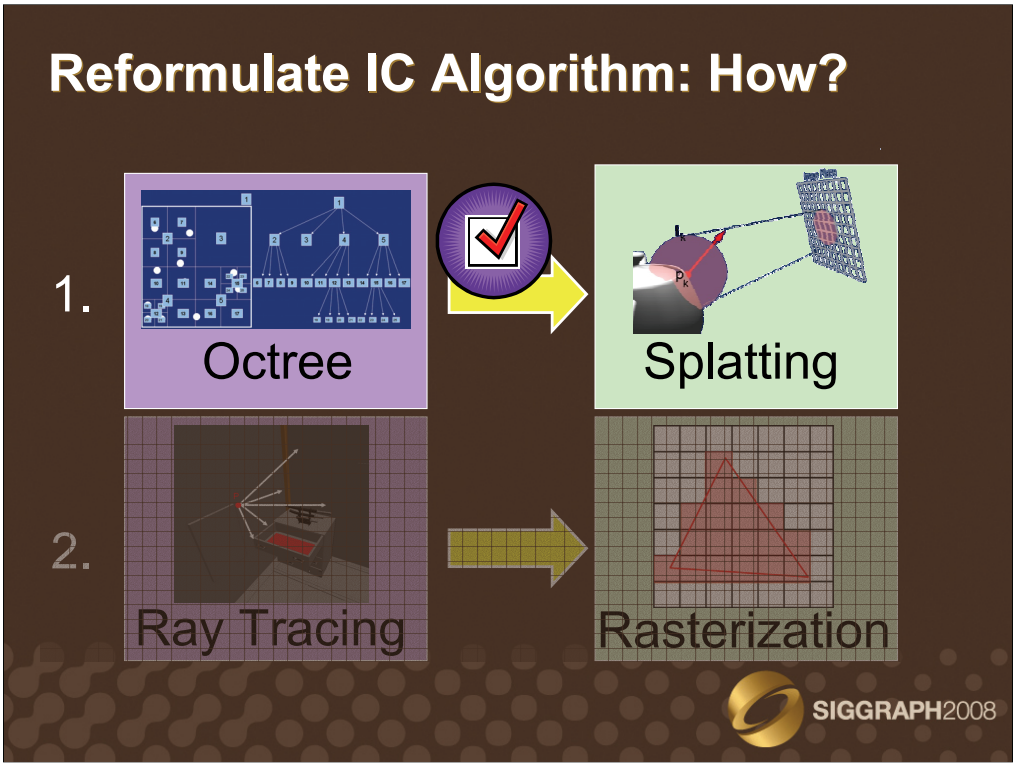
Then, each pixel contains both the weighted sum of the contributions, and the sum of the contribution weights.

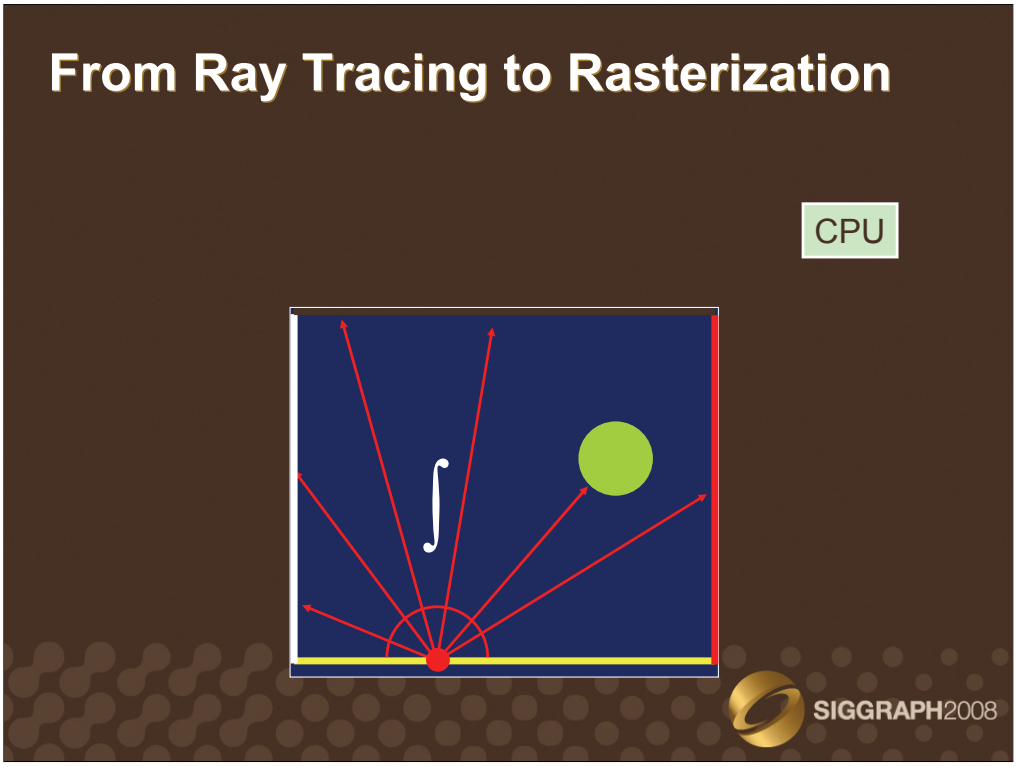


Once all the necessary records have been splatted, the result of the irradiance interpolation equation can then be obtained by dividing the weighted sum of contributions by the sum of the weights. This operation can be easily performed within a fragment shader, by dividing the RGB components (that is, the weighted sum of contributions) by the alpha channel (the sum of weights).

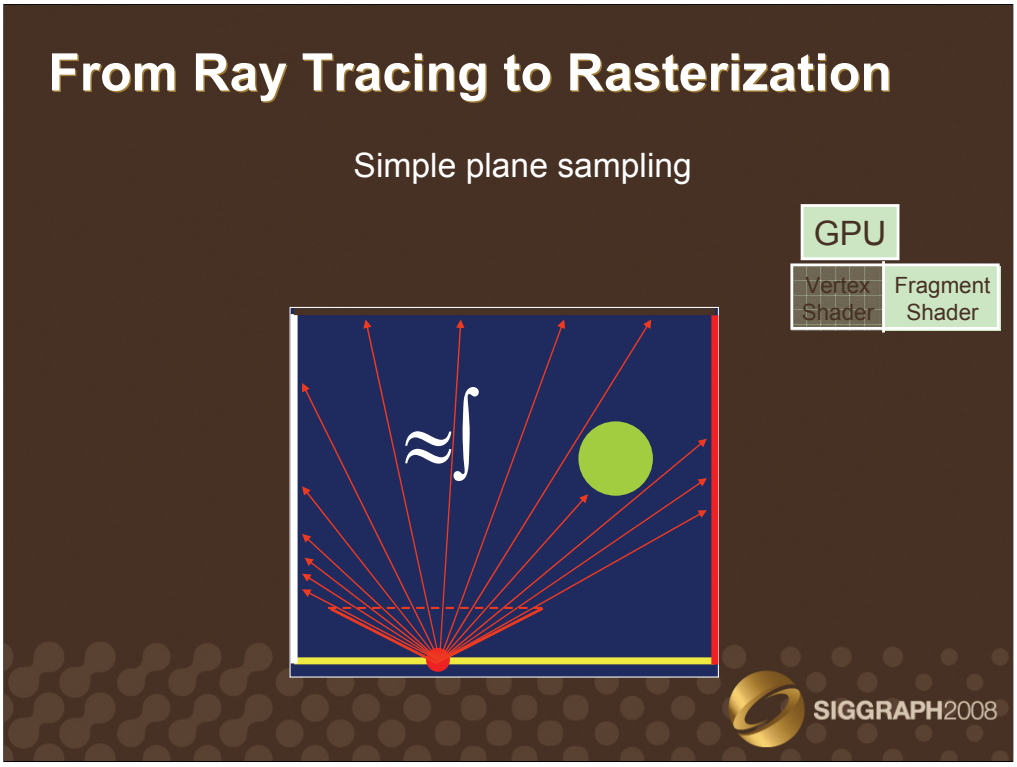
To summarize, let us consider an example. At the beginning of the algorithm, the cache contains no records. Hence the generated image only features direct illumination. Then, we add a record on the back wall. This record contributes to the points within its neighborhood. When adding a second record, the zones of influence overlap. For the pixels within the overlapping area, the estimated irradiance is calculated using the weighted average of the contributions. Other records are successively added to the cache, until the entire visible area of the scene is covered by the zones of influence of the records. The resulting image features both direct and indirect lighting for every visible point.

