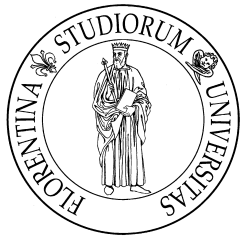
UNIVERSITÀ DEGLI STUDI DI FIRENZE
Facoltà di Scienze Matematiche, Fisiche e Naturali
Corso di Laurea in Informatica



Thesis in Computer Science

PHOTON MAPPING REVISITED

Candidate:
FRANCESCO SARRACCO

Supervisor:
Vincenzo Ancona

Academic Year 2008-2009

1 INTRODUCTION

Computer graphics is concerned with the generation and manipulation of images through the computer. Within this subject beginning from the academic year 2005/2006 in the class of computational geometry[1], it's been developed an engine for rendering images (basically a ray tracer). Based on that it's been developed a more complex software to produce photorealistic images called aRt (another ray tracer).; this software it's been fully developed by computation geometry students from year to year. The algorithm that we are going to introduce here above (photon mapping revisited), developed under the supervision of Alberto Mancini, add to aRt an important element contributing then to the elaboration of this bigger project.

1.1 PHOTOREALISTIC GRAPHICS

The photorealistic graphics is a branch of computer graphics that concerns the generation of images beginning from a given set of data. Usually, we start from a full description of a three-dimensional scene stating:

- dimensions;
- objects collocation;
- properties of the materials that the objects in it are made of;
- position of the light sources in the scene;
- emission properties of every light.

Merging and processing all these information, we can generate a new image with a point of view based on a virtual camera inside the scene. The main idea is to generate images as much photorealistic as possible, up to the stage that it's hard to see the difference between our images and a picture of the same scene in the reality. In order to do that we need to have accurate physical properties for materials and lights behavior, and we need to have them well modelized and simulated. Only a good knowledge of what we are trying to simulate will bring us to simplify the simulation and to know how this will affect the resulting

1 Class of the degree in Computer Science, taught by Prof. Vincenzo Ancona in collaboration with Alberto Mancini, at the Faculty of Mathematical, Physical and Natural Sciences of the University of Florence.

image.

1.2 GOALS

The reason for implementing this algorithm is the lack, in aRt, of representing the inter-reflection effect, that, as we will see, is a limit of the ray tracer itself. Especially, we are going to focus on the caustic effect. In optics, a caustic or caustic network is the envelope of light rays reflected or refracted by a curved surface or object, or the projection of that envelope of rays on another surface. The caustic is a curve or surface to which each of the light rays is tangent, defining a boundary of an envelope of rays as a curve of concentrated light.

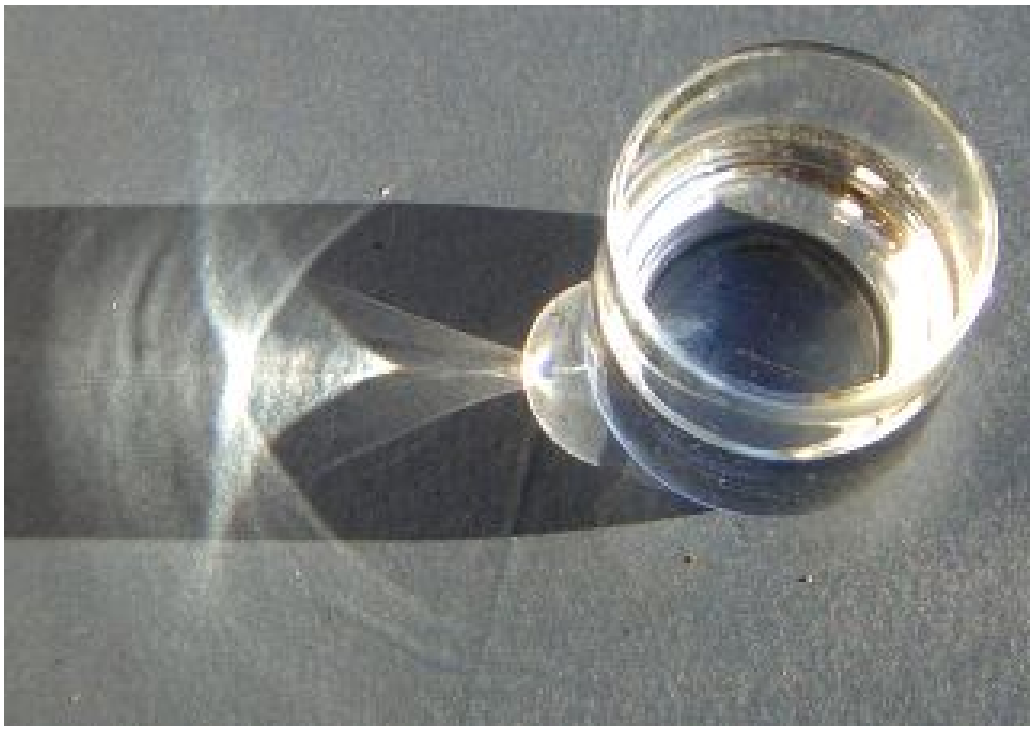


Figure 1: *Caustic*

We are now going to present some of the main technique used in the photorealistic graphics that are the basis of the theory from which our algorithm is born.

2 COMPUTER GRAPHICS ALGORITHMS

2.1 RAY TRACING

The first algorithm that we are going to introduce is the classic ray tracer. It works considering the approximation of the "vision" phenomenon commonly called geometrical optics (we assume that the light moves along straight lines) to simulate the lights. There are mainly two kind of algorithm that use this technique: the backward ray tracing (the classic one) and the forward ray tracing (usually never used by itself for reason we will explain below). The difference between those two approaches is the origin of the rays that we trace during the algorithm.
The big disadvantage that they have in common is the high computational cost due to the compute of all the intersections and to the difficulty to process scene with a a high amount of different surfaces. On the other hand it allow to process in a easy and intuitive way the illumination of a scene with satisfactory results.

2.1.1 *Classic ray tracing*

The main idea of the backward ray tracing is to emit rays from the point of view to the scene, following all the paths we have from the intersection of them with the objects inside the scene, going back to the source of the light, in order to outline all the visible surfaces.
The benefit of using this technique is that we have a good representation of the refraction and reflection effect, and shadows.
The biggest disadvantage, and the reason why we decided to implement our algorithm, is the inability to represent the inter-reflection effect between the light source and the objects in the scene, like caustics. The reason why it can't achieve these effects follows from the consideration that the probability of emitting enough rays that, beginning from the point of view, will pass through a focal point (just as the caustic) and then go back to the light source, is really low. To overcome this problem there is an approach called forward ray tracing that works in the reverse manner of the classic one.

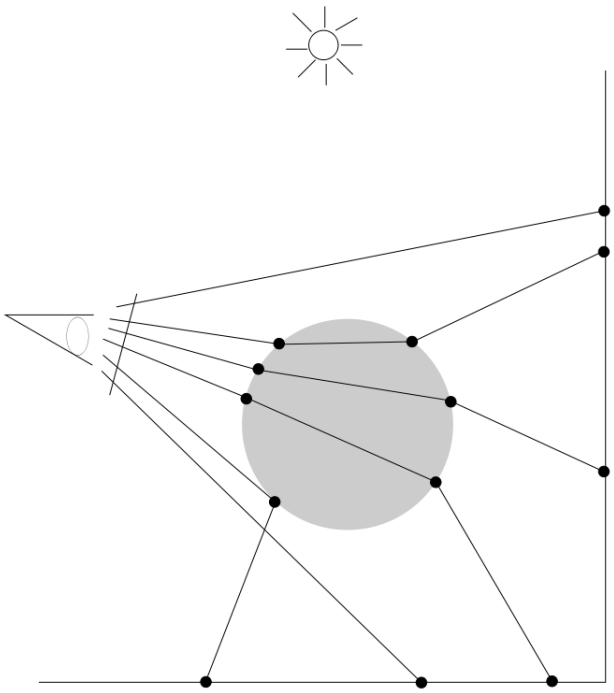


Figure 2: *Backward ray tracing*

2.1.2 *Forward ray tracing*

Forward ray tracing works, on the contrary of the backward ray tracing, from the light source to the point of view. In this way it's easier to find rays that generate inter-reflection effect, since there is a higher chance to shoot rays whose *path*, beginning from the light source, will end up in a focal point. This approach has an obvious weakness: we need to emit a really high amount of rays to get a detailed image, since the probability of tracing paths that will end up to the point of view is really low.
Let's have an example (Image 3): let's assume to have a light source and an object that stands between the point of view and this light source; with forward ray tracing we will trace a lot of rays that end up on the invisible surface, and way lesser that end up on the foreground one. With backward ray tracing, instead, we have a similar situation but completely reverse.
Beginning from this consideration Henrik Jensen came up with the photon mapping approach in [2].

