Volume xx (200y), Number z, pp. 1–31

Specular Effects on the GPU: State of the Art

László Szirmay-Kalos, Tamás Umenhoffer, Gustavo Patow, László Szécsi, Mateu Sbert

Budapest University of Technology and Economics, Budapest, Magyar Tudósok krt. 2., H-1117, HUNGARY
Email: szirmay@iit.bme.hu

Abstract
This survey reviews algorithms that can render specular, i.e. mirror reflections, refractions, and caustics on the GPU. We establish a taxonomy of methods based on the three main different ways of representing the scene and computing ray intersections with the aid of the GPU, including ray tracing in the original geometry, ray tracing in the sampled geometry, and geometry transformation. Having discussed the possibilities of implementing ray tracing, we consider the generation of single reflections/refractions, inter-object multiple reflections/refractions, and the general case which also includes self reflections or refractions. Moving the focus from the eye to the light sources, caustic effect generation approaches are also examined.

Keywords: Ideal reflection/refraction, ray tracing, environment mapping, distance maps, geometry transformation, photon mapping, caustics, GPU

ACM CCS: I.3.7 Three-Dimensional Graphics and Realism

1. Introduction

Mirrors, very shiny metallic objects, refracting transparent surfaces, such as glass or water, significantly contribute to the realism and the beauty of images (Figure 1). Thus, their simulation in virtual environments has been an important task since the early days of computer graphics [Whi80]. Unfortunately, such phenomena do not fit into the *local illumination model* of computer graphics, which evaluates only the *direct illumination*, i.e. light paths originating at a light source and arriving at the eye via a single reflection. In a mirror, however, we see the reflection of some other surface illuminated by the light sources, thus rendering a mirror requires the simulation of longer light paths responsible for the *indirect illumination*. While in the local illumination model a surface point can be shaded independently of other surface points in the scene, indirect illumination introduces a coupling between surfaces, so the shading of a point requires information about the scene globally.

Before 2006, GPUs could only be controlled through graphics APIs such as Direct3D or OpenGL. These APIs are based on the concepts of the incremental rendering pipeline and present the GPU to the application developer as a pipeline of programmable vertex, geometry, and fragment shaders, and also non-programmable but controllable fixed function stages. One of these fixed function stages executes rasterization, which is equivalent to tracing a bundle of rays sharing the same origin, and passing through the pixels of the viewport. The first hits of these rays are identified by the z-buffer hardware. Since this architecture strongly re-

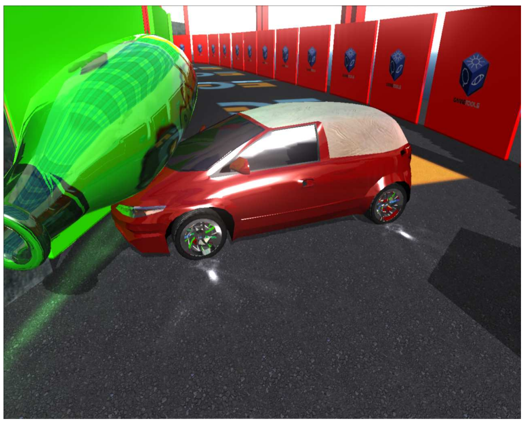

Figure 1: *Specular effects in a game, including reflectors, refractors, and caustics (http://www.gametools.org).*

flects the concepts of incremental rendering, and processes a vertex and a fragment independently of other vertices and fragments, the utilization of the GPU for other algorithms requires their translation to a series of such rendering passes [Bly06, OLG*05]. The result of a pass can be stored in a texture that can be read by the shaders during the subsequent passes.

The recently introduced *CUDA* [NVI07], on the other hand, allows the programming of the GPU as a general data-parallel computing device, and does not require the translation of the algorithm to the concepts of the incremental rendering pipeline.