Note that for explanation purposes, the reconstruction of the indirect lighting is very coarse to highlight the zones of influence of each record. Also, the gradients are not used. As with the classical irradiance caching algorithm, high quality can be obtained by using irradiance gradients and setting an appropriate value for the user-defined parameter a.


Reformulate IC Algorithm: How?
1.
Octree
Splatting
2.
Ray Tracing
Rasterization
SIGGRAPH2008



In classical irradiance caching, the irradiance at a point is estimated by Monte Carlo ray tracing: random rays are traced through the scene, gathering the lighting incoming from the surrounding environment. This estimate is usually computed on the CPU.



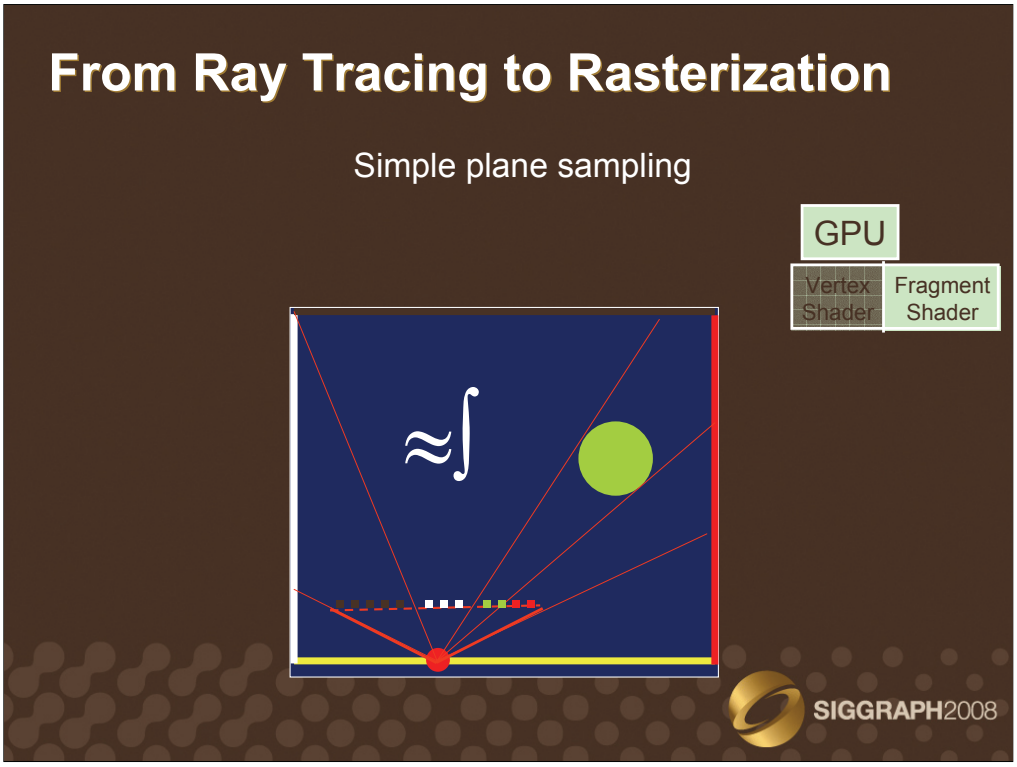
A well-known approximate method for hemisphere sampling on the GPU is the simple plane sampling: a virtual camera with a large aperture is placed at the point of interest. The scene is then rendered on graphics hardware, yielding an image of the surrounding objects. On recent graphics hardware, this data can be computed in high dynamic range (HDR) using floating-point render target and programmable shaders. The shadowing effects can be efficiently accounted for using fast shadowing techniques such as shadow mapping [Wil78].

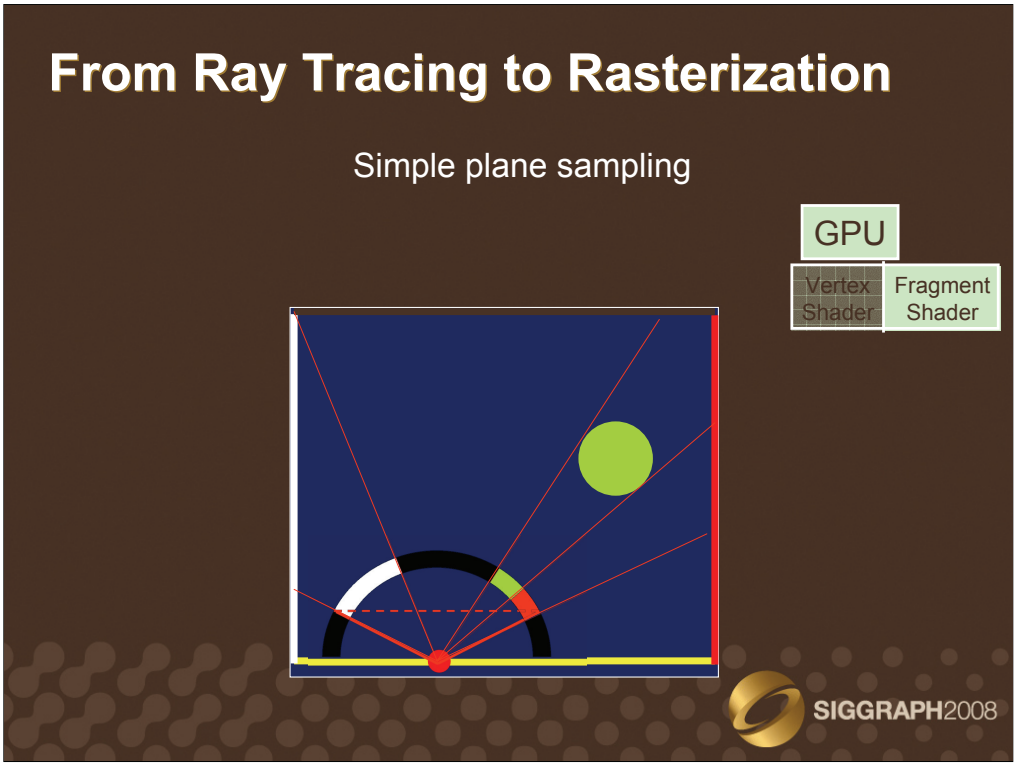
The solid angle subtended by a pixel p is defined as $\Omega_p = A*\cos(\theta)/(d*d)$ where:

•A is the surface of a pixel

•θ is the angle between the direction passing through the pixel and the surface normal

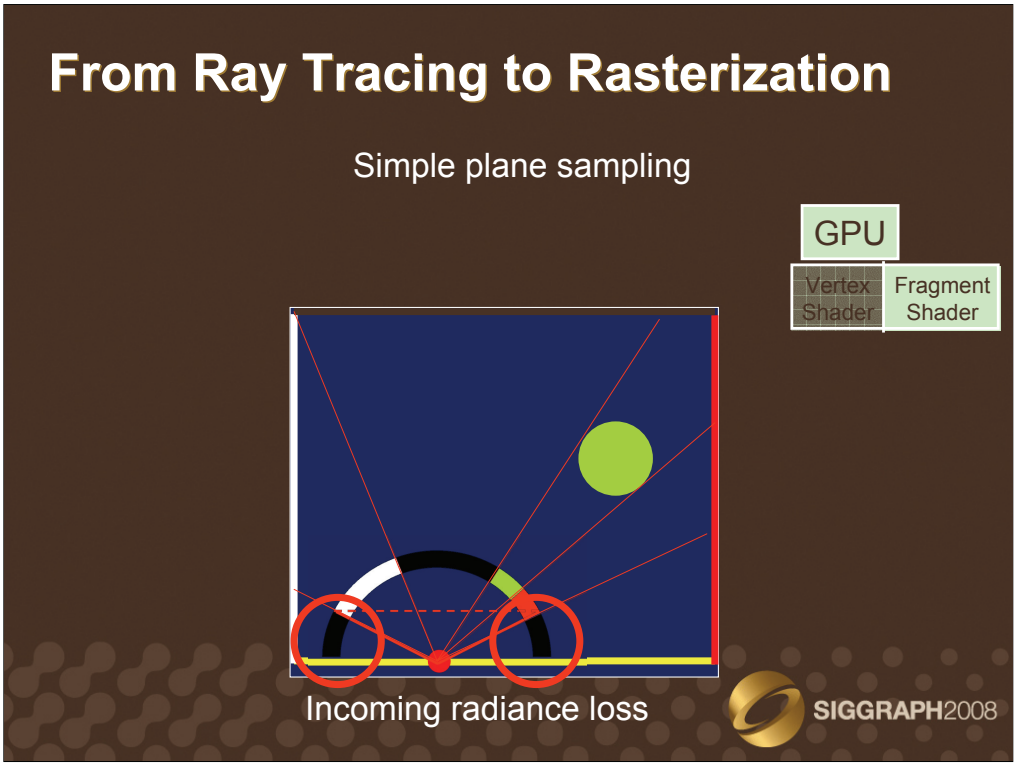
•d is the distance between the point of interest and the image plane

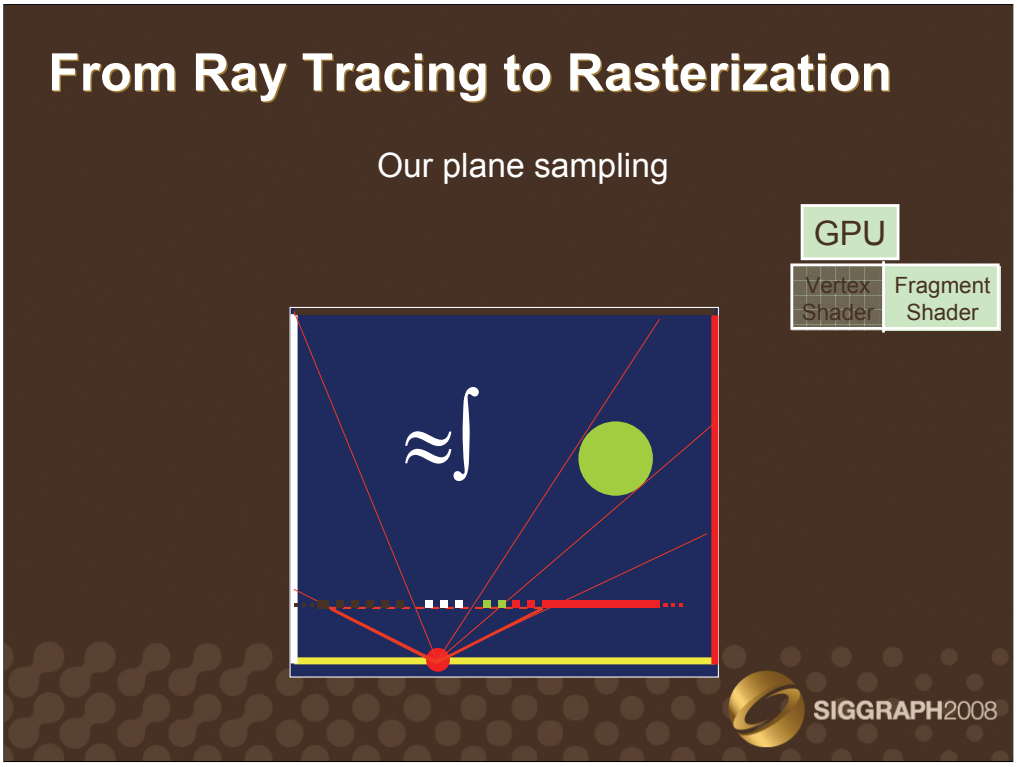

From Ray Tracing to Rasterization
Simple plane sampling
GPU
Vertex Shader
Fragment Shader
≈∫
SIGGRAPH2008



However, the aperture of the camera cannot allow us to sample the entire hemisphere: a perspective camera is represented by a perspective projection matrix which is applied to the visible contents of the scene. In OpenGL, such a camera is modeled using the gluPerspective function, whose values are calculated using the field of view (FOV) of the camera. More precisely, a key value in this matrix is f = cot(FOV/2), which is undefined for FOV=180°. Furthermore, using a very large aperture such as 179.999° leads to important perspective deformation and sampling problems.

In [LC04], Larsen et al. show that an aperture of 126.87° is sufficient for capturing 86% of the incoming directions. However, the remaining 14% are unknown and must be compensated to avoid a systematic underestimation of the incoming radiance. Furthermore, this compensation must respect the directional information of the incoming radiance to allow for a later implementation of radiance caching for global illumination computation on glossy surfaces.


