Speeding up global illumination computations using programmable GPUs

G. Fournier B. Péroche

L.I.R.I.S : Lyon Research Center for Images and Intelligent Information Systems, CNRS / INSA de Lyon / Université Lyon 1 / Université Lyon 2 / Ecole Centrale de Lyon, Bâtiment Nautibus, 8 boulevard Niels Bohr, 69622 Villeurbanne Cedex, FRANCE

Abstract

In the context of realistic image synthesis, many stochastic methods have been proposed to sample direct and indirect radiance. We present new ways to use hardware graphics to sample direct and indirect lighting in a scene. Jittered sampling of light sources can easily be implemented on a fragment program to obtain soft shadow samples. Using a voxel representation of the scene, indirect illumination can be computed using hemispherical jittered sampling. These algorithms have been implemented in our briefly presented multi-mesh caching framework but can be used in other contexts like radiosity or final gathering of the photon map.

Key words: realistic image synthesis, soft shadow, indirect illumination, jittered sampling, hemispherical projection

1 Introduction

Realistic image synthesis requires to simulate the behaviour of light and its interactions with scene objects. The quality of this simulation is important for artworks realism and essential when synthesized images are used in contexts such as architecture, lighting design, car design... A lot of algorithms allow very accurate simulation, but the time required by such methods to produce an image can exceed hours according to the scene complexity and to the desired quality. Unfortunately, in many situations, the user wants real time

Email addresses: `gabriel.fournier@liris.cnrs.fr` (G. Fournier), `bernard.peroche@liris.cnrs.fr` (B. Péroche).

interaction or at least progressive rendering that would allow him to modify the simulation parameters if unsatisfied with the current resulting image. Recent hardware-based methods developed for the video game industry allow visually convincing real time rendering. Unfortunately, the approximations involved in these algorithms are not easily controlled. The sampling techniques exposed in this article use the graphics hardware to speed up accurate illumination computations. Using these techniques within a progressive and adaptive rendering algorithm, called *multi-mesh caching*, we are able to interactively render images that progressively converge toward a high quality solution that includes soft shadows and indirect illumination. Our method provides walk-through interactivity: while the scene is being rendered, the user can move around the scene, but scene objects and light sources are static in our current implementation.

The remainder of this paper is organized as follows. Previous work on global illumination and on the use of graphics hardware is reviewed in section 2. Section 3 succinctly describes the framework in which our work is developed. Our methods to sample direct illumination and indirect illumination are described in Section 4 and 5. Our results are presented in Section 6 and discussed in Section 7.

2 Previous work

Rendering a synthesized image requires two main tasks. First, visible objects at each pixel of the image are determined; then the shading value of each pixel is computed. This second task is by far the most time consuming; it requires to compute the quantity of light (radiance) reflected by visible objects toward the observer. This radiance can be split according to the paths taken by light rays from light sources toward the observer. The direct radiance, that involves only one bounce on an object, is the quicker to compute. It is also the most visually important part of radiance in regularly lit scenes. Indirect radiance, that involves inter-objects reflections, is by far longer to compute. Though indirect radiance is often masked by the direct one, it brings realism to the obtained images by softening direct radiance and by revealing the *atmosphere* of the scene.

Computing the radiance at point x in direction ω_r involves solving the rendering equation (1) introduced by Kajiya [1]

$$L_r(x,\omega_r) = L_e(x,\omega_r) + \int\limits_{\Omega_i} f_r(x,\omega_i,\omega_r)L_i(x,\omega_i)cos\theta_i d\omega_i \qquad (1)$$

where f_r is the BRDF of the object that tells which amount of the incoming radiance L_i from direction ω_i is reflected toward the observer in direction ω_r and L_e is the self emitted radiance of the object. Computing radiance L_r at point x of an object requires to compute radiance L_i on all objects in sight of x. Two main approaches have been proposed to solve this recursive equation: finite element methods and stochastic sampling. Some attempts have been made to speed up these methods using graphics hardware.

2.1 Radiosity

Radiosity is a finite element method introduced by Goral et al. [2] inspired by works on heat transfers. The scene is subdivided in small patches in order to compute light transfers between these patches. First, form factors between patches are computed. A form factor represents the contribution of one patch to another one; it depends on the mutual visibility and on the orientation of the patches. Once form factors are known, a huge linear system, that links all patches together, is solved. To provide high quality images, the number of patches has to be important, increasing the memory and computation time requirements. The provided solution is independent on the viewing point and can be re-used from one frame to another. Through *hierarchical radiosity* [3] reduces this memory and time cost, we preferred to use a stochastic sampling method.

2.2 Stochastic methods

Kajiya [1] proposed the *path tracing* algorithm to solve his rendering equation (1). Path tracing consists of randomly following light paths from the eye to a light source, bouncing from one object to another in random directions. To take into account as many kinds of paths as possible, an enormous amount of paths has to be traced through the scene. Even so, this basic method leads to very noisy images. To speed up the method, many variations have been proposed. The *next event estimator* [4] adds direct radiance to the path each time the light ray bounces on an object; this reduces the variance of the image using a less important number of rays. The *metropolis light transport* method [5] mutates the most important light paths to reduce the variance with as few rays as possible.

To reduce the variance, stratified sampling can be used. When a light path meets an object, a new direction has to be chosen. Instead of randomly picking this direction, an hemisphere is built over the object and split into cells through which rays are being sent. Serpaggi et al. [6] proposed to progressively adapt the hemisphere subdivision to send more rays through non uniform cells.

Another widely used approach is the *photon map* [7] algorithm, which is a two-pass method. First, photons are sent from light sources and randomly traced through the scene to be stored in the photon map (a kD-tree). Then, the rendering pass requires to collect photons from the photon map in the neighbourhood of the point to shade and to estimate the radiance from these collected photons.

2.3 Hardware implementation

Graphics hardware can be used at different places of realistic image synthesis. First, it can be used to compute illumination at some (if not all) points of the scene. Second, it can be used to render these illuminated points. The GPU can be used either as a general processor or for its specific functionalities which are hidden surfaces removal (through the z-buffer algorithm), fast triangle rasterization and fast interpolations through rendered triangles.