1.1. Specular surfaces

A light beam travels along a straight line in a homogeneous material, i.e. where the *index of refraction* ν and *extinction coefficient* κ are constant (note that in inhomogeneous materials the light does not follow a straight line but bends according to the change of the refraction index [IZT*07]). The index of refraction is inversely proportional to the speed of light and the extinction coefficient describes how quickly the light diminishes inside the material. We shall assume that these properties change only at surfaces separating different objects of homogeneous materials.

According to the theory of electromagnetic waves, when a light beam arrives at a surface it gets reflected into the *ideal reflection direction* specified by the *reflection law* of the geometric optics, and gets refracted into the refraction direction according to the *Snellius–Descartes law of refraction*. Incident radiance L^{in} coming from a direction that encloses *incident angle* θ' with the surface normal is broken down to *reflected radiance* L^r and *refracted radiance* L^t according to *Fresnel function* $F(\theta')$, which depends both on the wavelength and on the incident angle:

$$L^r = L^{in}F(\theta'), \quad L^t = L^{in}(1-F(\theta')). \tag{1}$$

The Fresnel function can be expressed from incident angle θ' and from material properties ν and κ. If the light is not polarized, the Fresnel factor can be approximated in the following form [Sch93, LSK05]:

$$F(\theta') = F_0 + (1-F_0)(1-\cos\theta')^5,$$

where

$$F_0 = \frac{(\nu-1)^2+\kappa^2}{(\nu+1)^2+\kappa^2} \tag{2}$$

is the Fresnel function (i.e. the probability that the photon is reflected) at perpendicular illumination. For polarized light and crystals, more complicated models can be used [WTP01, GS04, WW08].

The normal vector and consequently incident angle θ' may change on the surface that is visible in a single pixel, thus the pixel color will be the average of reflections or refractions. Surfaces exhibiting strong normal vector variations on microstructure level are called optically rough. Rough surfaces are characterized by their *BRDF*s that depend not only on the Fresnel function but also on the probabilistic properties of the microstructure geometry [CT81, KSK01, HTSG91].

On the other hand, when the surface is smooth, the normal vector can be assumed to be constant in a small area corresponding to a pixel, thus the laws of geometric optics and equation 1 describe the reflected radiance even on the scale of pixels (i.e. on human scale). Smooth surfaces are often called *specular surfaces*, *mirrors*, or *ideal reflectors/refractors*.

In this review we consider only ideal reflector and refractor surfaces separating homogeneous regions. However, we note that as rough surface reflection can be imagined as the average of smooth surface reflections, the combination of the surveyed methods with integration or filtering can render not perfect, so called *glossy reflections* as well [KVHS00, KM00, RH01, HSC*05].

1.2. Rendering specular surfaces

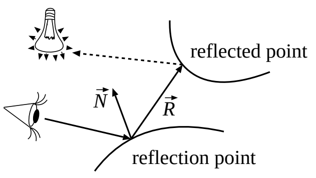

Figure 2: *Specular effects require searching for complete light paths in a single step.*

Rendering specular surfaces can be regarded as a process looking for light paths containing more than one scattering point (Figure 2). For example, in the case of reflection we may first find the point visible from the camera, which is the place of reflection. Then from the *reflection point* we need to find the *reflected point*, and reflect its radiance at the reflection point toward the camera.

Mathematically, the identification of the reflected point is equivalent to tracing a ray of parametric equation

$$\vec{r}(d) = \vec{x} + \vec{R}d, \tag{3}$$

where ray origin $\vec{x}$ is the reflection point, normalized ray direction $\vec{R}$ is the reflection direction, and ray parameter d is the distance along the ray. We have to calculate ray parameter d^* that corresponds to the first ray–surface intersection.