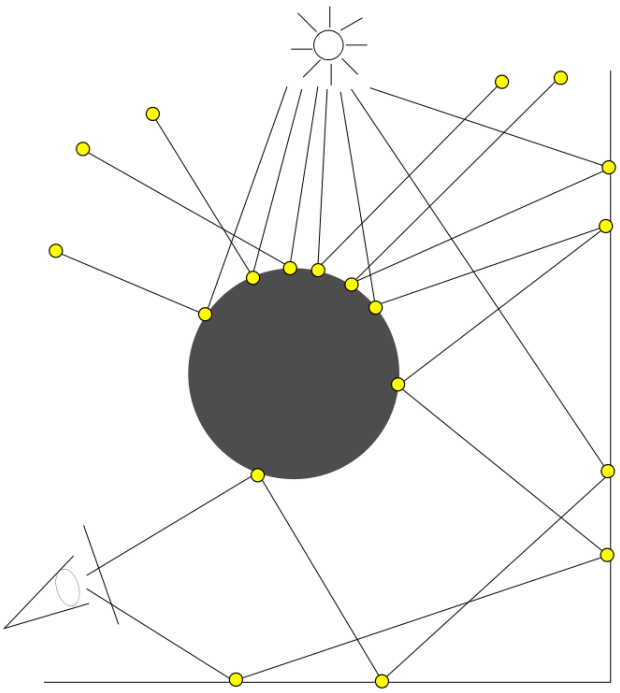


Figure 3: *Forward ray tracing applied in a scene with just one object in the middle and without any refraction properties, in reference to the example.*

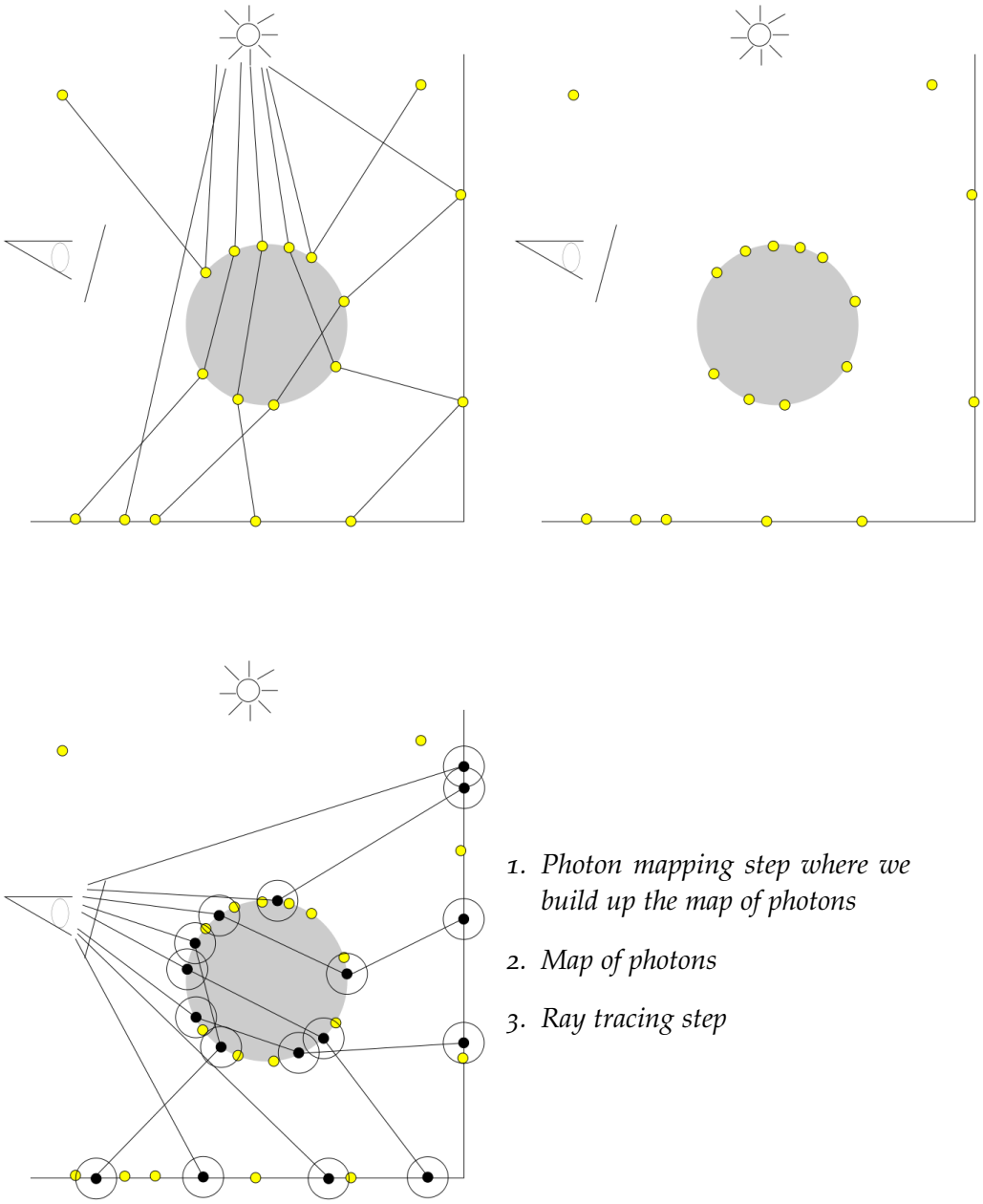
2.2 PHOTON MAPPING

Photon mapping is a bidirectional algorithm that sum up the efficiency of backward ray tracing and the completeness of forward ray tracing. It's divided in two steps: the ray tracing one (backward) and the photon mapping one (forward). Such as the ray tracing, we are going here below to introduce the two main approaches to photon mapping: the classic photon mapping and the reverse photon mapping, from which the progressive photon mapping comes, focus of this thesis. The main difference between this two algorithms is the order in which they use this two steps above.

2.2.1 *Classic photon mapping*

Classic photon mapping has a first step of photon tracing in which we trace rays from the light source to the scene, storing all the intersections and building up a real map of photons. The second step exploits ray tracing to render the image, using the map of photons to estimate the radiance in each point of the scene. This estimation is done by localizing, for each point that we get from the ray

tracing step, all the photons inside a proper research sphere and calculating the radiance in that point based on the density of the adjacent photons.



1. *Photon mapping step where we build up the map of photons*
2. *Map of photons*
3. *Ray tracing step*