From Ray Tracing to Rasterization
Simple plane sampling
GPU
Vertex Shader
Fragment Shader
Incoming radiance loss
SIGGRAPH2008

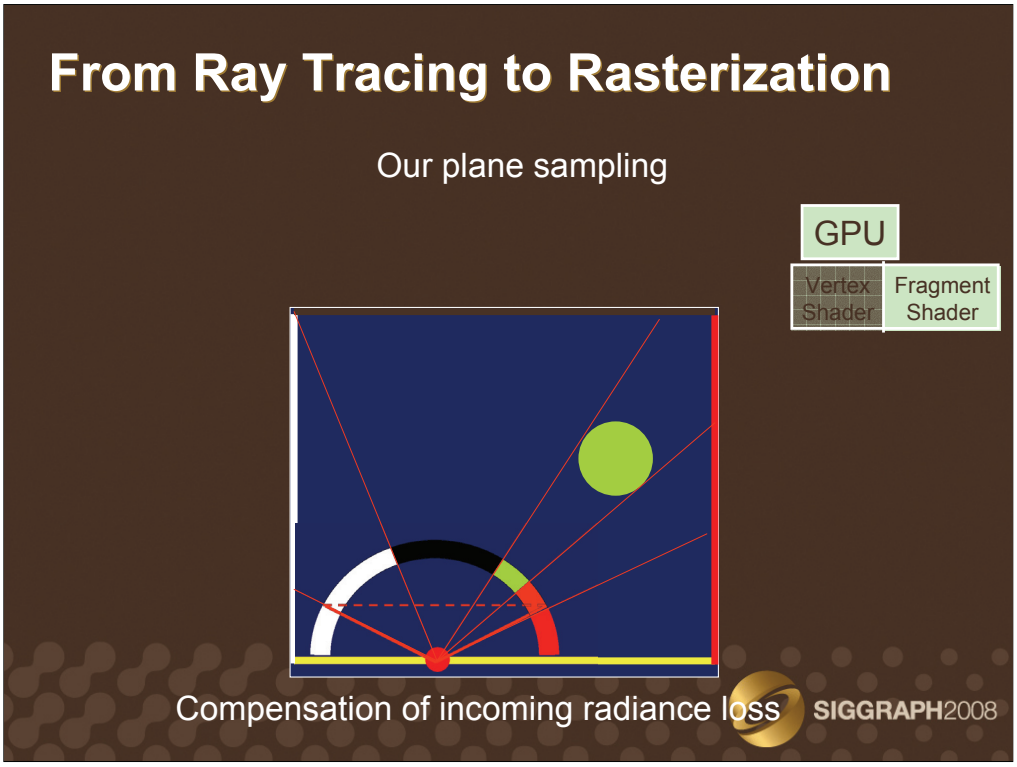


We propose a very simple compensation method, in which the border pixels are virtually « extended » to fill the parts of the hemisphere which have not been actually sampled. To this end, border pixels are considered as covering a solid angle of:

$\Omega_{border} = \Omega_p + \cos(\theta_{border}) * \delta_\Phi$

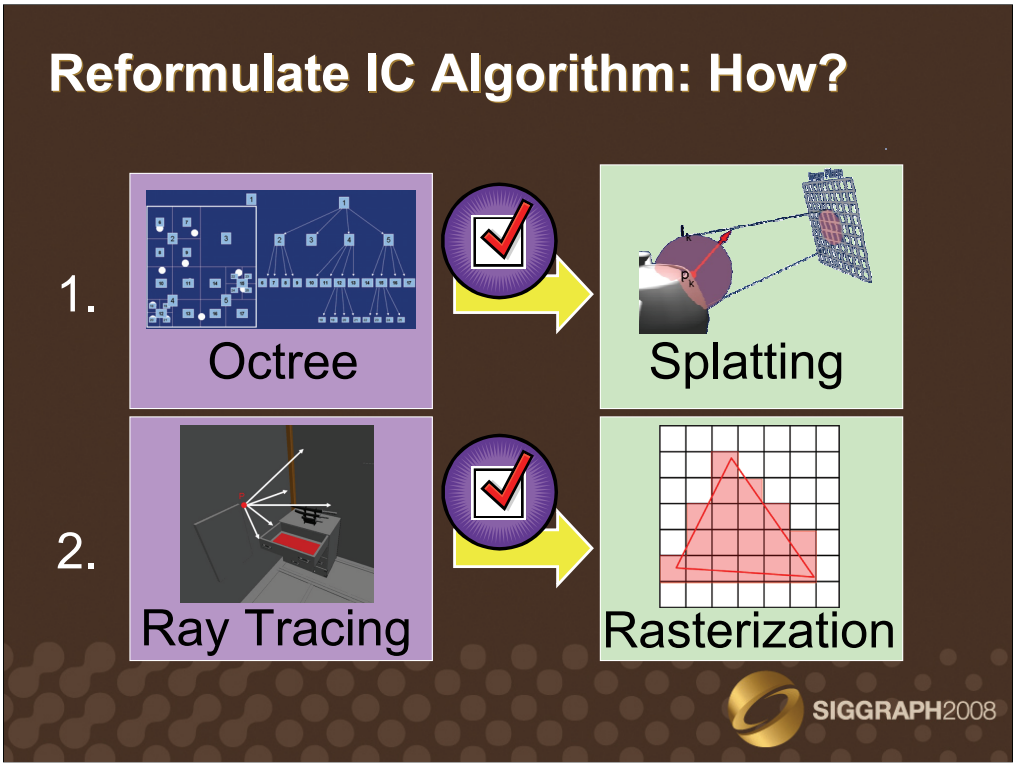
Where:

- Ω_p is the solid angle subtended by the pixel
- θ_{border} is the largest θ covered by the aperture of the camera
- δ_Φ is the interval of Φ spanned by the pixel

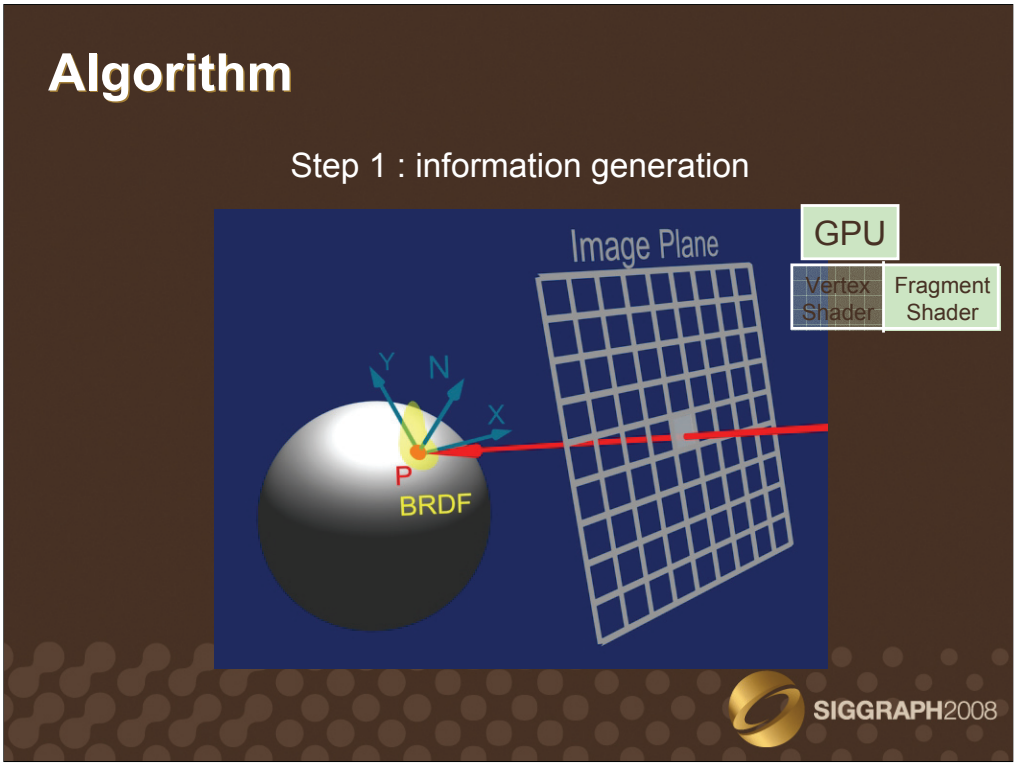


This allows us to compensate the missing information by extrapolating the radiance values at the extremities of the sampling plane. This extrapolation provides a plausible estimate of the missing radiances, hence making it suitable for radiance caching also.