Real-time ray tracing and particularly real-time specular effects have been targeted by both CPU and GPU approaches and also by custom hardware [WSS05]. CPU approaches use sophisticated data structures [Hav01, RSH05, WK06], handle data in a cache efficient way, try to trace packets of rays simultaneously, and heavily exploit the SSE2 and 3DNow! instruction sets of current CPUs [WKB*02, WBS03]. With these optimizations and exploiting the power of a cluster of computers, these methods can provide real-time reflections, refractions, and even caustics [GWS04, Wym05b]. However, single CPU solutions are still not fast enough, especially when the scene is dynamic.

Computing specular reflections and refractions involves tracing a primary ray up to the reflective/refractive surface, computing the reflected or refracted ray, and recursively tracing the successive rays reflecting or refracting on specular surfaces until a non-specular surface is reached. Most of the GPU based techniques take advantage of the GPU rasterizing capabilities to compute the first intersection point with the primary rays. In order to trace secondary rays on the GPU, we should ensure that GPU processors have access to the scene geometry. If the GPU is controlled by a graphics API, vertex, geometry, and fragment shaders may access global (i.e. uniform) parameters and textures, thus this requirement can be met by storing the scene geometry in uniform parameters or in textures [SKSS08].

In this STAR report we are going to establish a taxonomy of methods to compute real-time specular effects on the GPU, based on the different ways of representing the scene and handling ray intersections with the aid of the GPU:

- *Ray tracing in the original geometry* where traditional, object space accurate ray tracing algorithms are adapted to work on the GPU, which is possible due to the high parallelism inherent to these computations.
- *Image based lighting and rendering* where a part of the scene is represented by images, which are looked up with the parameters of the ray.
- *Ray tracing in the sampled geometry* where the scene is sampled in such a way that it is feasible to store samples in textures and compute the intersection of a ray with reasonably few texture fetches.
- *Geometry transformation* where the geometry of the scene is transformed to build the virtual scene that is seen on the reflector.

We also provide a classification for *caustic* generating algorithms, based on how the caustic hits are identified, how the photon hits are stored, and how they are filtered and projected onto the receiver surfaces.

2. Ray tracing in the original geometry

Ray tracing is a classical image synthesis approach, which offers accurate non-local lighting phenomena, including specular effects. For a long time, it was synonymous to high-quality off-line rendering as opposed to real-time incremental graphics with an artificial appearance. With evolving GPUs the gap has been closing, and both the need and the possibility for high fidelity ray tracing algorithms in real-time graphics emerged. Reformulating the wealth of earlier results for parallel architectures has remained an important direction of research.

Efficient implementations of ray tracing algorithms that work with the original geometry need fast ray–object intersection calculation and complex data structures partitioning space to quickly exclude objects that are surely not intersected by a ray. Many data structures and algorithms have been developed having a CPU implementation in mind [AW87, Arv91, AK87, FTK86, Gla89, OM87, Hav01, WKB*02, RSH05, HHS06]. GPU ray tracing have benefited from the advancements in accelerating CPU ray tracing, but not all techniques run equally well on both platforms. Most ideas had to be adapted to account for the special nature of GPUs.

Object precision ray tracing is a data parallel algorithm, which is fundamentally different from the incremental rendering pipeline. In order to leverage the GPU's computational power for this task, one possibility is to view the incremental rendering pipeline as a general purpose stream processing device. This allows algorithms and data structures originally proposed for CPU ray tracers to be ported to vertex, geometry, and fragment shader programs. Still, peculiarities of the rendering pipeline and features like the z-buffer can be exploited. The other possibility is the application of the CUDA environment, which departs from the incremental pipeline and allows access to the widest range of stream processor features. CUDA has been used in the latest successful implementations, not only for actual ray tracing but also for the construction of the acceleration hierarchy.

2.1. Ray tracing primitives

One of the striking advantages of the ray tracing approach is that primitives more complex than triangles are easily supported. To name a few recent examples, algorithms for tracing piecewise quadratic [SGS06] and NURBS surfaces [PSS*06] have been proposed. These algorithms use the incremental pipeline to splat enclosing objects onto the render target image and perform ray intersection in fragment shaders. Therefore, they assume coherent eye rays are to be traced, which makes these approaches not yet suitable to trace reflection and refraction rays, which would be required for specular effects.