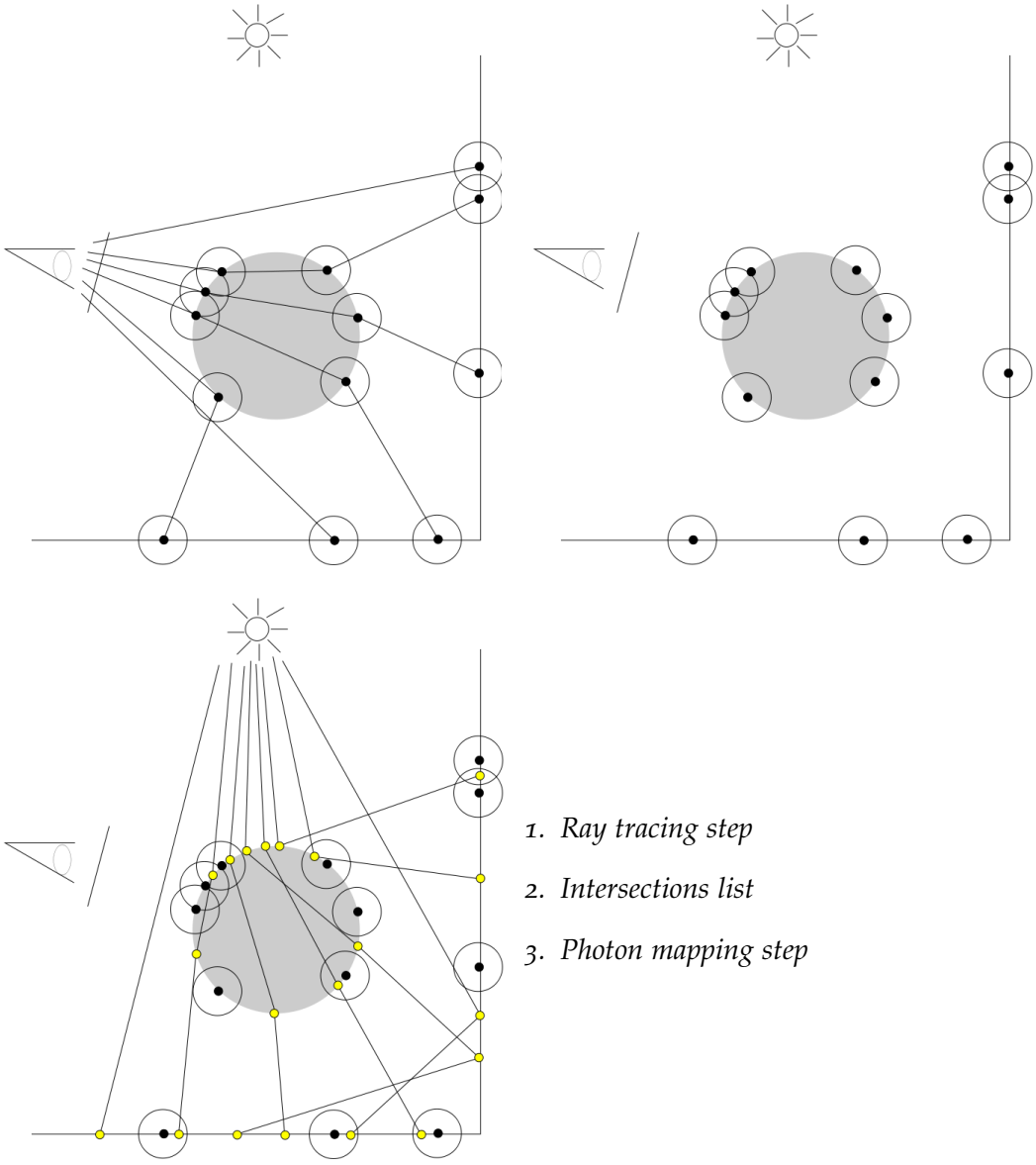
Figure 4: *Photon Mapping*

This approach has an obvious disadvantage: we need a huge amount of memory to store the photons found during the forward ray tracing step and moreover the final accuracy is usually limited from the total amount of photons stored.

To overcome this problem we could think to store the points found during the ray tracing step, that are way lesser than the ones we get from the photon tracing step (in fact a forward ray tracing step), and adjust the radiance according to the influence of the photons processed, beginning from the light source, with the photon tracing step. This is the approach brought up by Vlastimil Havran in [4] called reverse photon mapping.

2.2.2 *Reverse photon mapping*

In reverse photon mapping we have first a ray tracing step during which we store the intersections of every ray path with the scene, building up some sort of "map of the visible"; then we trace photons (photon tracing step) looking for the point, in a given neighborhood, among the stored ones, that each of these photons affect.
This approach keeps the advantages of the previous algorithm reducing drastically the memory occupation at the same time.



1. *Ray tracing step*
2. *Intersections list*
3. *Photon mapping step*

Figure 5: *Reverse Photon Mapping*

3

PROGRESSIVE PHOTON MAPPING

Progressive photon mapping comes from the reverse photon mapping and it optimizes the main steps. The first one is a ray tracing step in which we store all the intersections considering also rays reflections and refractions until the first non-reflecting and non-refracting surface[1]. Along with these intersections we also need to store other elements that we will process in the second step of the algorithm to calculate the right radiance in that point. Therefore we need to store:

- three-dimensional coordinates of the intersection point[2];
- the normal at that point;
- a vector that represents the direction of the radius generating the intersection;
- the BRDF[3];
- the abscissa and ordinate of the location of the pixel in the image[4];
- the coefficient that determines the brightness of that pixel;
- the radius of influence of the photons;

1 It's possible to limit the path length at a maximum number of intersections, since the light scatter at every intersection ensure that, for an appropriate maximum number of intersections, the remaining contribution of the light path is almost irrelevant to the overall radiance

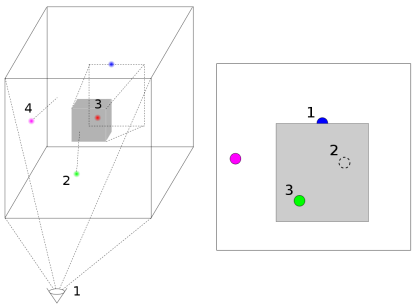
2 They are the coordinates of the point in the three-dimensional scenario that are actually going to appropriately overlap the actual coordinates of the two-dimensional pixel in the image.

Figure Note 1: 3D projection

1. *Point of view*
2. *Sphere in foreground*
3. *Sphere in background*
4. *Outer sphere*

Figure Note 2: 2D projection

1. *Visible hemisphere*
2. *Not visible red sphere*
3. *Sphere in front of the object*



3 Bidirectional Reflectance Distribution Function.

4 This is the actual location in the final image of this point.

- the number of photons that have influenced the radiance in this point;
- the non-normalized radiance.

Rays generated by the intersection:

1. *Emitted ray*
2. *Ray from the light*
3. *Reflected ray*
4. *Refracted ray*
5. *Primary*
6. *New emitted ray*

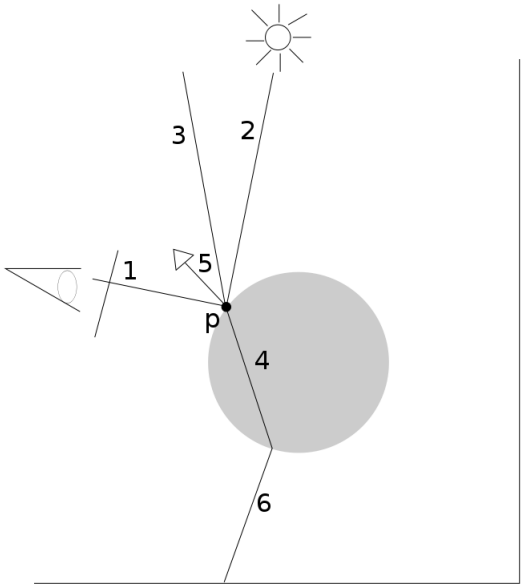


Figure 6: *Intersection of a ray*

In the specific case of the ray tracer used as the basis for the implementation (aRt), the first four components (point of intersection, the normal, direction of the main ray and BRDF) are all present in the data called *Hit* (the structure that identifies the intersection of the ray with the scene); instead, the location of the pixel and its brightness can be found in *ImageRay* (the structure that identifies the ray that propagates from the point of view to the scene). Finally the ray is set to a value in the range of $(0, 1)$ and both the number of photons and radiance are, obviously, set to zero.
The second step is a photon tracing step in which a series of rays are traced from the light sources to the scene, and for each intersection we check, among those stored, if one or more *hit* exist, in a neighborhood of that intersection, influenced by the brightness in that point; if so the radiance is updated as shown below, otherwise we consider the next intersection.
For the example image reference is made to Figure 5 since the order of the steps is the same one.