Instant radiosity [8] is an interesting method that uses the graphics hardware, for its specific functionalities, to speed up visibility computations. First, a few photons are propagated through the scene and are used during the rendering phase as virtual light sources whose shadows are computed on the graphics hardware using the *shadow map* algorithm [9]. During the rendering pass, the scene is rendered multiple times, each time lit by a different virtual light source. The obtained images are summed in the accumulation buffer to provide the final image. The more photons are used, the higher is the quality of the obtained image. Tests showed that 500 virtual point light sources are enough to obtain fair quality images.

Recent progresses of graphics hardware allowed to port some of the previously mentioned algorithms on GPUs, which are then used as a SIMD coprocessor. Purcell et al. used the GPU to compute ray triangle intersection in a fragment program. This led to the implementation of a ray tracer [10] and of the photon map algorithm [11]. Using the GPU as an SIMD coprocessor relieved the burden on the CPU, but does not make full use of the GPU functionalities. Moreover, sending and retrieving data to or from the GPU requires sometimes complex parallelization of the algorithms.

Larsen and Christensen [12] use the GPU in a more regular way. They implemented the last pass of the the *photon map* algorithm on the GPU. First, photons are traced through the scene by the CPU. Then, the final gathering step uses the GPU to sample the photon map which is rendered from the point to shade.

3 Rendering of cached samples

3.1 Samples caching

Computing direct illumination from area light sources and indirect illumination for each pixel of a picture is very costly and most of the time not necessary as radiance signal is coherent in space, and in time in case of an animation. Many methods, inspired by *irradiance caching* [13], have been proposed to reduce the number of sample points and interpolate irradiance in between. To better adapt illumination sampling over the scene objects, radiance signal should be split up and so that each sub-signal can be sampled according to its frequency. Zaninetti et al. [14] proposed to sample separately direct radiance from each light source, indirect diffuse radiance and caustic radiance.

These computed samples can be stored in space partitions like octrees, grids or 3D-trees. In this case, software interpolations and software rendering have to be used to obtain an image. Another option is to use an image space cache. Pighin et al. [15] built a triangular mesh over the image that is progressively refined and hardware rendered. Simmons and Séquin proposed to build this progressively refined mesh over an hemisphere so that samples can easily be reused from one frame to another. The last solution is to store the samples directly on a mesh built and refined over the geometrical mesh [16].

Each of these caching techniques have there pros and cons. Space partition caches permit any kind of scene description (not only triangles), and they allow an easy control of the sampling density. Their main problem is that they only allow complex software interpolations between a variable number of samples gathered in the neighbourhood of the interpolation point. Image space caches permit hardware interpolations, but they lead to interpolate not only radiance but also geometry over the image. This results in blurry geometrical edges while the mesh is being subdivided. Storing the samples on a progressively refined geometrical mesh guarantees to correctly render the scene geometry with its progressively computed radiance. The main problem of this last technique is that the sampling density cannot easily be controlled. In case of highly tessellated objects, at least one sample of radiance has to be computed per vertex of the geometrical mesh.

3.2 Multi-mesh caching

The method, we proposed in [17] and extended since, tries to keep the best from the previously mentioned caching techniques. We propose to sample separately indirect irradiance and direct irradiance of each light source. These

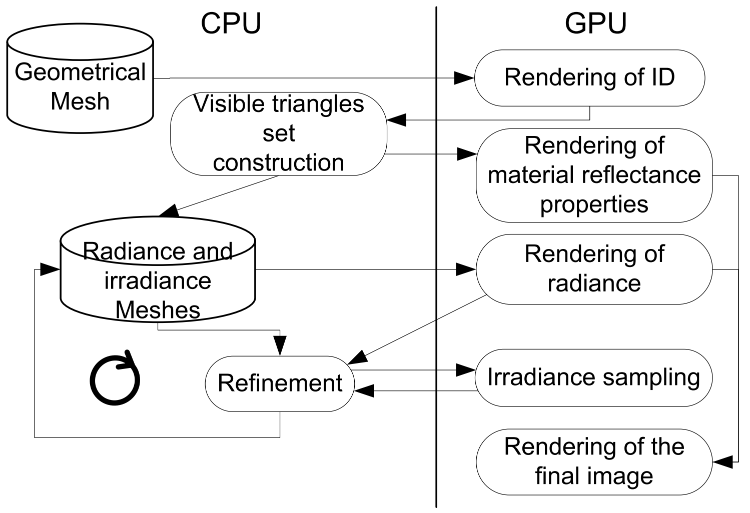
CPU
GPU
Geometrical Mesh
Rendering of ID
Visible triangles set construction
Rendering of material reflectance properties
Radiance and irradiance Meshes
Rendering of radiance
Refinement
Irradiance sampling
Rendering of the final image

Fig. 1. Overview of the multi-mesh caching method

samples are cached in different meshes. We use small grid caches built over highly tessellated objects to keep a reduced sampling density of irradiance over these objects. Irradiance at the vertices of tessellated objects is interpolated using the samples stored in the grid. For the other objects (not highly tessellated), we build and refine the multiple meshes cache over the objects to store the different irradiance samples.

The different meshes are progressively refined until mesh elements are small enough not to miss small radiance discontinuities, and until the user cannot see any discontinuity due to T-vertices. While the meshes are being subdivided, the image is progressively updated through multiple rendering passes (see Figure 1). First, using the geometrical mesh, a colour image, or more precisely a reflectance properties image, is once and for all rendered for each frame. Second, a direct radiance image is rendered using the direct irradiance mesh of each light source. Specular and diffuse radiances are computed on the fly in a fragment program using multiple cached irradiances. Third, an indirect radiance image is rendered using the indirect irradiance mesh. Fourth, to obtain the image displayed to the user, colour and radiance images are combined. This final rendering pass also includes a tone mapping operator to obtain displayable colour values.

Our multi-mesh caching method has some interesting properties compared to caching methods that use a single cache. First, it allows to better tune the sampling density over objects, but it also allows more re-use of already computed radiance and irradiance values. Direct irradiance of each light source can be re-used from one frame to another, even when the observer moves, as it is the only data required to compute the view dependent direct specular radiance. Moreover, direct irradiance can be re-used to compute indirect radiance.