It must be noted that other techniques can be used, such as the full hemicube sampling (which requires several passes), or a hemispherical parametrization of the hemisphere in vertex shaders.

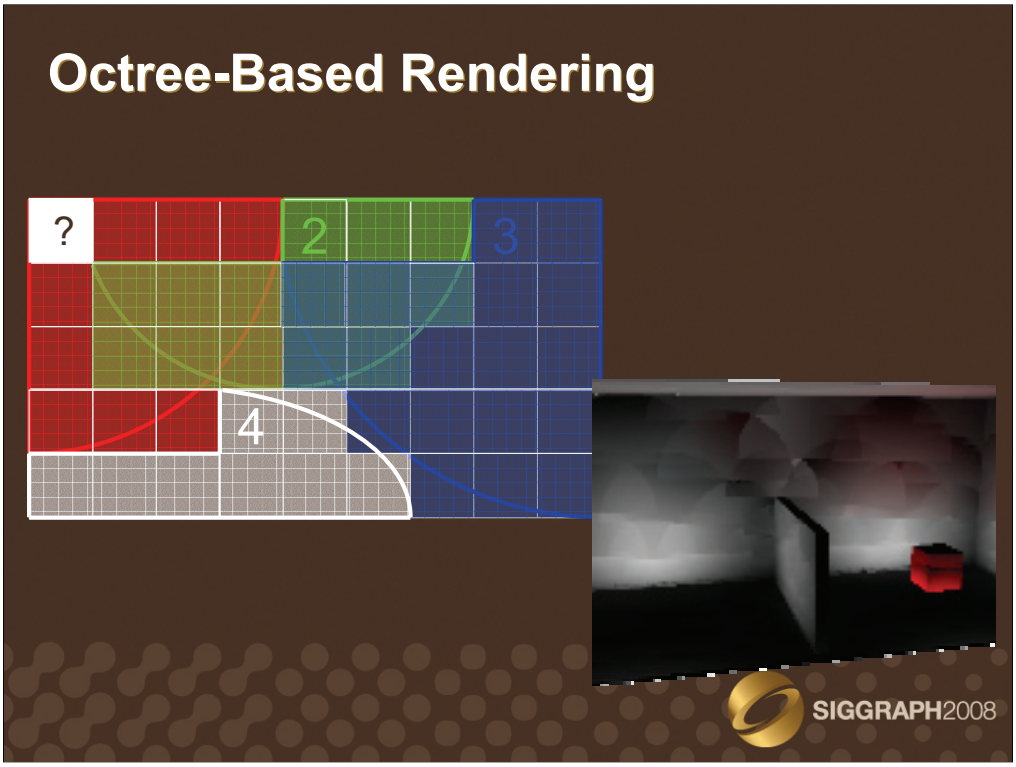


Now the algorithm has been reformulated for implementation on graphics hardware. Next section will present the entire algorithm for global illumination computation using irradiance splatting.



The first step of the algorithm consists in obtaining basic information about the points visible from the user point of view. In particular, this information includes the position and normal of the visible points. This data can be easily generated in one pass on the GPU using floating-point multiple render targets and programmable shaders.

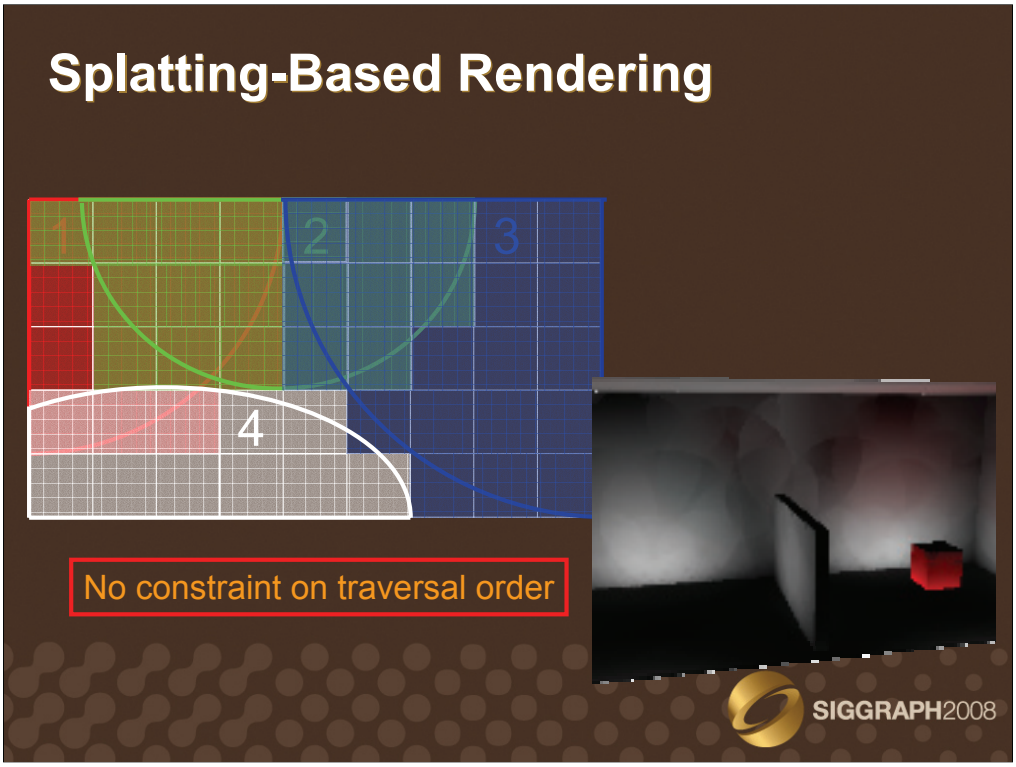
In the next step, the algorithm transfers this data into the main memory. The CPU is then used to traverse the image and detect where new records are required to render the image.