3.1 RADIANCE

Before moving on to examine in detail the expression of the radiance of the subject algorithm, we will briefly examine the photon mapping expression from which it is derived, according with [1].
Given a map of photons, the radiance of the point x can be estimated as:

$$L(x, \vec{\omega}) \approx \sum_{p=1}^{n} \frac{f_r(x, \vec{\omega}, \vec{\omega}_p)\phi_p(x_p, \vec{\omega}_p)}{\pi r^2} \tag{3.1}$$

where n is the number of photons adjacent used to estimated the radiance in x, ϕ_p is the flux of the p-th photon, f_r is the BRDF, $\vec{\omega}$ and $\vec{\omega}_p$ are the incoming and outgoing direction, and r is the radius of the sphere containing the n photons adjacent[5].
Increasing the density of photons the radiance estimation converges to the correct solution, making the photon mapping algorithm consistent. To ensure the proper convergence we need to use an infinite number of photons in the map of photons and in the radiance estimation. Furthermore the radius should converge to zero. These demands can be met using N photons in the map of photons, but only N^β within $\beta \in (0, 1)$ photons in the radiance estimation. As N becomes infinite both N and N^β become infinite as well, but N^β is infinitesimally smaller than N, which guarantees that r converges to zero. It's then possible to rewrite the formula (3.1) as in [2]:

$$L(x, \vec{\omega}) = \lim_{N\to\infty} \sum_{p=1}^{\lfloor N^\beta \rfloor} \frac{f_r(x, \vec{\omega}, \vec{\omega}_p)\phi_p(x_p, \vec{\omega}_p)}{\pi r^2}$$

We can now analyze the formulas for the convergence of the radiance that are characteristics of the subject algorithm.

3.2 REDUCTION OF THE RADIUS

We can analyze as first the reduction of the radius of influence of the photons. Suppose $R(x)$ is such radius and suppose $N(x)$ is the number of photons accumulated within it; suppose, then, $d(x)$ is the density of photons in the *hit* x defined as:

$$d(x) = \frac{N(x)}{\pi R(x)^2}.$$

5 We assume that the local set of photons represents the radiance of x, and that the surface is locally flat around x.

After $M(x)$ more photons in x, the new density $\hat{d}(x)$ is defined as follows:

$$\hat{d}(x) = \frac{N(x) + M(x)}{\pi R(x)^2}$$

Hence we need to see how to reduce the radius $R(x)$ of a value $dR(x)$. Assuming that the density of photons is constant around $R(x)$ we can compute the new number of photons $\hat{N}(x)$ around a disk of radius $\hat{R}(x) = R(x) - dR(x)$ as:

$$\hat{N}(x) = \pi \hat{R}(x)^2 \hat{d}(x) = \pi (R(x) - dR(x))^2 \hat{d}(x)$$

obviously we get that $\hat{N}(x) > N(x)$. Using, then, a relaxation parameter α within $(0, 1)$ the reduction of the brightness at each iteration is ensured. Thus rewriting $\hat{N}(x)$ as:

$$\hat{N}(x) = N(x) + \alpha M(x)$$

it is possible to compute the reduction of the radius as:

$$\pi (R(x) - dR(x))^2 \hat{d}(x) = \hat{N}(x)$$

$$\Leftrightarrow \pi (R(x) - dR(x))^2 \frac{N(x) + M(x)}{\pi R(x)^2} = N(x) + \alpha M(x)$$

$$\Leftrightarrow dR(x) = R(x) - R(x) \sqrt{\frac{N(x) + \alpha M(x)}{N(x) + M(x)}}$$

It follows that the reduced radius $\hat{R}(x)$ is calculated as:

$$\hat{R}(x) = R(x) - dR(x) = R(x) \sqrt{\frac{N(x) + \alpha M(x)}{N(x) + M(x)}} \tag{3.2}$$

In the specific case of our implementation we have $M(x) = 1$ since the ray and the other values are updated at every and each photon and not after sampling other $M(x)$ photons.

3.3 FLUX CORRECTION

After defining the criterion of reduction of the radius, we examine how the accumulated flow in x is calculated, where x is the intersection point of the ray coming from the direction $\vec{\omega}$; said $\tau(x, \vec{\omega})$ for $N(x)$ photons, we calculate it in the following way:

$$\tau_N(x, \vec{\omega}) = \sum_{p=1}^{N(x)} f_r(x, \vec{\omega}, \vec{\omega}_p) \phi'_p(x_p, \vec{\omega}_p)$$

where $\overrightarrow{\omega}_p$ is the direction of the incident photon and $\phi'_p(x_p, \overrightarrow{\omega}_p)$ is the non-normalized flux carried by the photon p. Analogously for $M(x)$ we get that:

$$\tau_M(x, \overrightarrow{\omega}) = \sum_{p=1}^{M(x)} f_r(x, \overrightarrow{\omega}, \overrightarrow{\omega}_p)\phi'_p(x_p, \overrightarrow{\omega}_p)$$

Assuming that the lighting and the density of photons within the disk is constant[6], we can write that:

$$\tau_{\hat{N}}(x, \overrightarrow{\omega}) = (\tau_N(x, \overrightarrow{\omega}) + \tau_M(x, \overrightarrow{\omega}))\frac{\pi\hat{R}(x)^2}{\pi R(x)^2}$$

$$= \tau_{N+M}(x, \overrightarrow{\omega})\frac{\pi\left(R(x)\sqrt{\frac{N(x)+\alpha M(x)}{N(x)+M(x)}}\right)^2}{\pi R(x)^2}$$

$$= \tau_{N+M}(x, \overrightarrow{\omega})\frac{N(x) + \alpha M(x)}{N(x) + M(x)} \tag{3.3}$$

where $\tau_{N+M}(x, \overrightarrow{\omega}) = \tau_N(x, \overrightarrow{\omega}) + \tau_M(x, \overrightarrow{\omega})$ and $\tau_{\hat{N}}(x, \overrightarrow{\omega})$ is the lightning reduction for $\hat{N}(x)$ photons.

3.4 RADIANCE ESTIMATION

After analyzing all the elements necessary to calculate the radiance, and taking into account that $\tau(x, \overrightarrow{\omega})$ must be normalized, and therefore we have to keep track of how many photons have been used (let's say they are $N_{emitted}$), we can estimate the radiance as follows:

$$L(x, \overrightarrow{\omega}) = \int_{2\pi} f_r(x, \overrightarrow{\omega}, \overrightarrow{\omega}')L(x, \overrightarrow{\omega}')(\overrightarrow{n} \cdot \overrightarrow{\omega}')d\omega'$$

$$\approx \frac{1}{\Delta A}\sum_{p=1}^{n} f_r(x, \overrightarrow{\omega}, \overrightarrow{\omega}_p)\Delta\phi_p(x_p, \overrightarrow{\omega}_p)$$

$$= \frac{1}{\pi R(x)^2}\frac{\tau(x, \overrightarrow{\omega})}{N_{emitted}} \tag{3.4}$$

6 The assumption that the density of photons and the surrounding illumination is constant within the disk may not be correct initially, but becomes incrementally true as the radius decreases, except for the points that are exactly in the illumination discontinuity. That's not a problem because the lighting in the discontinuity is not defined and the probability of having an *hit* exactly in a discontinuity is zero, [1].

It should be noted that the radius $R(x)$ does not get reduced if the disk defined by $R(x)$ is within a region not illuminated (ie $M(x) = 0$). Despite this fact seems to compromise the condition of consistency, however the radiance converges to the correct solution $L(x, \overrightarrow{\omega}) = 0$, since $\tau(x, \overrightarrow{\omega})$ does not increase and $L(x, \overrightarrow{\omega}) \to 0$ if $N_{emitted} \to \infty$.
Although intuitively we are convinced of the correctness of these conclusions, it has not yet been done an analysis of the convergence properties of $R(x)$ and $N(x)$; however, please refer to the chart in [1] in which it is shown that the estimate of the radiance gradually converges to the correct value of $L(x)$, as the radius $R(x)$ reduces and the number of photons $N(x)$ increases.

4 PHOTON MAPPING REVISITED

The algorithm presented differs from progressive photon mapping for some aspects of the implementation: the progressive photon mapping stores the intersections of the ray tracing pass in a list and it also stores in a kd-tree (a k-dimensional storage structure described below) the intersections of the photons produced by the photon tracing pass, creating a map of photons with which you update the *hit* of the list; then this map is discarded and a new one is created repeating the process as many times as necessary for the radiance to converge. In the algorithm that we implemented the *hit* of the ray tracing pass are stored in the kd-tree and the photons are processed one at a time updating the *hit* that they affect.
Following, there are the formulas of the original progressive photon mapping, according with [1], and there are clarifications regarding the different implementation.
Certain implementation choices have been made with the aim to realize an algorithm that could be used on any type of hardware ensuring a certain speed in runtime and a constant memory allocation in relation to a satisfactory convergence of the radiance, at the expense, although, of the speed of the latter.

4.1 STORAGE STRUCTURES

We introduce here below the two main *struct*: *HitPoint*, the data containing all the necessary information on a single intersection obtained during the ray tracing pass, and the kd-tree, the structure of research of the *hit* influenced by the photon processed at that time.