4 Speeding up direct illumination computations

Graphics hardware is designed to quickly solve visibility problems using the z-buffer algorithm. Computing direct irradiance at a given point requires to determine the visibility of each light source from this point. Direct irradiance E_d of a light source ls can be computed as :

$$E_d(x, ls) = \int_{ls} vis(x, x')L(x')cos\theta cos\theta' \frac{dS}{r^2} \tag{2}$$

where $L(x')$ is the emittance of element x' of surface dS of the light source, θ is the angle between the object normal and light-eye direction, θ' is the angle between the light source normal and the light-eye direction, $vis(x, x')$ is either 1 or 0 depending on the visibility of point x' from point x. Traditionally, ray tracing was used to estimate the visibility, but graphics hardware offers interesting possibilities. According to the desired quality, different algorithms are possible from very fast ones, that offer coarse approximations, to slower but more accurate ones.

All the algorithms presented in the next subsections lay on the same basis. The graphics hardware is used in a standard way. The camera is placed on the point to shade and aims at the light source. Actually, a small frustum that fully includes the light source is computed to parametrize the viewing transformation and perspective projection. The light source and all the potential occluders that are in the viewing frustum are then rendered. Each pixel of the obtained image, where the light source projects itself without being occluded, brings some energy to the point to shade. Summing the energy brought by each pixel, we obtain the direct irradiance from the light source on the point to shade.

4.1 Coarse approximations

4.1.1 Fast visibility estimation

The fastest way to count the number of pixels of the obtained image where the light source projects itself without being occluded makes use of the occlusion query functionality of recent graphics hardware. The potential occluders are rendered first to set up the z-buffer, then an occlusion query is used while rendering the light source. The occlusion query will contain the number of pixels where the light source is not masked. Assuming that each pixel carries the same energy to the point to shade, we easily obtain the direct illumination value. The main problem is that this assumption is false. Giving each pixel the same weight does not take into account the fact that each pixel is not seen under the same solid angle from the point to shade. Moreover, this simplification does not allow to take the $cos\theta$ term from equation (2) into account. This method, despite being fast, can only give a coarse approximation of the real direct irradiance.

4.1.2 Regular sampling

To obtain more accurate values, we have to weight each pixel according to the solid angle it sustends and to the $cos\theta$ value computed at the pixel. Occlusion queries are useless to take into account these considerations, as they do not allow to distinguish between the pixels. The rendered image has to be read back to the CPU where pixels can be weighted according to their position, before being summed. Reading back an image can be quite costly. To avoid wasting bandwidth, the rendered image has to contain only the fewest bits per pixel as possible; only 1 bit per pixel is really required to distinguish light source pixels from other pixels (occluders and background). Each pixel solid angle can be estimated on the CPU as

$$\omega = \frac{(\overrightarrow{xx'} \;.\; \overrightarrow{N_L})(\overrightarrow{xx'} \;.\; \overrightarrow{N_S})dS}{(\overrightarrow{xx'} \;.\; \overrightarrow{xx'})^2} \tag{3}$$

where $\overrightarrow{xx'}$ is the vector from the eye to the pixel, $\overrightarrow{N_L}$ is the normal to the image plan, $\overrightarrow{N_S}$ is the normal to the object at the point to shade and dS is the size of a pixel of the image.

The main problem with this method is that it uses a regular grid (the computed image) to sample the light source. To avoid bias and visible structure in the computed irradiance, some randomness should be introduced in the light source sampling. The regular sampling process has also the disadvantage of not allowing progressive sampling. Once the light source has been sampled, resampling it won't increase the obtained irradiance quality, as the same result will be obtained again and again.

4.2 Hardware jittered sampling

4.2.1 Custom rasterization

Jittered sampling is unbiased and allows progressive sampling. Instead of regularly sampling the light source, each sample is randomly displaced a little. Although current graphics hardware is programmable, the rasterization pro-

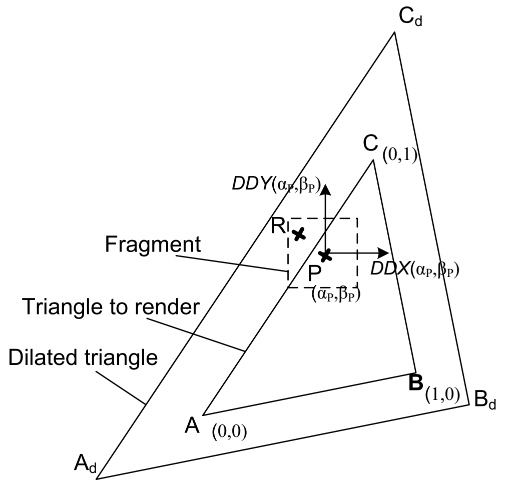

Fig. 2. Jittered rendering

cess cannot currently be modified. However, a fragment program can be used to introduce some randomness in the rasterization process. While rendering a polygon, the regular rasterization process tests each pixel for membership to the polygon at the pixel centre. If the pixel belongs to the polygon, then a fragment is generated and sent to the fragment program. Our method consists in testing each fragment for membership to the polygon at a random position in the pixel. If this random point does not belong to the fragment, then the fragment is killed and not rendered. Using this technique, edges of polygons are modified, leading some fragments to be deleted. Correct jittered sampling process would lead to create some fragments that were not created by the regular rasterization process. As fragment programs cannot create new fragments, we dilate the rendered polygons. This way, more fragments than needed are sent to the fragment program that tests their membership to the original polygon.

The position of point P in a plan containing a triangle $[A, B, C]$ can be expressed as

$$P = A + \alpha \overrightarrow{AB} + \beta \overrightarrow{AC} \tag{4}$$

P belongs to triangle $[A, B, C]$ if and only if $0 \leq \alpha$, $0 \leq \beta$ and $\alpha + \beta \leq 1$.

To perform a jittered rendering of triangle $[A, B, C]$, we first dilate it and compute triangle $[A_d, B_d, C_d]$. We also compute coordinates α and β of A_d, B_d and C_d in the $(A, \overrightarrow{AB}, \overrightarrow{AC})$ basis. Triangle $[A_d, B_d, C_d]$ is rendered, providing these coordinates with each vertex through texture coordinates parameters. For each generated fragment P, the fragment program will receive as inputs interpolated coordinates (α_P, β_P) of P in the $(A, \overrightarrow{AB}, \overrightarrow{AC})$ basis. Coordinates (α_P, β_P) are interpolated at the pixel centre of each fragment. We want to test a random point R of the pixel for membership to triangle $[A, B, C]$. Coordinates (α_R, β_R) of point R have to be evaluated. This can be done using partial derivatives of the coordinates provided by the DDX and DDY functions included in *GLSL* or in the *fragment_program_NV* extension.

$$(\alpha_R, \beta_R) = (\alpha_P, \beta_P) + uDDX(\alpha_P, \beta_P) + vDDY(\alpha_P, \beta_P) \tag{5}$$

where u and v are random floats taken in interval $[-0.5, 0.5]$. If coordinates (α_R, β_R) of point R satisfy $0 \leq \alpha_R$, $0 \leq \beta_R$ and $\alpha_R + \beta_R \leq 1$, then R belongs to triangle $[A, B, C]$.