Usually, the detection is performed by querying the irradiance cache (hence the octree) for each pixel as follows:

- For each visible point P with normal N corresponding to pixel p
    - W = GetSumOfContributions(P,N)
    - If (W < a)
        - R = CreateNewRecord(P,N)
        - IrradianceCache.StoreRecord(R)
        - p.radiance = R.irradiance*SurfaceReflectance(P)
    - Else
        - E = EstimateIrradiance(P,N)
        - P.radiance = E*SurfaceReflectance(P)
    - EndIf
- EndFor

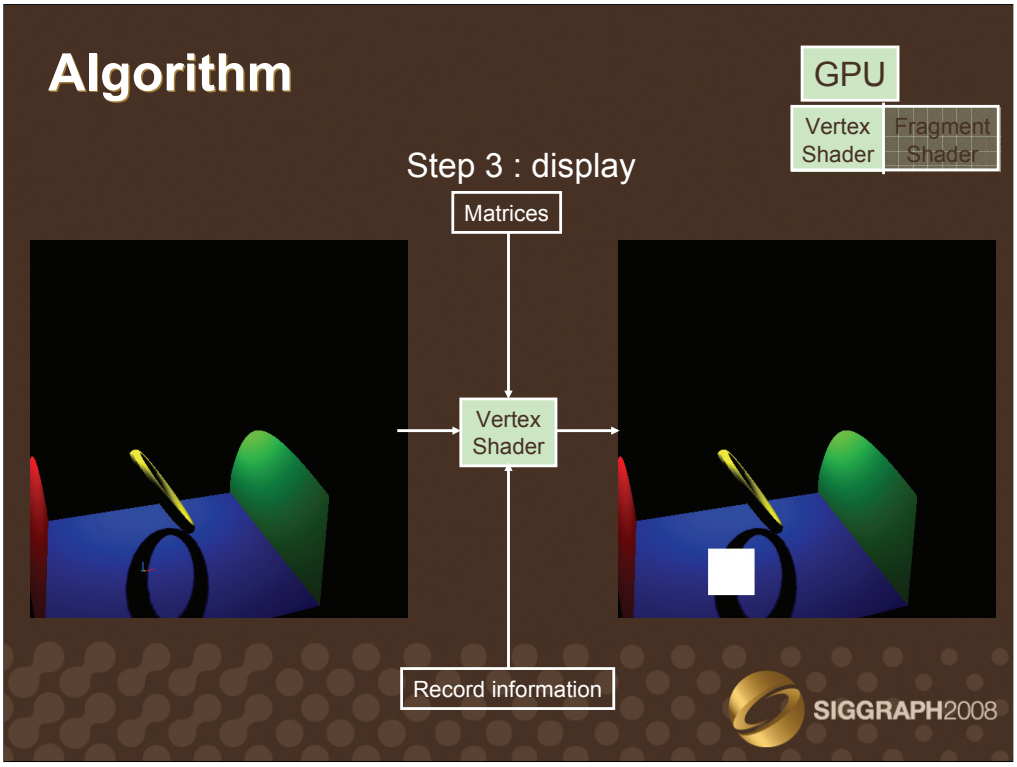
While this algorithm ensures the presence of a sufficient number of records, such records are not used in an optimal way: when record 2 is created, the algorithm only propagates the contribution of 2 to the pixels which have not been checked yet. The previous pixels remain unchanged, even though record 2 may contribute to their radiance. If the image is traversed linearly, disturbing artifacts may appear: in this example the image is traversed from bottom to top. Therefore, the records contribute only to the points located above them in the image. This problem is usually compensated by using other traversal algorithms, typically based on a hierarchical subdivision of the image.



When using irradiance splatting, the detection code becomes:

- For each visible point P with normal N corresponding to pixel p
  - W = GetSumOfContributions(P,N)
  - If (W < a)
    - R = CreateNewRecord(P,N)
    - IrradianceCache.StoreRecord(R)
    - SplatRecord(R)
    - p.radiance = R.irradiance*SurfaceReflectance(P)
  - Else
    - *// Nothing to do: the radiance value depends of surrounding records and may be updated*
  - EndIf
- EndFor

In our method, the records are splatted onto their entire zone of influence: even pixels which have been previously checked can be updated by the addition of a novel record. Hence this method removes the constraint on the image traversal order. In the example image, we used a simple linear traversal. The zone of influence of each record is completely accounted for by our splatting method.

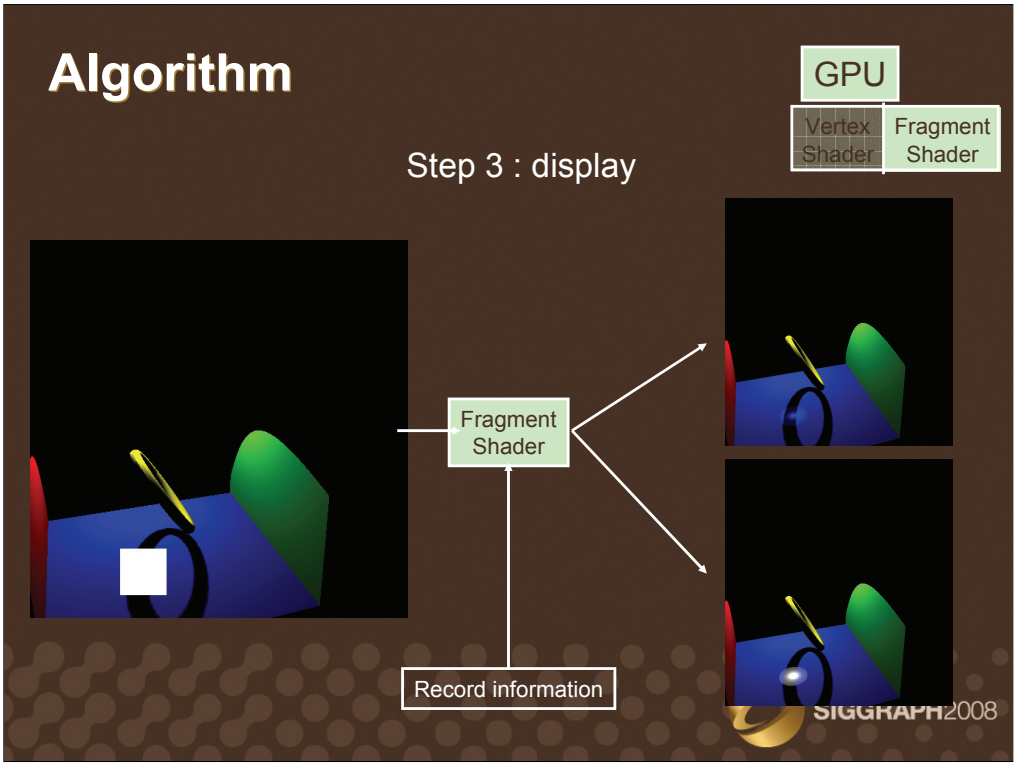


Once all records have been computed, each record must be rendered on the GPU. Let us consider a record located at point P with normal N, and with an harmonic mean distance to surrounding objects H. The zone of influence of the record is contained within a screen-aligned square. Let us consider the vertices of a square such that:

v0 = (1.0, 0.0, 0.0)
v1 = (1.0, 1.0, 0.0)
v2 = (0.0, 1.0, 0.0)
v3 = (0.0, 0.0, 0.0)