In general, considering scenes of typical size in interactive applications, surface points visible through pixels are more effectively determined using incremental techniques than ray tracing. Thus, it is often required that the same geometry is rendered incrementally, and ray traced for specular effects. Therefore, ray tracing primitives can be assumed to be triangles in practice.

2.1.1. Ray–triangle intersection

The fundamental operation of ray tracing is evaluating the ray–triangle intersection test. Depending on the acceleration scheme, a high number of intersection tests may have to be performed, making it a time-critical operation. Möller and Trumbore [MT97] introduced a fast algorithm, which does not require preprocessing and is fairly popular in GPU programs. It solves the linear system of equations to obtain the barycentric coordinates of the intersection point. Applying pre-computation and passing other information instead of the triangle vertices, the test can be further speeded up [Wal04, SKSS06, Ken07].

The 4-channel ALUs of GPUs can also be exploited to compute intersections with four rays simultaneously [SSK07]. These methods work best with *ray packets* [WIK*06, HSHH07].

2.2. The ray engine

The *ray engine* [CHH02] method uses the rasterization hardware to initiate ray–triangle intersection tests, the fragment shader to evaluate these tests, and the z-buffer to find the nearest hit for a ray. It associates the set of rays with pixels and the objects with rasterization primitives. In order to guarantee that every object is tested with every ray, the viewport resolution is set to make the number of pixels equal to the number of rays, and an object is rendered as a *full-viewport rectangle*, i.e. a quadrilateral that covers all pixels of the viewport.

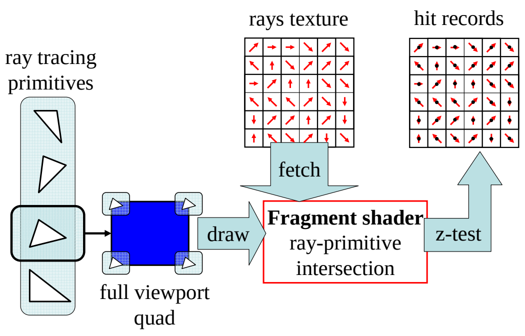

Figure 3: *Rendering pass implementing the ray engine. The fragment shader computes the ray–primitive intersection and outputs the hit information as the color and the ray parameter of the hit as the depth of the fragment. The z-test executed in the z-buffer will keep the first intersections in the resulting texture. Results are further processed by the CPU.*

Figure 3 depicts the rendering pass of the ray engine algorithm. Every pixel of the render target is associated with a ray. The origin and direction of rays to be traced are stored in textures that have the same dimensions as the render target. One after the other, a single primitive is taken, and it is rendered as a full-viewport rectangle, with the primitive data stored in the vertex records. Thus, pixel shaders for every pixel will receive the primitive data, and can also access the ray data via texture reads. The ray–primitive intersection calculation can be performed in the shader. Then, using the distance of the intersection as a depth value, a depth test is performed to verify that no closer intersection has been found yet. If the result passes the z-test, it is written to the render target and the depth buffer is updated. This way every pixel will hold the information about the nearest intersection between the scene primitives and the ray associated with the pixel.

Sending every object as a full-viewport rectangle tests every ray against every primitive, thus the CPU program is expected to provide only those sets of rays and objects which may potentially intersect. In this sense the ray engine can be imagined as a co-processor to CPU based ray tracing. The CPU part of the implementation finds the potentially visible objects, and the GPU tries to intersect these objects with the set of specified primary, shadow, or secondary rays.

This concept of using the GPU as a co-processor for a CPU based rendering algorithm is also exploited in final rendering software *Gelato*. However, such approaches require frequent communication of results from the GPU to the CPU.