This algorithm has a minor flaw. To make sure that more fragments than required are sent to the fragment program, we dilate the triangle before rendering them. Using a fixed percentage dilatation does not guarantee that the rendered triangle will be at least one pixel wider on all sides for all viewing points and directions. Using a variable dilatation would solve this problem but would prevent us from using display lists. Another solution we are exploring is to dilate triangles in the vertex program.

4.2.2 Multi-sampling

Actually, the fragment program can test a same fragment multiple times for triangle membership at different random positions of the fragment. The membership can be encoded per bit in the fragment colour result. Testing multiple times a fragment has an impact on the fragment program speed but it makes each read back of an image more interesting as more (if not all) bits read back can contain meaningful information. Testing multiple times a same fragment makes it impossible to kill a fragment that would only partially belong to the rendered triangle. This problem can be solved using logical operations on the frame buffer.

The goal of our algorithm is to obtain an image where each bit of each pixel colour is 1 if the light source is not occluded and 0 if the light source is occluded or if it does not cover the part of the pixel represented by the bit. Each bit of a pixel colour contains a sample randomly taken in a portion of the surface of the pixel that is regularly subdivided. We do not want to switch the fragment program when rendering light source instead of occluders. We, arbitrarily, decided that the fragment program would always output 1 for each bit that represents a part of the pixel that does not belong to the rendered triangle, and 0 when the part belongs to the triangle. First, we render the light source, setting the frame buffer logical operation to NOT; then, we render the potential occluders, setting the logical operation to AND. When rendering the occluders, we use depth testing but depth writing is disabled. This way, partially visible fragments of occluders do not prevent other occluders to fully

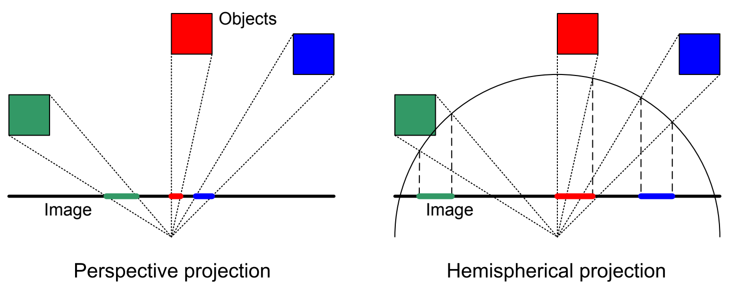

Fig. 3. Differences between perspective (left) and hemispherical (right) projection

mask the light source for a given pixel. Depth testing, using the depth buffer initialized with the light source depths, is nevertheless required to insure that occluders are in front of the light source.

Writing a fragment program that outputs bit fields instead of floating values was not straightforward on our hardware. Fragment programs can only manipulate floating values and do not currently provide per bit operators like the & or | operators of the C language. GLSL specifications let us think that these operators will be added in the future. It is nevertheless possible for a fragment program to output bit fields using their float value representation.

5 Speeding up indirect illumination computation

5.1 Iterative indirect illumination computations

As explained in Section 2.2, many stochastic methods can be used to evaluate indirect illumination. We propose to use an iterative method: first, we compute indirect irradiance that includes only one-bounce paths. This one-bounce radiance is stored in a mesh that is then used to compute two-bounce indirect radiance. Progressively, we add indirect irradiance with more bounces. Depending on the objects albedo, a variable number of bounces will be required before the remaining missing energy becomes negligible.

5.2 Hemispherical sampling

To compute (n)-bounce irradiance $E_n(x)$ at point x, $(n-1)$-bounce radiance L_{n-1} that reaches point x is used.

$$E_n(x) = \int_{\Omega} L_{n-1}(x, \omega) d\omega \tag{6}$$

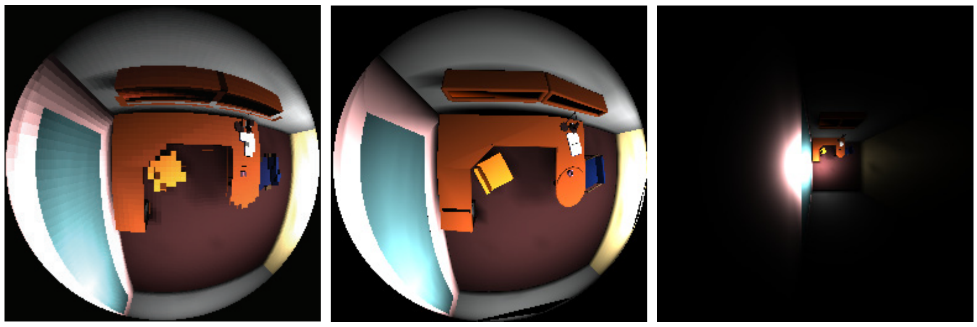

Fig. 4. Voxelised (left) and standard (center) sampling using hemispherical projection and regular sampling using perspective projection (right) (pixels are weighted according to their contribution).

Software hemispherical sampling is traditionally used to compute this integral. Hardware can also be used to speed up the computation. The standard hardware way was either to render the scene on an hemicube, or on a single plan in front of point x, and to sum the obtained images to a single radiance value. The first way required to render the scene at least 5 times (once for each side of the hemicube), while the second neglected radiance coming from grazing angles. Sampling incoming radiance using a single plan with Opengl perspective projection is not the best way to distribute the samples on the rendering plan. Perspective projection gives more importance to peripheral samples (see Figure 3). A better way to distribute samples is to use a fish eye projection that naturally gives each pixel the same importance.

Using a vertex program, approximate hemispherical rendering can be performed on current hardware. To project a point onto the image using the fish eye projection, its x and y coordinates in the eye basis just have to be normalized to obtain its coordinates in the screen basis. The depth coordinate is the norm of the eye to point vector. The only problem is that Opengl uses the z and w coordinates computed by the vertex program to clip points that are outside of the viewing frustum. Using a fish eye projection, such points do not exist. To avoid point clipping, the vertex program must always output 1 for z and w coordinates; the real depth of the point is sent through another parameter to the fragment program that writes it back in its regular register to benefit from the z-buffer algorithm.