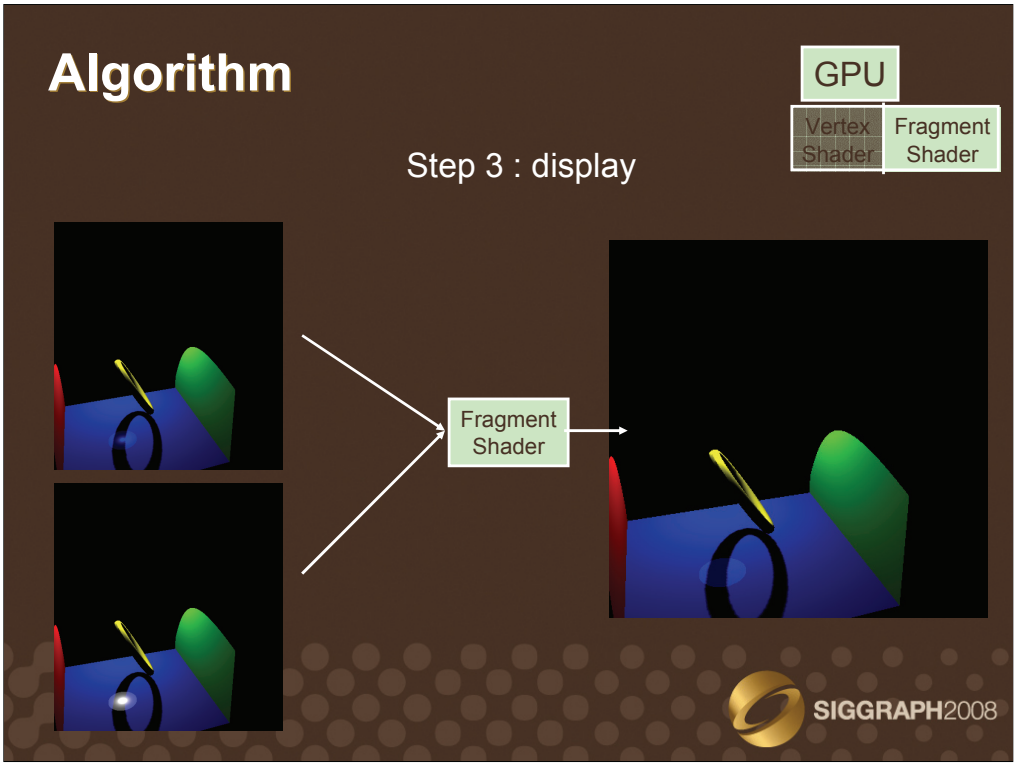
Each vertex can be transformed into screen space using the following function:

```
float4 TransformVertex(Vertex v)
{
                        float4 projPos = ModelViewMatrix*P; // Transform the point into camera space
                        float scale = projPos.w;
                        float4 projPos /= projPos.w;

                        float radius = a*H; // Radius of the zone of influence in camera space

                        float4 radiusV = float4(radius, radius, 0.0, 0.0);
                        // Project the corner of the bouding box into screen coordinates
                        float4 topRight = ProjectionMatrix*(projPos + radiusV*scale);
                        float4 bottomLeft = ProjectionMatrix *(projPos - radiusV*scale);
                        topRight /= topRight.w;
                        bottomLeft /= bottomLeft.w;

                        float2 tr = topRight.xy;
                        float2 bl = bottomLeft.xy;
                        float2 delta = tr - bl;

                        // return the screen coordinates of the vertex
   return float4(
     bl.x + vertex.x*delta.x,
     bl.y + vertex.y*delta.y,
     0.0, 1.0);
}
```

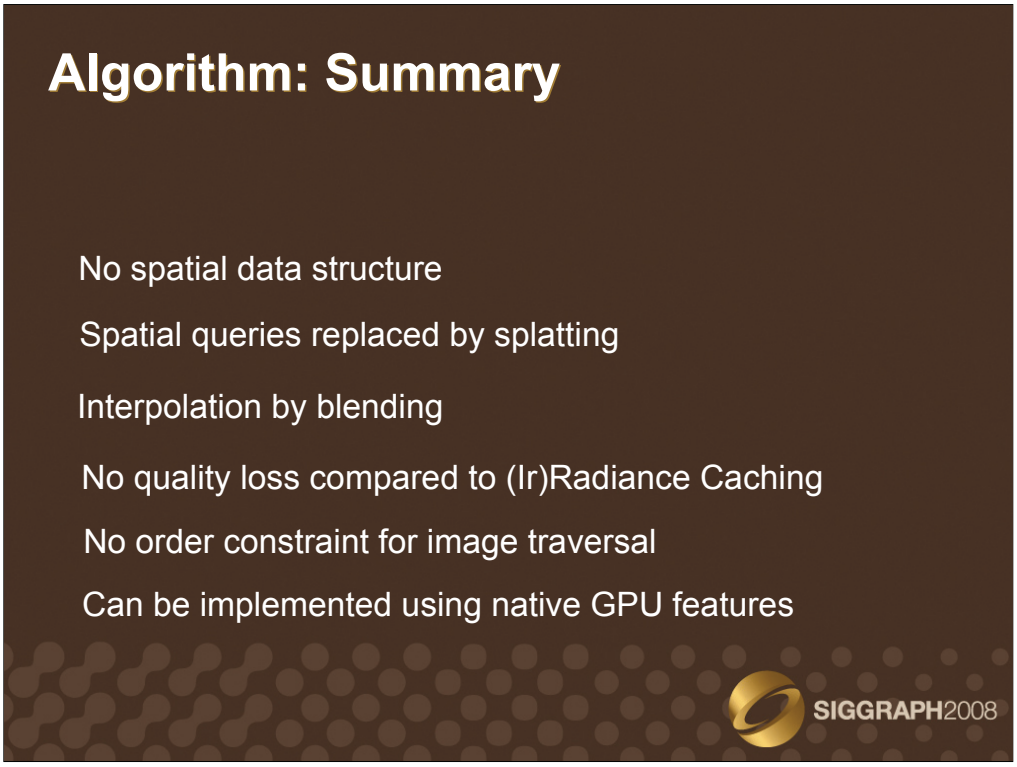


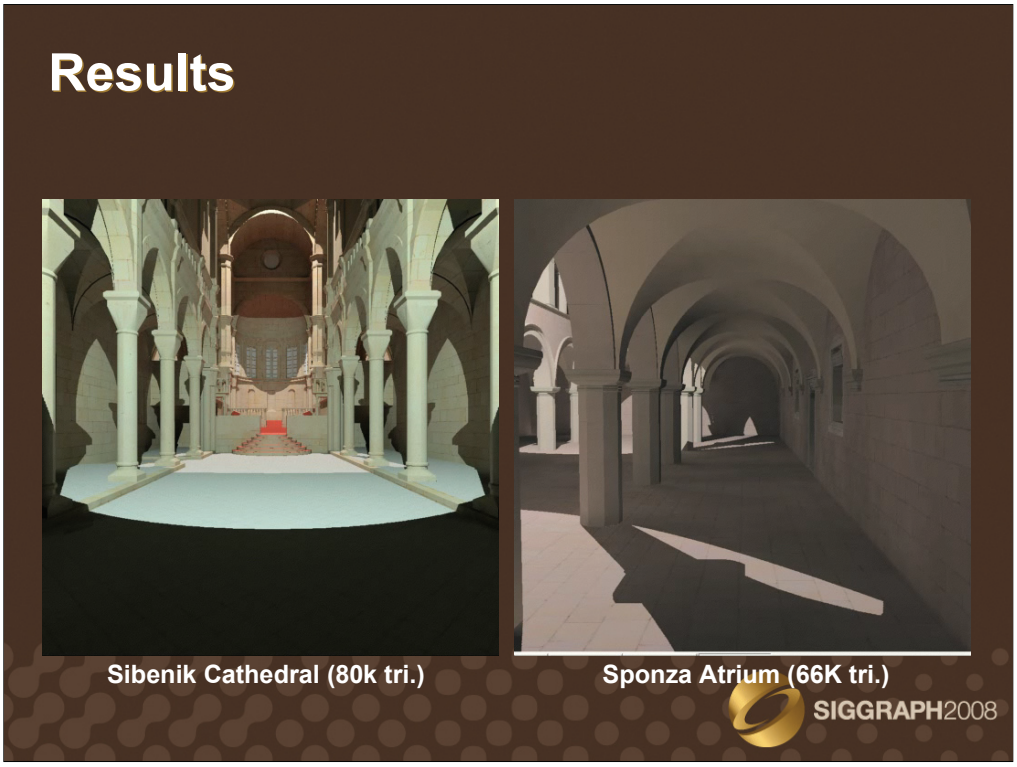
The fragment shader then computes the actual value of the weighting function for each point within the quadrilateral, using the formula defined in [Ward88]. If the record is allowed to contribute to the lighting of a visible point, its contribution is computed using irradiance gradients such as in [Ward92]. The output of the shader for the corresponding fragment is a HDR RGBA value, where RGB represents the weighted contribution of the record, and A represents the weight of the record's contribution.