2.2.1. Acceleration hierarchy built on rays

The CPU–GPU communication bottleneck can be avoided if the GPU not only computes the ray–object intersections but also takes the responsibility of the ray generation and the construction of the acceleration scheme to decrease the number of intersection tests. Acceleration schemes are usually spatial object hierarchies. The basic approach is that, for a ray, we try to exclude as many objects as possible from intersection testing. This cannot be done in the ray engine architecture, as it follows a per primitive processing scheme instead of the per ray philosophy. Therefore, it is also worth building an acceleration hierarchy over the rays, not over the objects [Sze06].

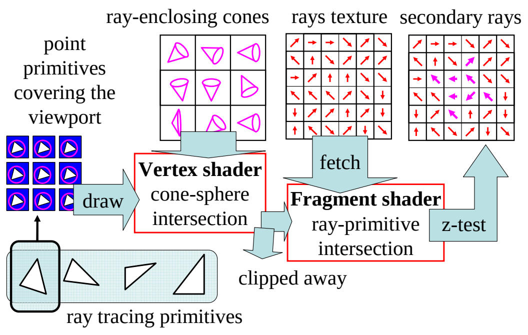

Figure 4: *Point primitives are rendered instead of full viewport quads, to decompose the array of rays into tiles. The vertex shader tests the enclosing objects of rays and primitives against each other, and discards tiles with no possible intersections. Refracted or reflected secondary rays are generated at the hit points. These can be processed on the GPU in a similar pass, without reading the data back to the CPU.*

Primary ray impact points are determined by rendering the scene from either the eye or the light. As nearby rays hit similar surfaces, it can be assumed that reflected or refracted rays may also travel in similar directions, albeit with more and more deviation on multiple iterations. If we compute enclosing objects for groups of nearby rays, it becomes possible to exclude all rays within a group based on a single test against the primitive being processed. Whenever the data of a primitive is processed, we do not render a rectangle covering the entire viewport, but invoke the pixel shaders only where an intersection is possible. The solution (as illustrated in Figure 4) is to split the render target into tiles, render a set of tile quads instead of a full viewport one, but make a decision for every tile beforehand whether it should be rendered at all. The enclosing objects for rays grouped in the same tile are infinite cones [Ama84]. By testing them against the enclosing sphere of each triangle, tiles not containing any intersections are excluded. The computation of ray-enclosing cones is also performed on the GPU.

Figure 5 shows a scene of multiple refractive objects in a cube map environment. The rendering resolution is 512×512, divided into 32×32 tiles, all rendered as 16×16 sized DirectX point primitives at 40 FPS on an NV6800 GPU.

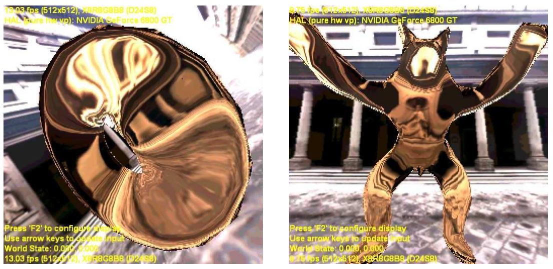

Figure 5: *Images rendered using an acceleration hierarchy of rays [Sze06].*

Roger et al. [RAH07] extended the ray hierarchy approach to a complete ray tracer and examined the exploitation of the geometry shader of Shader Model 4 GPUs. As has been pointed out in this paper, a particular advantage of the ray hierarchy method is that the construction cost of the acceleration scheme is low, thus the method is a primary candidate for dynamic scenes.

2.3. Spatial acceleration schemes

As GPUs were advancing, the GPU implementation of more and more complex acceleration schemes became possible. Additionally, parallel algorithms for the construction of these data structures have also been proposed, making them applicable for real-time rendering of dynamic scenes. A detailed state-of-the-art analysis focusing on the construction and traversal of such hierarchies for dynamic scenes (considering both CPU and GPU methods) has been given in [WMG*07].