5.3 Voxelisation

To obtain an indirect radiance value using hemispherical rendering, an important part of the scene has to be rendered as any object in front of the point to shade can contribute to its indirect illumination. Rendering all these objects is very costly. A coarser representation of the scene can be used without introducing too much change in the obtained indirect radiance. Mesh simplification

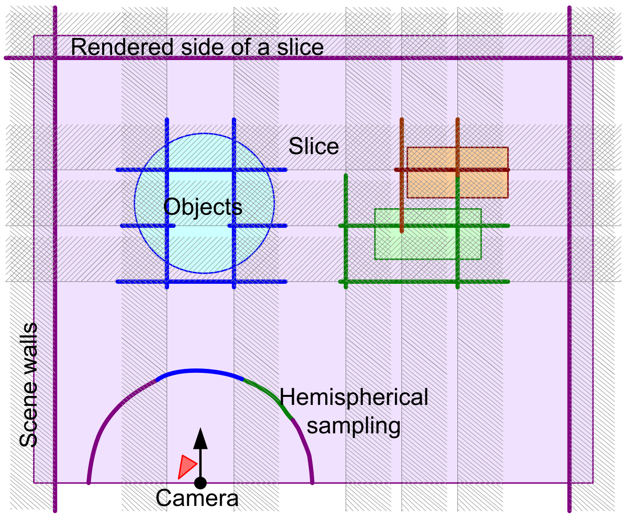

Fig. 5. Slices rendered to compute an indirect illumination sample

algorithms could be used, but an easiest way is to use a voxel representation of the scene. This way, the time spent to render the scene is only dependent on the desired quality that implies a given number of voxels.

Instead of storing a single radiance value per voxel, we consider voxels as little cubes and compute a value for each side of the cube. To compute, store and use these values, we work with slices of space. Each slice has two faces that are two parallel plans separated by the width of the voxels. Slices are built in three orthogonal directions so that the intersection of 3 orthogonal slices creates a voxel. Voxels' radiance values are stored in the texture associated with each slices' face. To fill the texture of a slice's face, the part of the scene inside the slice is rendered into the texture using a parallel projection. Each face of a slice is filled using opposite parallel projections. When a slice's face contains only empty values, the texture is freed and the slice's face won't be rendered to save up memory and time.

Instead of rendering the objects composing the scene, we render the scene's slices. Actually, we only render front facing faces of the slices that are in front of the point to shade (see Figure 5). As this point is, itself, inside a voxel, at the intersection of 3 orthogonal slices whose faces are not rendered (because they are back facing), some close objects, like the close to the eye triangle in Figure 5, are not taken into account. To correct the obtained image, we render the objects that lie inside this voxel. Actually, we expand a little the volume containing close objects that may greatly influence the computed radiance, because the voxel representation may not be precise enough in the close to the point to shade area.

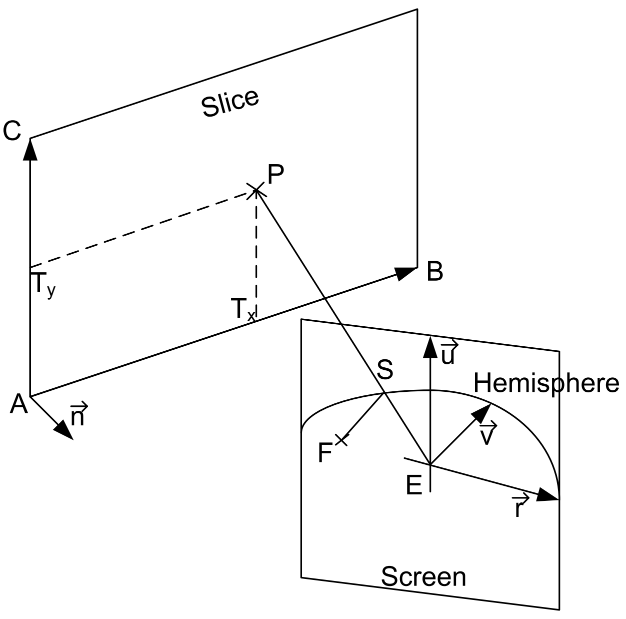

Fig. 6. From fragment F to texture coordinates T

5.4 Slices hemispherical projection

The main problem of hemispherical projection is that it requires important tessellation of the rendered objects so that they appear correctly curved. Tessellating the slices leads to so many polygons that a sliced scene representation does not provide an important speed up of the method compared to a regular scene representation.

To render the slice, we propose an unusual rendering method. Using a fragment program, we find for each pixel of the rendered image the point on each slice and the corresponding texel that falls on the pixel. This is done by rendering for each slice, a quadrilateral, textured with the slice's texture, that covers the full screen. The fragment program that performs the backward projection is built in 3 steps. First, point S on the hemisphere that falls over fragment F, is computed (see Figure 6). Screen coordinates of point F can be randomly perturbed to perform jittered sampling instead of grid sampling. To compute S, the required parameters given to the fragment program are the viewing basis components: eye position E, right direction $\vec{r}$, up direction $\vec{u}$ and viewing direction $\vec{v}$. S is computed as

$$S = E + F_x.\vec{r} + F_y.\vec{u} + \lambda\vec{v} \tag{7}$$

where $\lambda = \sqrt{1 - F_x^2 - F_y^2}$ is looked up in a texture that easily allows us to kill fragments where $F_x^2 - F_y^2 > 1$, that cannot be obtained with an hemispherical projection.

Intersection P of line ES with the slice can be computed from S. The slice is defined through a point A and a normal $\vec{n}$ that are provided to the fragment program as parameters.

$$P = E + \kappa \vec{ES} \tag{8}$$

where

$$\kappa = \frac{-\overrightarrow{AE}\ .\vec{n}}{\overrightarrow{ES}\ .\vec{n}} \tag{9}$$

To finish, we have to find coordinates T of point P in the slices texture orthogonal basis $(A, \vec{AB}, \vec{AC})$.

$$T_x = \frac{\overrightarrow{AT}\ .\ \overrightarrow{AB}}{\left\|\overrightarrow{AB}\right\|^2}, T_y = \frac{\overrightarrow{AT}\ .\ \overrightarrow{AC}}{\left\|\overrightarrow{AC}\right\|^2} \tag{10}$$

Once the texture coordinates have been computed, we fetch the colour in the slice's texture and update the image.