4.1.1 *HitPoint*

HitPoint is used to store the location of an *hit*, its normal, the direction from which the incident ray comes and the BRDF; all these information are already inside an existing structure in aRt called, precisely, *Hit*. We also need the position of the pixel (two integers for the coordinates 2d), the "pixel weight" (a *ColorWeight*), a *double* for the radius of that *hit*, an integer that counts how many photons have influenced the radiance in that point and a *Color* that represents the accumulated

flux.
Hence was created a *struct* in C ++ like this:

```
struct HitPoint {

  Hit *hitpt;

  int i, j;

  ColorWeight w;

  double R;

  int N;

  Color t;

};
```

data stored inside the kd-tree.

4.1.2 *Kd-tree*

For the purposes of the subject algorithm, we need to look for all the *hit* adjacent to the photon examined in a range defined by a sphere with center at the intersection of this with the scene and having radius as a default value that represent the maximum distance between the photon and the *hit* so that it influences it. Since, as it has been said, the image is ideally in 3d, the search must be made among all the possible three-dimensional points obtained with the ray tracing pass. It follows that these points should all be stored in an appropriate structure so that we can access them as required. We need, therefore, a structure that let us search in the space quickly. In this regard, we have used the kd-tree, the complexity of which is satisfactory for the operations used by the algorithm, as we will see shortly.
A three-dimensional kd-tree is a structure that stores at the position given by the point P(x,y,z) any data (in this algorithm the data is the *HitPoint* concerning the point examined). The technical construction of the kd-tree, as stated in [5], is the following:

> The kd-tree is a binary tree in which every node is a k-dimensional point. Every non-leaf node can be thought of as implicitly generating a splitting hyperplane that divides the space into two parts, known as subspaces. Points to the left of this hyperplane represent the left subtree of that node and points right of the hyperplane are represented

by the right sub-tree. The hyperplane direction is chosen in the following way: every node in the tree is associated with one of the k-dimensions, with the hyperplane perpendicular to that dimension's axis. So, for example, if for a particular split the "x" axis is chosen, all points in the sub-tree with a smaller "x" value than the node will appear in the left sub-tree and all points with larger "x" value will be in the right sub tree. In such a case, the hyperplane would be set by the x-value of the point, and its normal would be the unit x-axis.

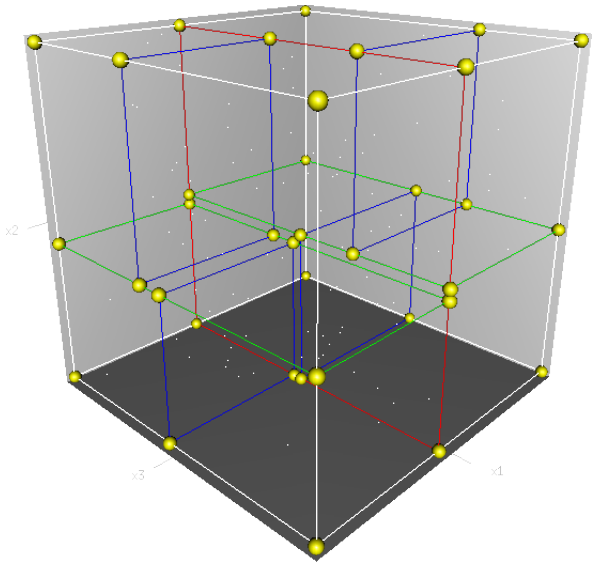


Figure 7: *Kd-tree*

In our implementation the point P(x,y,z) is defined as an array of three *double* called *KDPoint*:

```
typedef array<double, 3> KDPoint;
```

Given two points the search works in the cube that has as opposite vertices these two points.
Before showing the actual code, we analyze why, actually, this is the most suitable structure for the implementation of the algorithm, as described in [6]:

- Building: building a static kd-tree of n points takes $O(n \log^2 n)$ time if we use a sorting algorithm with complexity $O(n \log n)$, to compute the median in each level. The complexity is $O(n \log n)$ if we use a linear algorithm to compute the median.

- Insert: inserting a new point into a balanced kd-tree takes $O(\log n)$ time.

- Removal: removing a point from a balanced kd-tree takes $O(\log n)$ time.

- Query: querying in a cubic/hyper-cubic neighbor in a balanced kd-tree takes $O(n^{1-1/k} + m)$ time, where m is the number of points belonging to this neighbor, and k is the size of the kd-tree.

For the purposes of the algorithm the main interest is about the insert speed[1] and the research speed.
The kd-tree that stores at the location *KDPoint* the item *HitPoint* has been defined as:

```
typedef ssrc::spatial::kd_tree<KDPoint, HitPoint*> Tree;
```

In order to develop this algorithm we've been using the kd-tree of the package[7].

4.2 CLASSES USED TO IMPLEMENT THE ALGORITHM

The two classes that implement the algorithm are: the class that builds the tree during the ray tracing step (KDTrees.h) and one that updates the values of the *HitPoints* during the photon tracing step (photontrace.h).

4.2.1 *KDTrees.h*

The KDTrees.h class is the class containing the constructor of the kd-tree and the initialization of the values that compose it.
It's been said that the radius $R(x)$ has to decrease to ensure the proper convergence and that the parameter $\alpha \in (0, 1)$ is also an important parameter to achieve this goal, as follows from the formula (3.2); however we didn't say that the arbitrary choice of these parameters significantly affect the final result. Indeed the smaller the $R(x)$ initially is and the lesser photons will affect that point (for $R(x) = 0.01$ the resulting image is almost identical to the one obtained with only the ray tracer); for an $R(x)$ too big we would consider an excessive intake of brightness in every single *hit*. The parameter α instead represents a sort of relaxation parameter, which affects not only the reduction of the radius, but also the intensity of the color (for $\alpha = 0.01$ we get an image really bright, instead for $\alpha = 0.99$ the picture is almost entirely shaded off[2]). In this regard, therefore, these parameters are chosen by those who use the algorithm classes by passing them to the constructor:

1 It's about the insert speed and not about the construction speed because the kd-tree is built dynamically and not statically.
2 Obviously the lighting effect and the gradient effect are visible in the points affected by a larger number of photons.

```
KDTree (double radius, double param){
  initial_radius=radius;
  squared_initial_radius=radius*radius;
  alpha=param;
}
```

We can see that in the constructor we uses a variable `squared_initial_radius` set to the square of the radius. The reason is simply that it is better to work with squares rather than with roots, since, computationally speaking, the calculation of a root is much more expensive than the calculation of a product; hence the formula (3.2) becomes:

$$\hat{R}(x) = R(x) - dR(x) = R(x)\sqrt{\frac{N(x) + \alpha M(x)}{N(x) + M(x)}} \Leftrightarrow \hat{R}(x)^2 = R(x)^2 \frac{N(x) + \alpha M(x)}{N(x) + M(x)}$$

said $\bar{R}(x)$ the radius squared, we get:

$$\Rightarrow \hat{\bar{R}}(x) = \bar{R}(x) \frac{N(x) + \alpha M(x)}{N(x) + M(x)}$$

where $\hat{\bar{R}}(x)$ is the new radius squared and $\bar{R}(x)$ is the current radius squared. The formula (3.3) remains the same as it's not affected by $R(x)$:

$$\tau_{\hat{N}}(x, \overrightarrow{\omega}) = \tau_{N+M}(x, \overrightarrow{\omega}) \frac{N(x) + \alpha M(x)}{N(x) + M(x)}$$

And finally, the formula (3.4) is adjusted in this way:

$$L(x, \overrightarrow{\omega}) = \frac{1}{\pi R(x)^2} \frac{\tau(x, \overrightarrow{\omega})}{N_{emitted}} \Rightarrow = \frac{1}{\pi \bar{R}(x)} \frac{\tau(x, \overrightarrow{\omega})}{N_{emitted}}$$

With these optimized formulas, the variable identifying the radius squared is then placed in `initial_radius` in the inserting method of the tree, *KDAdd*:

```
inline HitPoint *KDAdd(Hit *h, ImageRay *ir) {

  Point kdtmp = h->getPoint();
  KDPoint kdpt = {{kdtmp.x, kdtmp.y, kdtmp.z}};
  HitPoint *hp = new HitPoint();

  hp->hitpt = h;
  hp->i = ir->getI();
  hp->j = ir->getJ();
  hp->w = ir->getW();
  hp->R = squared_initial_radius;
  hp->N = 0;
  hp->t = 0;
```

```
    (*this)[3][kdpt] = hp;

    return hp;
  }
```

This method is responsible only for the initialization of the variables at the correct values, for inserting those values to the position *kdpt* (the *KDPoint* of research) and for returning the *HitPoint*. It's called almost at the end of the ray tracer, before it adds the color calculated with it at the image. *KDAdd* returns an *HitPoint* since it is already possible to update the entire kd-tree during the ray tracing step; this algorithm works its way back from the point of view to the light source and, therefore, each intersection represents a photon emitted from the light source to the scene, but traced backward. Hence for each intersection it is possible to apply the reduction of the radius and the calculation of the flux by increasing the number of photons. To this end, therefore, we need the *HitPoint* so that we can update those values.