Algorithms that further increase the performance of acceleration schemes by exploiting ray coherence, including ray packets and frustum traversal, have also been adapted to the GPU. Combined with the SIMD execution scheme of GPU ALUs, these offer significant speedups. It has to be noted, however, that specular effects require tracing secondary reflection and refraction rays, which exhibit far less coherence than primary rays, so the gain in our context is likely to be less significant.

In the following subsections, we summarize the research results for all popular acceleration schemes from the basics to most recent improvements on traversal and construction.

2.3.1. Uniform grid

The uniform grid partitions the bounding box of the scene to equal size cells (Figure 6) independently of the distribution of the objects. The advantage of this scheme is that a cell can be located in constant time and cells pierced by the ray can be visited with an incremental 3D line drawing algorithm. The disadvantage is that many cells may be empty, and empty cells also cost storage space and processing time during traversal.

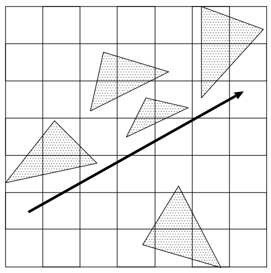

Figure 6: *A uniform grid [FTK86, AK89] that partitions the bounding box of the scene into cells of uniform size. In each cell objects whose bounding box overlaps with the cell are registered.*

To cope with the limited features of GPUs of that time, Purcell et al. [PBMH02, PDC*03] applied uniform grid partitioning and decomposed ray tracing into four separate GPU kernels: traversal, intersection, shading, and spawning, and associated each kernel with a separate rendering pass. This decomposition resulted in simple shader programs having no dynamic branching. The state information communicated by the passes is stored in floating point textures. The input stream and execution of all the fragment programs are initialized by rasterizing a viewport sized quadrilateral.

Even this early method revealed the fundamental challenge of GPU space partitioning: *traversal coherence*. For different rays, the above kernel operations have to be performed in varying number and order. It is often the case that processors that have already finished their tasks executing one type of kernel have to wait for others before they can go on to the next state.

The uniform 3D grid (Figure 6) partitioning of the scene geometry is stored in a 3D texture, which is cache-coherent and accessed in constant time. A grid cell is encoded by a texel storing the the number of referenced objects in one color channel, and a pointer to the first referenced object in the list of objects. This pointer is in fact two floats interpreted as a texture address. The list of referenced objects is stored in another texture where each texel corresponds to a pointer (i.e. texture address) to the actual object data, including vertex position, normals, texture coordinates or other material information.

Karlsson proposed the inclusion of the distance to the nearest non-empty cell in the texel of the 3D grid. This information can be used for skipping empty cells during traversing the grid [KL04].

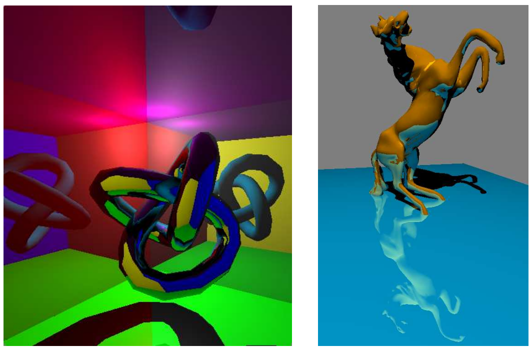

Figure 7: *Reflective objects rendered by Purcell's method at 10 FPS on* 512×512 *resolution using an NV6800 GPU [Ral07].*

Wald et al. [WIK*06] have extended the grid traversal scheme to handle ray packets, thus exploiting ray coherence. Grid cells that cover the frustum of rays are found for slices of voxels in the 3D grid, and rays in the packet are tested against objects in those grid cells. The authors' conjecture is that the method is very appropriate for GPUs, but this is yet to be proven. The dynamic, parallel construction of uniform grid schemes has been addressed in [IWRP06]. This paper targeted multi-core CPU systems, and a streaming GPU version has not been published yet.