6 Results

6.1 Direct illumination sampling

To test the accuracy of our hardware jittered sampler, we set up a bench test independently of our rendering software. We set up a single light source partially occluded by a single quadrilateral occluder. Instead of computing the radiance of the light source, we only computed the visible solid angle. This way, we could compare the sampled values with analytically computed solid angles. To obtain a good idea of the error induced by each sampling scheme, we computed the error between analytically computed and sampled solid angle over 10000 points spread over a plan. Regular sampling introduces more error when the sampled signal has discontinuities aligned with the sampling grid. This situation, that leads to visual artifacts (see Figure 10), sometimes occurs while rendering a real scene. For the purpose of our test, we deliberately choose a light source that is partly aligned (at least one edge) with the sampling grid. In real rendering situation, care can easily be taken while choosing a viewing frustum not to align it with the light source, but it is impossible to avoid alignment between the light source and all potential occluders. Figure 7 shows the average, standard deviation and maximum error over the 10000 test points for each sampling scheme: regular sampling, jittered with 1 sample per pixel, 4 samples per pixel and 8 samples per pixel.

The obtained results are mostly the expected ones. Jittered sampling reduces the average error; the maximum error and the variance are also reduced. Multisampling has a small impact on the average error, but it reduces greatly the maximum error and the variance. Computing 4 samples per pixel almost divides the maximum error by 2, computing 8 samples per pixel reduces a little more this maximum error. Nevertheless, an unexpected result is obtained. By construction, using a 32x32 sampling image with 4 samples per pixel (each sample randomly taken in a quarter of the pixel) should give the same results than using a 64x64 sampling image with 1 jittered sample per pixel. We observed that using a bigger image with only one sample per pixel gave better results. We may think of one explanation that we have not been able to validate. During our custom rasterization, we use partial derivatives to compute whether a point is in or out of the rendered triangle; these partial derivatives are only approximations discretely computed, whose quality is probably better when evaluated over smaller pixels (or a bigger image).

The spatial distribution of the error is also important. Figure 8 shows this distribution on a plan when light source sampling is done using a 48x48 image (results with other image sizes are quite similar). Sampling light sources using a regular grid introduces a clearly visible structure in the obtained error distribution, while this structure almost disappears when using a jittered

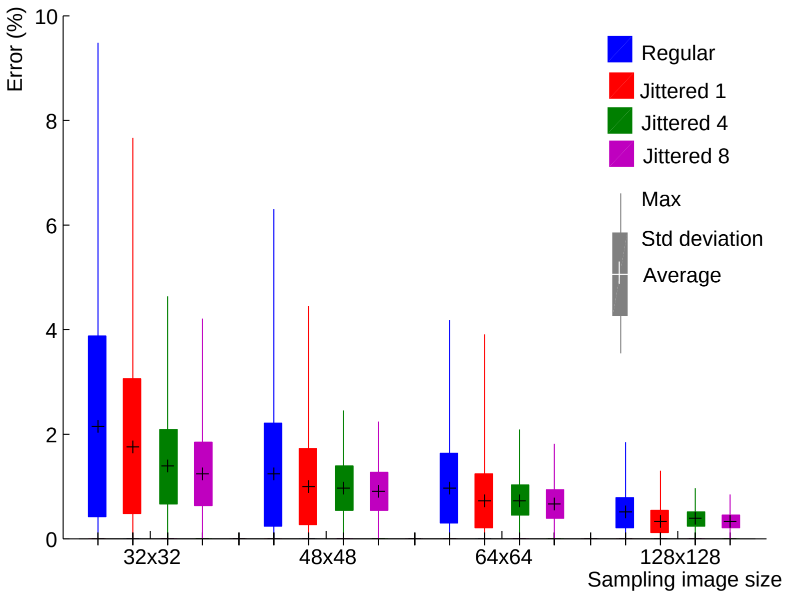

Fig. 7. Average, standard deviation, and maximum error of the estimation of the solid angle sustended by the light source

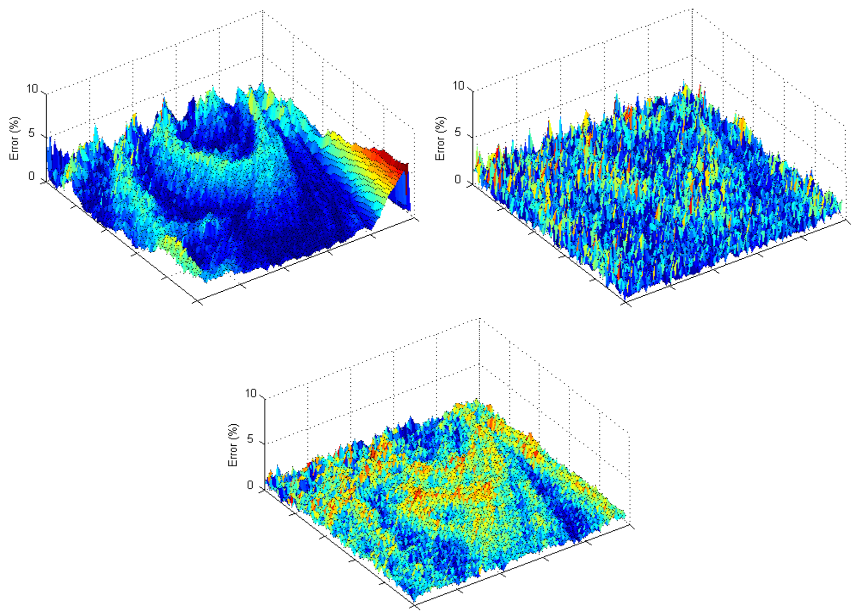

Fig. 8. Distribution of the error over the sampled surface when the solid angle is estimated using a regular grid (left), a jittered grid with 1 sample per pixel (right), or 4 samples per pixel (center)

grid. We obtain another unexpected result when using 4 samples per pixel: the structure in the error is again visible.

To choose between these sampling schemes, sampling speed has to be taken into account. Speed has been evaluated in a real rendering situation (the office scene, see Figure 12) to take into account the non negligible time required to render potential occluders. As our rendering software is multithreaded, it is impossible to be sure that no other threads are running while timing a function. This leads to not really representative average times. Instead of average time, Figure 9 shows median time (computed with 25000 samples spread over the whole scene) for each phase of the sampling process (rendering of the occluders, reading back of the image and weighting of each pixel) for each sampling scheme and 3 sampling image sizes.