4.2.2 *photontrace.h*

The file photontrace.h contains the real body of the algorithm. After the construction of the kd-tree during the ray tracing step, a list of lights and a "reserve of photons" are created, and after that the method photonTrace is called:

```
TrivialPhotonsSampler *photonsSampler =
  new TrivialPhotonsSampler(&ll);

PhotonsReservoir *photonsReservoir =
  new PhotonsReservoir(num_big_iter,photonsSampler,
                       outsideMedia,10,1.0e-3);

photonTrace(photonsReservoir,&world,&tree);
```

Let's analyze them individually:

- `TrivialPhotonsSampler` is a sampler of lights that takes as a parameter a list of light sources.
- `PhotonsReservoir` is a class that has to queue a new *PhotonRay*, after each intersection of the photons with the scene, with origin in the intersection and as direction the new direction depending from the type of surface intersected, so that it can then be reprocessed as it were a new photon; this

3 Despite being increasingly discouraged in any programming language, the use of explicit downcast, it was necessary to use one even if in total safety. In fact, we store to the location *kdpt*, that is a *KDPoint*, the value of *hp*, that is an *HitPoint*, exactly according with the definition used for the *tree*. Besides the fact itself that it is allowed to use the explicit downcast implies that in certain situation it is absolutely necessary.

is done in order to process entirely[4] and recursively each and every *path* originated from the light source.

- photonTrace is the method that handles the entire algorithm, calling again the method updateTree, if we get an intersection of the ray with the scene.

In updateTree the formulas are "translated" into code, selecting the *hit* to update and doing the update itself. We said that, after we find an intersection of *PhotonRay* with the scene, we need to verify that the illumination contribution of that photon falls within the sphere of influence of the *hit* belonging to the research neighborhood. In other words, we have to make sure that the *hit* inside the sphere of radius $R(x)$ and centered in P(x,y,z) in the intersection of the photon (which in our case becomes a cube with opposite vertices $P(xR(x), yR(x), zR(x))$ and $P(x+R(x), y+R(x), z+R(x))$) contain themselves that photon.
This is accomplished by the following code:

```
  double radius=tree->getInitialRadius();
  Point pp=h->getPoint();
  KDPoint lower={{pp.x-radius, pp.y-radius, pp.z-radius}};
  KDPoint upper={{pp.x+radius, pp.y+radius, pp.z+radius}};

  for(Tree::const_iterator it = tree->begin(lower, upper);
      !it.end_of_range(); ++it) {

    HitPoint *hp=it->second;
    Point hpt=hp->hitpt->getPoint();

    double r=((pp.x-hpt.x)*(pp.x-hpt.x)+(pp.y-hpt.y)*
    (pp.y-hpt.y)+(pp.z-hpt.z)*(pp.z-hpt.z));

    if ((hp->R)>=r) {
...
```

where p is the intersection point of the photon, lower and upper are the extremes of the searching cube, hpt is the point of the *hit* under consideration and r is the radius squared of the sphere with center in pp and having hpt on the surface; finally, the if checks if the photon pp belongs to the sphere of influence generated by the *hit* having as center the *hit* itself and as radius $R(x)$.
In the positive case we then proceed to update all the parameters of the *HitPoint* according with [1]:

```
  int M=1;
  double alpha=tree->getAlpha();
```

4 Actually this method takes also a maximum limit of subdivision of the *path*, since the light scatter ensures that the contribution to the radiance of a ray reflected or refracted an appropriate number of times (in this case 10), is nearly null.

```
  const Media *m = pr->getMedia();
  Vector d = pr->getD();
  double R1=hp->R;
  int N1=hp->N;

  hp->R = R1*(N1+(alpha*M))/(N1+M);

  hp->N=N1+1;

  Vector id(-d.x,-d.y,-d.z);
  pair<ColorWeight,const Media *>  vbsdf =  h->bsdf[5](id,m);

  const ColorWeight cw = vbsdf.first;

  Color tm= cw*intensity;

  Color tn=hp->t;

  hp->t=((N1+(alpha*M))/(N1+M))*(tn+tm);
```

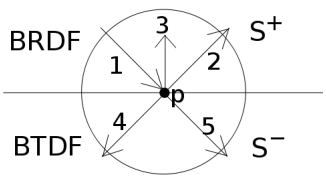
The formula (3.2) is then implemented as $hp \rightarrow R = R1 * (N1 + (alpha * M))/(N1 + M)$, then we increase $N(x)$ by a single photon ($M(x) = 1$), and therefore we get the formula (3.3) as $hp \rightarrow t = ((N1 + (alpha * M))/(N1 + M)) * (tn + tm)$. Once updated the entire kd-tree for each *PhotonRay* traced, the values for the color contribution of the *hit* are normalized by calculating the radiance as follows:

```
  for(Tree::const_iterator it = tree.begin(lower,upper);
  !it.end_of_range(); ++it) {

  HitPoint *hp=it->second;

  Color RE=1.0/(M_PI*(hp->R)) * ((hp->t)/num_big_iter);

  ImgRadEv->addColor(hp->i,hp->j,RE);

};
```

5 In this algorithm we use the BSDF (Bidirectional Scattering Distribution Function) and the BRDF because we consider both the reflections (BRDF) and the transmission (BTDF, Bidirectional Transmittance Distribution Function), whose union is the BSDF.

Picture Note: BSDF

1. *Incoming ray*
2. *Reflected ray*
3. *Normal*
4. *Transmitted ray*
5. *Transmitted ray*



where lower and upper represent the extremes of the image and num_big_iter represents the total number of photon traced, called $N_{emitted}$, thus getting the formula (3.4). We then proceed to add the color to the image in the corresponding pixels of coordinate P(i,j).

5

EXAMPLES AND RESULTS

We are now going to present two classes of images, Sphere and Glasses, with the relevant data, in the image obtained with only the ray tracer and in the one obtained by the above algorithm. These data are:

- Execution time;
- The maximum amount of photons that have influenced a single *HitPoint*;
- The total amount of *HitPoints* in the kd-tree.

These images were processed on a computer with an AMD Athlon dual core 2.1 GHz with 2GB of RAM, running Linux (Ubuntu 2.6.28-18-generic), with an iteration of 10 million photons[1] and with $R(x) = 0.5$ and with $\alpha = 0.5$[2].
The computational and intuitive simplicity of the images that belong to the type Sphere let us show the photorealism in the implementation of three specific interactions between caustics and the scene.
In Glasses, instead, despite being a complex image[3], as we can see by reading the information below, and despite it is not that easy to intuitively understand the correctness of the whole image, we can still note the likelihood of the 6 main caustics.