2.3.2. Octree

For octrees (Figure 8), the cells can be bigger than the minimum cell, which saves empty space traversal time.

Ray tracing with octree has been transported to GPU by Meseth et al. [MGK05]. The hierarchical data structure is

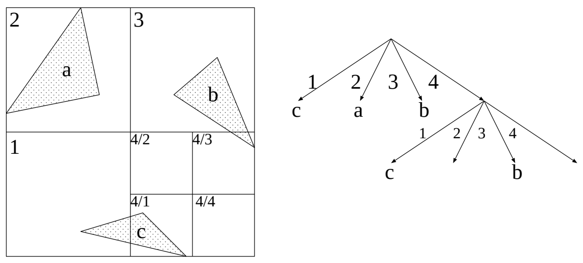

Figure 8: *A quadtree partitioning the plane, whose three-dimensional version is the octree [Gla84, AK89]. The tree is constructed by halving the cells along all coordinate axes until a cell contains "just a few" objects, or the cell size gets smaller than a threshold. Objects are registered in the leaves of the tree.*

mapped to a texture, where values are interpreted as pointers. Chasing those pointers is realized using dependent texture reads. During traversal, similarly to octrees managed by CPUs, the next cell along a ray is identified by translating the exit point from the current cell a little along the ray, and performing a top-down containment test from the root of the tree. In order to avoid processing lists of triangles in cells, every cell only contains a single volumetric approximation of visual properties within the cell. Thus, scene geometry needs to be transformed into a volumetric representation. Although the method makes use of spatial subdivision, it is in fact a clever approximate ray tracing scheme over a discretely sampled scene representation.

2.3.3. Kd-tree

Kd-trees adaptively and recursively subdivide the cells containing many objects. Unlike in octrees, the distribution of the objects inside the cell is also considered when searching for a subdividing plane (Figure 9).

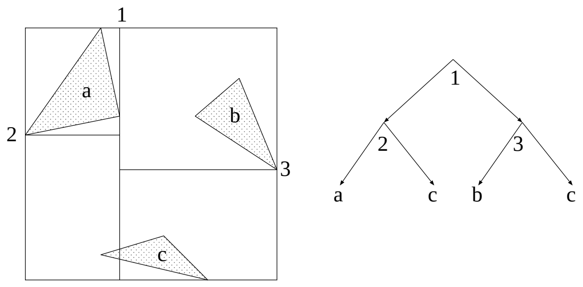

Figure 9: *A kd-tree. Cells containing "many" objects are recursively subdivided to two cells with a plane that is perpendicular to one of the coordinate axes. Note that a triangle may belong to more than one cell.*

Foley and Sugerman [FS05] implemented the kd-tree acceleration hierarchy on the GPU. Their algorithm was not optimal in algorithmic sense [Hav01, HP03, SKHBS02] but eliminated the need of a stack, which is very limited on Shader Model 1, 2, and 3 GPUs [EVG04, PGSS07]. While this decomposition showed that GPU hierarchy traversal is feasible, it achieved only 10% of the performance of comparable CPU implementations. They implemented two variations on a stackless traversal of a kd-tree: *kd-restart* that iterates down the kd-tree, and *kd-backtrack* that finds the parent of the next traversal node by following parent pointers up the tree and comparing current traversal progress with per-node bounding boxes. The kd-tree had to be constructed on the CPU.

Popov et al. [PGSS07] proposed a kd-tree structure augmented with additional pointers, called *ropes*, connecting neighboring cells to enable the identification of the next cell along the ray without the use of a stack. They used the CUDA architecture and achieved real-time ray tracing while supporting specular effects.