Our tests were performed on a Nvidia QuadroFX 3000. The results show that the costliest part of the sampling process is the rendering of occluders. When using the regular sampling scheme, the rendering phase cost is practically independent on the image size. When using jittered sampling, increasing the image size slows down the rendering. Computing more than one sample per pixel slows down further more the rendering process. Reading back the image seems to be in $O(\sqrt{n})$, where n is the number of read back pixels (within the image sizes tested). Weighting the pixels is in $O(n)$, where n is the number of pixels. According to our tests on our hardware, a 48x48 image with 4 jittered samples per pixel or a 64x64 image with a single jittered sample per pixel, seems to be the best compromise between speed and quality.

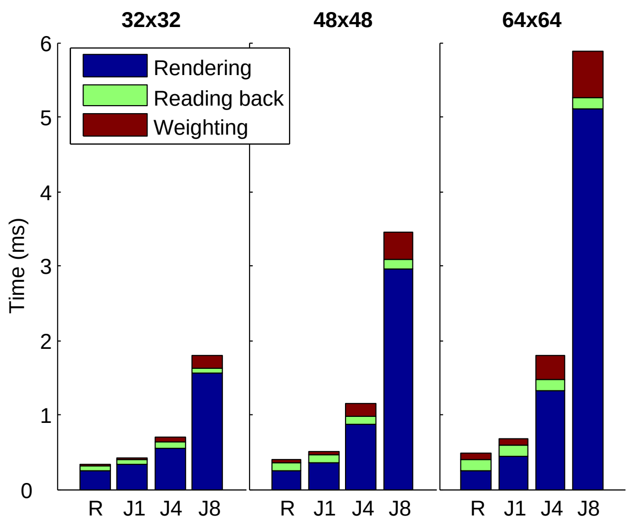

Fig. 9. Timings of each phase of the sampling process per sampling scheme and sampling image size

6.2 Indirect illumination sampling

We tested our indirect illumination technique on the office scene that contains 7000 triangles. The hemispherical projection allows to reduce the size of the image used to sample indirect illumination. Using a perspective projection with a field of view of 160° required images of 256x256 pixels to get 45x45 pixels in the central 90° of the field of view. Using an hemispherical projection, only a 64x64 pixels image is required. The rendered image size can be reduced by a factor 4 without loosing quality. The smaller the image is, the fastest it will be read back and summed. If sampling direct illumination required only 8 bits per pixel (only 1 bit is in fact required for visibility determination), 128 bits per pixel are required to sample indirect illumination which is composed of 3 unbounded floating values (96 bits are actually required). Reading back the 64x64x8 bits images used for direct illumination sampling takes about 150μs, reading back 128x128x128 bits images takes 26ms. This read back cost is far from being negligible. Reducing the image size to 40x40 with 128 bits per pixel gives 2ms read back. We choose to use 40x40 images that give visually satisfying results (for our test scene); further reducing the sampling image size leads to visible artifacts. More tests would be needed to determine precisely the best sampling image size for a given scene.

The main problem of hardware sampled indirect illumination is that it requires to render an important part of the scene. Rendering all these objects can be the most time consuming phase of indirect illumination sampling according to the scene complexity. In our quite simple scene, rendering the objects with a perspective projection has an average cost of 20ms. Using an hemispherical projection requires to tessellate the rendered objects, leading to an average rendering cost of $42ms$. Rendering our voxelised representation of the scene with our slice hemispherical projection algorithm has an average cost of $8ms$. Unfortunately, the voxelised representation is too imprecise in the close to the eye area where objects have to be tessellated and rendered separately. Our current implementation, which uses a single grid to partition space, is not really optimized to limit the number of close to the eye triangles that are tessellated and rendered.The $8ms$ they require to be rendered could probably be reduced with better space partitions like octrees or multi-grids. To sum up, on our test scene, our algorithm requires $18ms$ to sample indirect radiance with a 40x40 pixels image, when a classical approach, with a perspective projection that requires a 128x128 pixels image, needs $46ms$.

Using a triangulated representation of the scene when sampling indirect illumination makes the time required to compute a sample dependent on the mesh complexity. Voxelising the scene allows to easily reduce the scene without using any algorithmically complex mesh simplification algorithm. Using a voxelised representation of the scene allows the user to choose a compromise between speed, memory requirements and quality. The more voxels used, the better the indirect sample quality will be. The memory required is rather

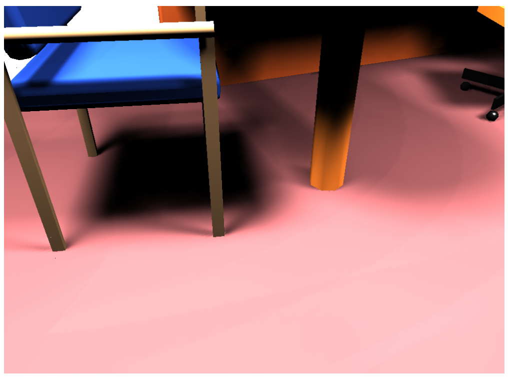
Fig. 10. Visible artifacts on direct radiance regularly sampled with a 48x48 pixels image

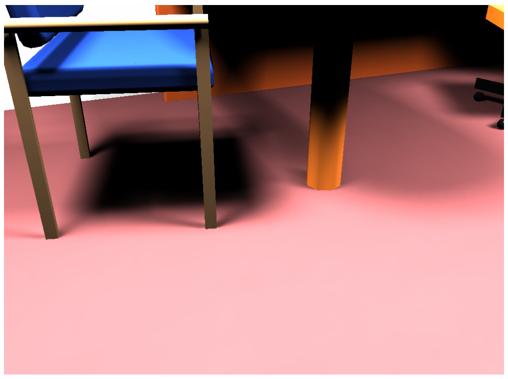
Fig. 11. Direct radiance sampled with our jittered sampling method with a 48x48 pixels image

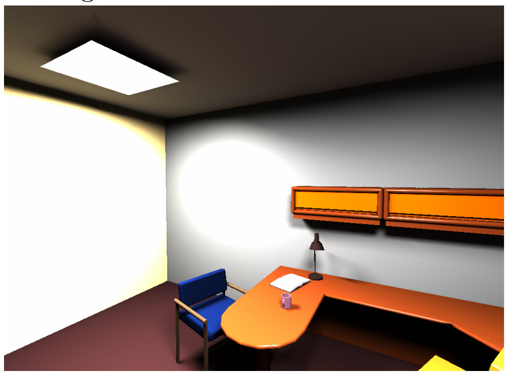
Fig. 12. Office scene rendered with direct and 1-bounce indirect illumination