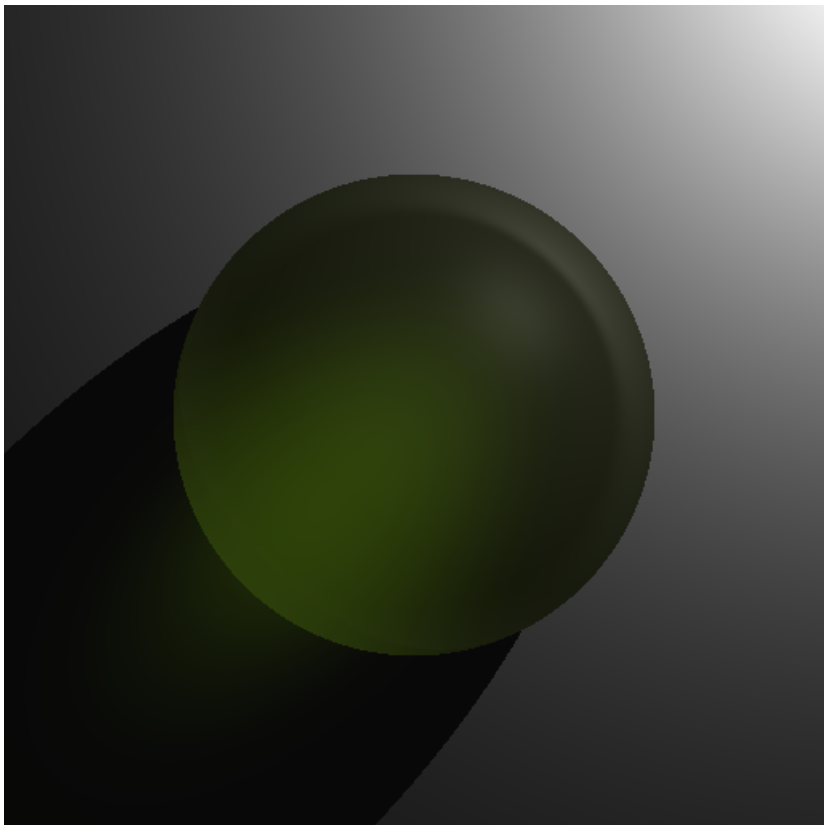
1 The algorithm processes a quantity of photons equal to the number of iterations specified on the basis of the resolution for the desired number of light sources; This ensures a uniform resolution between images of the same scene with a different number of lights.

2 After sampling many images with different $R(x)$ and α, we have experimentally come up to the conclusion that these were the two best values for our implementation.

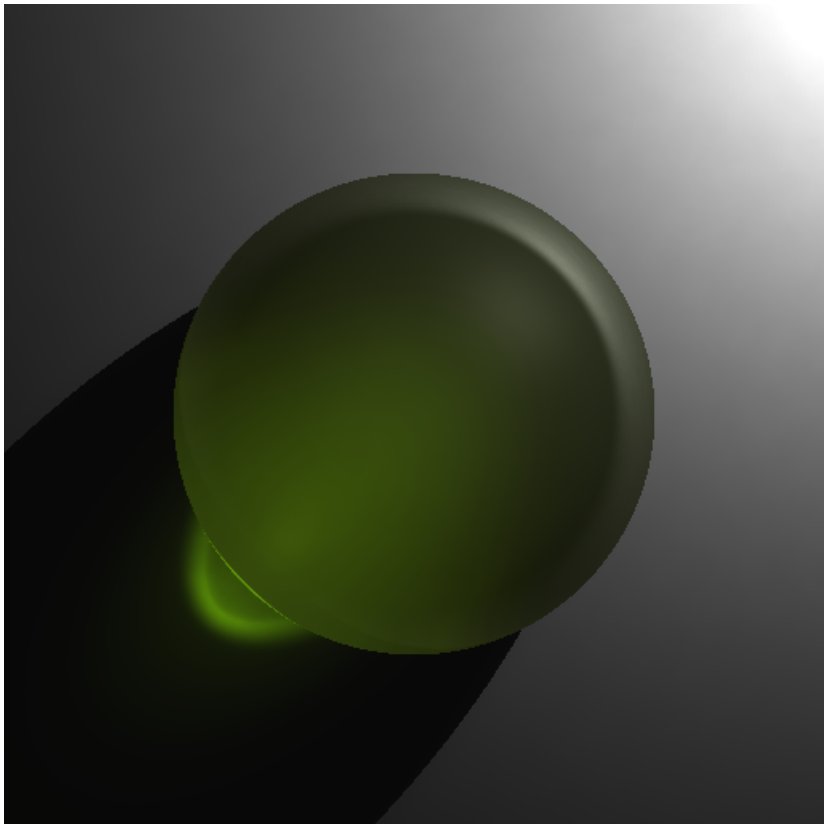
3 The execution of the algorithm showed a considerable number of interruption of the *path*.

5.1 SPHERE

Sphere is an image depicting a ball leaning on a flat surface, made out of a material simulating the glass with a green shade, and with the light source in the top right corner.



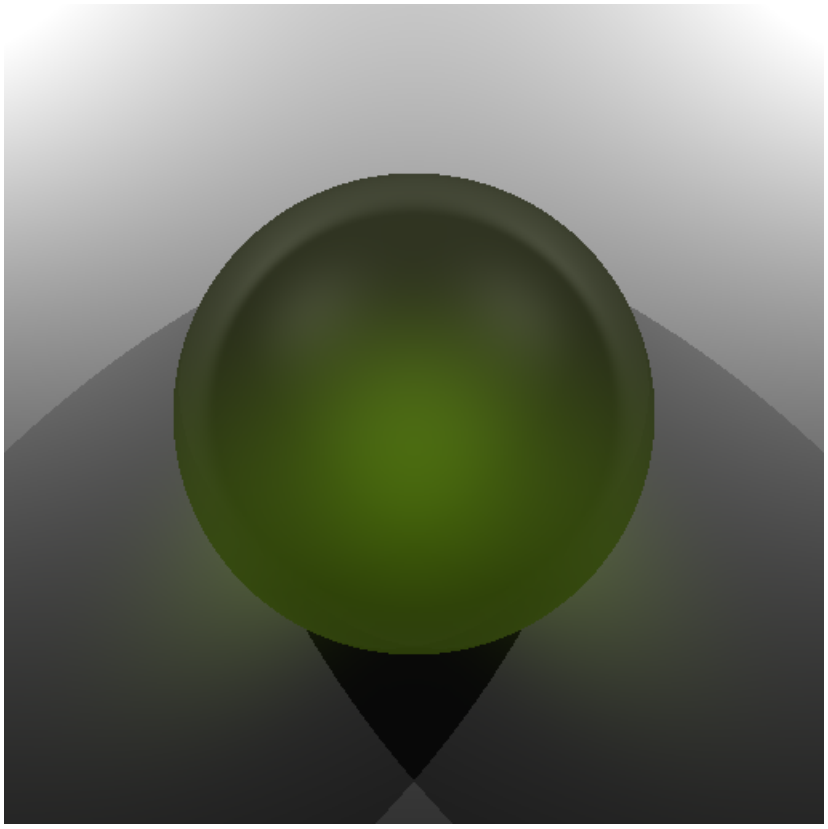
The image at the top was made with the only use of the ray tracer.



The image located at the bottom was made with the subject algorithm and with an iteration of 10 million photons.

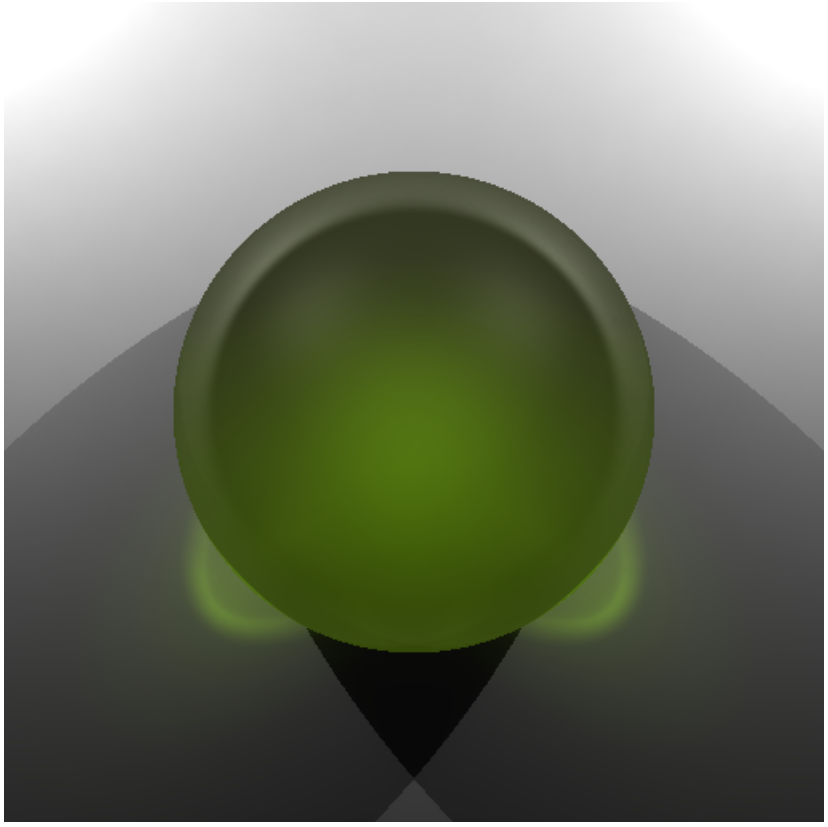
- *Execution time: 1:24:17*
- *N. Max Photons: 3·635*
- *Num HitPoint: 356·180*

5.1.1 *Double Sphere*



Double Sphere is a variant of Sphere with an extra light source on the top left corner, in order to show two caustics and the reduction of the contrast between these and the shadow.