When several records are rendered, their RGBA values are simply added together using the floating-point alpha blending of graphics hardware (glBlendFunc(GL_ONE, GL_ONE)). This yields a sum of weighted contributions in the RGB components, and a sum of weights in the A component.

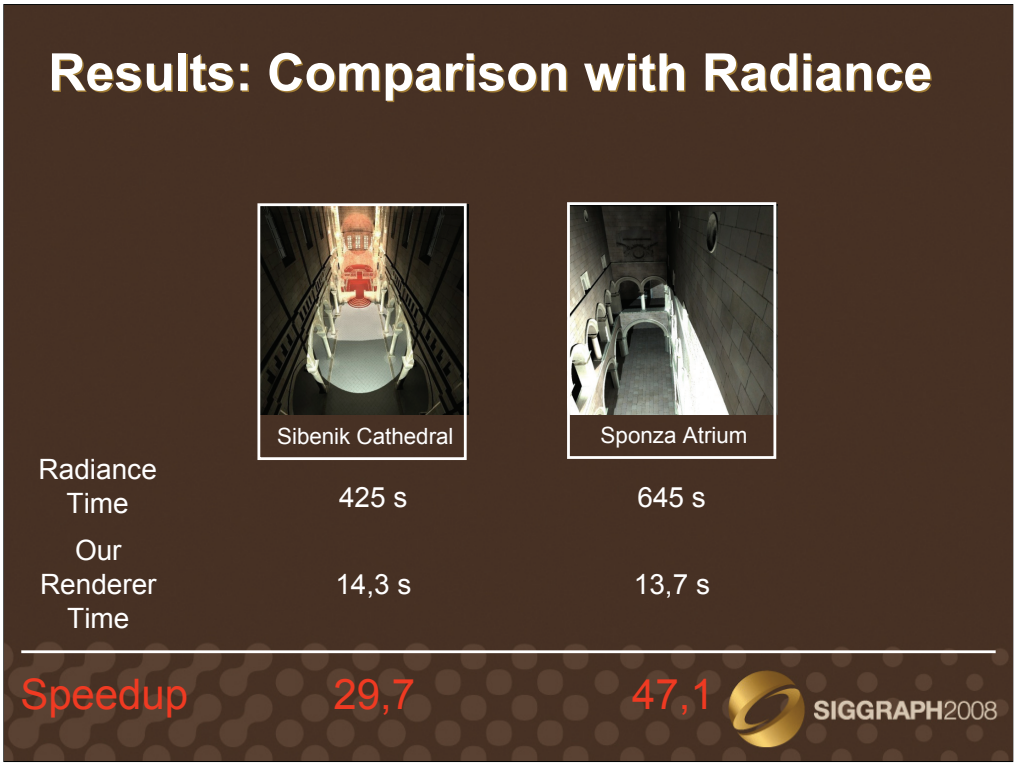


In a final step, a texture containing the RGBA values discussed above is used in a final render pass. This pass renders a single screen-sized quadrilateral. For a given pixel the fragment shader fetches the corresponding RGBA value, and outputs RGB/A to perform the irradiance estimation described in [Ward88].

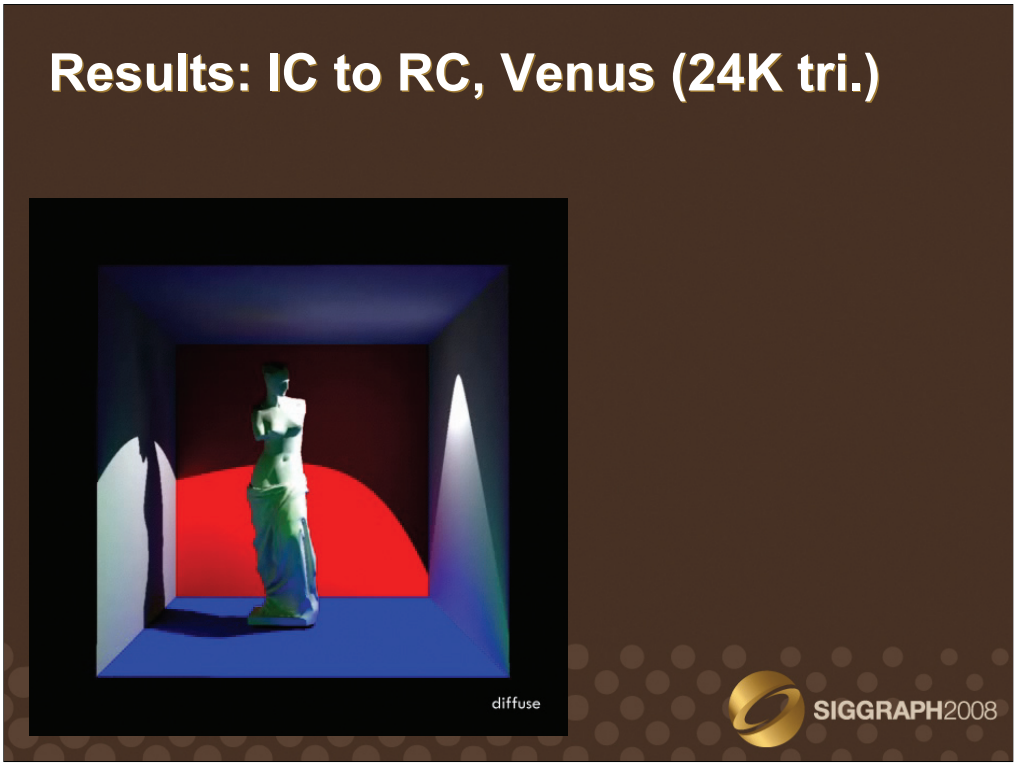

Algorithm: Summary
No spatial data structure
Spatial queries replaced by splatting
Interpolation by blending
No quality loss compared to (Ir)Radiance Caching
No order constraint for image traversal
Can be implemented using native GPU features
SIGGRAPH2008



Those videos were rendered on a GeForce 6800. The scene being static, the irradiance cache is reused across frames. Hence the first frame has been computed in approximately 20s, while the other frames took approximately 1s to render.



A comparison between our GPU-based method and the Radiance Software.



This method can be straightforwardly extended to glossy global illumination using radiance caching.