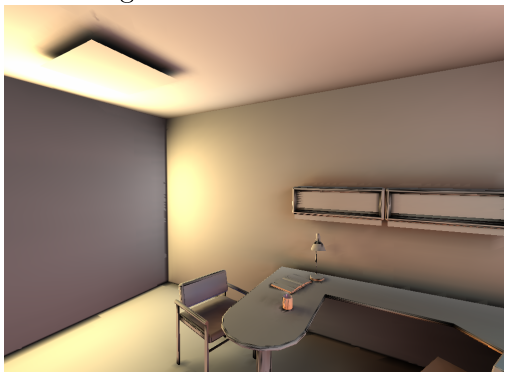
Fig. 13. 1-bounce indirect irradiance

important. Using 64 slices of 64x64 pixels of 128 bits per face in each direction requires 25 MB of memory. It must be remembered that one voxelisation representation of the scene must be stored for each bounce of indirect lighting taken into account. Creating a voxelised representation with the graphics hardware requires 2.9s on our test scene. This creation time is dependent on the mesh complexity and on the number of slice desired.

7 Discussion

Using the graphics hardware to sample direct radiance of area light sources offers an interesting alternative to ray tracing. Our method, that allows jittered hardware sampling, increases the quality of the obtained samples and reduces their variance. Jittered hardware sampling also allows progressive sampling. If unsatisfied with the current result, a light source can be sampled multiple times to increase the sample quality. Computing more than one sample per pixel increases the length of the fragment shader used, reducing the sampling speed. Our graphics hardware is not the most recent one, the new generation of GPUs allows longer fragment programs and is certainly faster.

Sampling indirect lighting is far more costly than sampling the direct one. Our hardware sampling method that uses an hemispherical projection of voxels is faster than sampling an hemisphere with software traced rays. Our method can be used in other contexts than our multi-mesh caching method. For instance, it could be used to perform the final gathering pass in a photon map algorithm like the one presented in [12]. The main problem of our hardware voxelisation of the scene is that it samples the scene regularly and not very densely, producing an aliased representation of the scene. Better voxelisation might be obtained by creating the scene slices' textures at a higher resolution than the one used to store them. The simple grid might be replaced by more adaptive structures like octrees or multi-grids, in order to take the local complexity of the scene into account.

8 Conclusion

Graphics hardware is still evolving but it already offers interesting features to replace CPUs for many algorithms. In this paper, we tried to use the current programmability of the GPU to implement a not standard rasterization and a not standard projection that could be useful in many image synthesis algorithms. Maybe up-coming GPUs will propose such features in standard.

References

[1] J. T. Kajiya, The rendering equation, in: Computer Graphics (Proceedings of SIGGRAPH 86), Vol. 20, 1986, pp. 143–150.

[2] C. M. Goral, K. E. Torrance, D. P. Greenberg, B. Battaile, Modelling the interaction of light between diffuse surfaces, in: Computer Graphics (Proceedings of SIGGRAPH 84), Vol. 18, 1984, pp. 213–222.

[3] P. Hanrahan, D. Salzman, L. Aupperle, A rapid hierarchical radiosity algorithm, in: Computer Graphics (Proceedings of SIGGRAPH 91), Vol. 25, 1991, pp. 197–206.

[4] S. N. Pattanaik, S. P. Mudur, The potential equation and importance in illumination computations, Computer Graphics Forum 12 (2) (1993) 131–136.

[5] E. Veach, L. J. Guibas, Metropolis light transport, in: Proceedings of SIGGRAPH 97, Computer Graphics Proceedings, Annual Conference Series, 1997, pp. 65–76.

[6] X. Serpaggi, B. Péroche, An adaptive method for indirect illumination using light vectors, Computer Graphics Forum 20 (3) (2001) 278–287.

[7] H. W. Jensen, Global illumination using photon maps, in: Eurographics Rendering Workshop 1996, 1996, pp. 21–30.

[8] A. Keller, Instant radiosity, in: Proceedings of SIGGRAPH 97, Computer Graphics Proceedings, Annual Conference Series, 1997, pp. 49–56.

[9] M. Segal, C. Korobkin, R. van Widenfelt, J. Foran, P. E. Haeberli, Fast shadows and lighting effects using texture mapping, in: Computer Graphics (Proceedings of SIGGRAPH 92), Vol. 26, 1992, pp. 249–252.

[10] T. J. Purcell, I. Buck, W. R. Mark, P. Hanrahan, Ray tracing on programmable graphics hardware, ACM Transactions on Graphics 21 (3) (2002) 703–712.

[11] T. J. Purcell, C. Donner, M. Cammarano, H. W. Jensen, P. Hanrahan, Photon mapping on programmable graphics hardware, in: Graphics Hardware 2003, 2003, pp. 41–50.

[12] B. D. Larsen, N. J. Christensen, Simulating photon mapping for real-time applications, in: Rendering Techniques 2004: 15th Eurographics Workshop on Rendering, 2004, pp. 123–132.

[13] G. J. Ward, F. M. Rubinstein, R. D. Clear, A ray tracing solution for diffuse interreflection, in: Computer Graphics (Proceedings of SIGGRAPH 88), Vol. 22, 1988, pp. 85–92.

[14] J. Zaninetti, X. Serpaggi, B. Péroche, A vector approach for global illumination in ray tracing, Computer Graphics Forum 17 (3) (1998) 149–158.

[15] F. P. Pighin, D. Lischinski, D. H. Salesin, Progressive previewing of ray-traced images using image plane discontinuity meshing, in: Eurographics Rendering Workshop 1997, 1997, pp. 115–126.

[16] P. Tole, F. Pellacini, B. Walter, D. P. Greenberg, Interactive global illumination in dynamic scenes, ACM Transactions on Graphics 21 (3) (2002) 537–546.

URL http://www.graphics.cornell.edu/pubs/2002/TPWG02.html

[17] G. Fournier, B. Péroche, Multi-mesh caching and hardware sampling for progressive and interactive rendering, in: Proceedings of WSCG 2005, Plzen, Czech Republic, Union Agency, 2005, pp. 63–70.