The image at the top was made with the only use of the ray tracer.

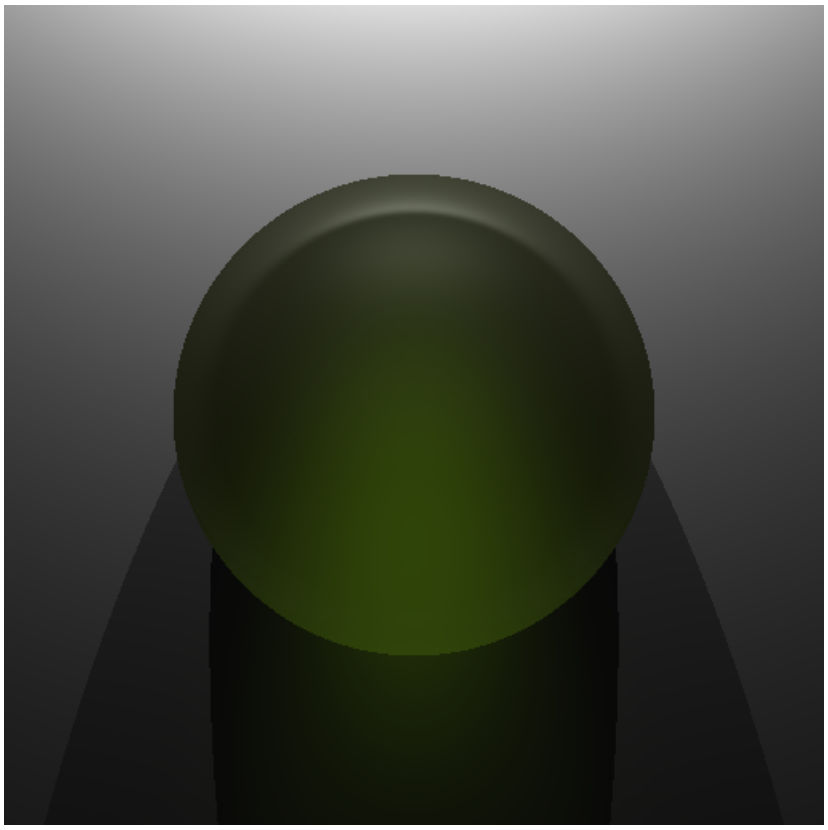


The image located at the bottom was made with the subject algorithm and with an iteration of 10 million photons.

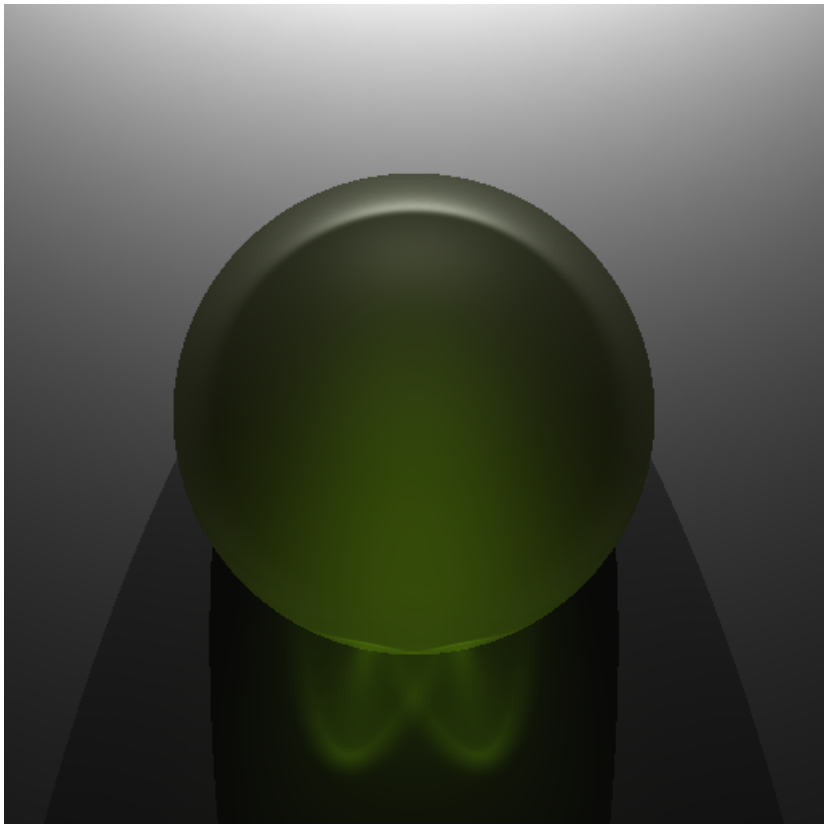
- *Execution time: 2:47:38*
- *N. Max Photons: 3·879*
- *Num HitPoint: 356·180*

5.1.2 *Intersection Sphere*

Intersection Sphere is a variant of Sphere with two light sources, close to each other, placed in the center on the top of the scene, in order to obtain two overlapping caustics.



The image at the top was made with the only use of the ray tracer.



The image located at the bottom was made with the subject algorithm and with an iteration of 10 million photons.

- *Execution time: 2:14:08*
- *N. Max Photons: 3·330*
- *Num HitPoint: 356·180*

5.2 GLASSES

Glasses is an image depicting 9 balls arranged in 3 aligned rows, made out of a material simulating the glass with the ball in the middle having a red shade and with a light source in the top right corner.

The image at the top was made with the only use of the ray tracer.

The image located at the bottom was made with the subject algorithm and with an iteration of 10 million photons.

- *Execution time: 29:52:22*
- *N. Max Photons: 10·671*
- *Num HitPoint: 1·107·174*

6 CONCLUSIONS

We developed an algorithm, in order to integrate it in aRt, to allow the realization of the inter-reflection phenomena between objects beginning from an approach called progressive photon mapping presented in [1].
The objective was to create an algorithm with a constant use of memory that would combine a good computational speed on common hardware with a photorealistic realization of this phenomena.
The comparison with the images generated with the only ray tracer shows clearly how the photon mapping revisited ensure a more complete representation of the distribution of the light in the space. In particular, examining the data on the three images of Sphere, we notice that the memory, represented by the number of *HitPoints* in the kd-tree, is constant despite the number of light sources in the scene.

BIBLIOGRAPHY

[1] T. Hachisuka, S. Ogaki, H. W. Jensen, *Progressive Photon Mapping*, ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia 2008), Singapore, December 2008. (Cited on pages 11, 13, 14, 15, 21, and 31.)

[2] H. W. Jensen, *Global Illumination using Photon Maps*, Dagstuhl Seminar on Rendering, pages 18, June 1996. (Cited on pages 4 and 11.)

[3] H. W. Jensen, *Realistic Image Synthesis using Photon Mapping*, AK Peters, Natick (Massachusetts) 2001.

[4] V. Havran, R. Herzog, H.-P. Seidel, *Fast final gathering via reverse photon mapping*, Computer Graphics Forum (Proceedings of Eurographics 2005) 24, 3 September 2005. (Cited on page 7.)

[5] Wikipedia, *Kd-tree*, http://en.wikipedia.org/wiki/Kd-tree, construction section. (Cited on page 16.)

[6] Wikipedia, *Kd-tree*, http://en.wikipedia.org/wiki/Kd-tree, complexity section. (Cited on page 17.)

[7] libssrckdtree-1.0.0, *libssrckdtree Generic k-d tree C++ template library*, http://www.savarese.com/software/libssrckdtree/. (Cited on page 18.)