In [HSHH07] Foley and Sugerman's approach has been extended and turned to a single pass algorithm using current GPUs' branching and looping abilities. This paper additionally introduced three optimizations: a packeted formulation, a technique for restarting partially down the tree instead of at the root, and a small, fixed-size stack that is checked before resorting to restart. We can say that the classical recursive kd-tree traversal algorithm of CPUs has been essentially reproduced on the GPU with the short-stack approach (for reasonable scenes the stack can be long enough to support the entire depth of the kd-tree), and packetization opened the door to coherence-exploiting methods.

Classically, kd-tree construction is a recursive, depth-first search type algorithm, where triangles in a cell must be sorted according to one of their coordinates, then possible splitting planes are evaluated by scanning the array. The *surface area heuristics* (SAH) is the mostly used estimate for the traversal cost [Hav01]. The splitting position with the lowest cost is selected, and triangles are sorted into the child cells, for which the same process is repeated. This method requires expensive sorting and recursive execution, which are not feasible in real time and on the GPU. In [PGSS06] an approximate approach has been proposed that evaluates only a fixed number of evenly distributed possible splitting positions without sorting. With that, kd-tree construction was possible on the GPU in a depth-first search manner, but on the lowest levels it has to fall back to the classical algorithm to get acceptable traversal performance, and for complex scenes the construction time was still too high.

In [ZHWG08] the kd-tree approach has been shown to be applicable to dynamic scenes through a real-time parallel construction algorithm implemented in CUDA. Further rationalizing the idea of [PGSS06], on higher levels of the tree only median splitting and cutting-off-empty-space (Figure 10) are considered. On lower levels with only a few triangles, the extrema of triangles are the considered candidates, which can be evaluated without sorting. That way, the kd-tree structure is still close to optimal, but can be constructed in real-time. A photon mapping application is presented which is capable of running interactively for dynamic scenes.

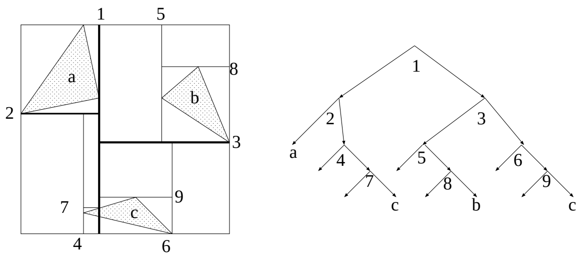

Figure 10: *A kd-tree cutting away empty spaces. Note that empty leaves are possible.*

2.3.4. Bounding kd-tree variants

In conventional kd-trees a triangle may belong to both child cells of a supercell, and the triangle lists for the two leaves together may be longer than the original list, complicating and slowing down the building process. Triangle bounding boxes typically overlap, and it is neither always possible nor desirable to find cutting planes that do not intersect any triangles. During traversal, these triangles will be tested for intersection more than once. Shared triangles either need to be clipped, or extra tests are necessary to assure that a hit actually happened within the cell we are processing.

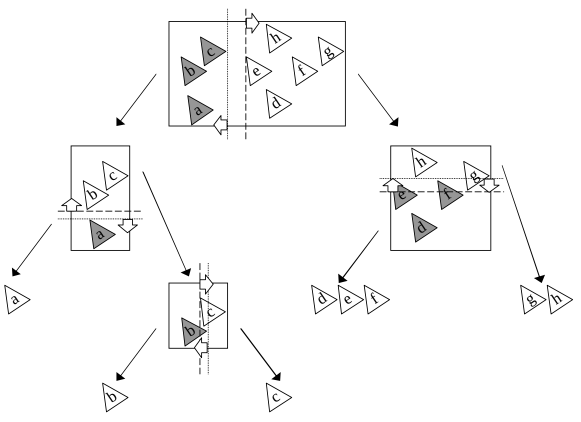

Figure 11: *Structure of an skd-tree. A cell is cut into two possibly overlapping cells by two planes. Every triangle belongs strictly to one cell that completely contains it.*

All these problems, along with the rapid reconstruction for dynamic scenes, are elegantly addressed by *bounding interval hierarchies* [WMS06], which have been proposed for ray tracing on a dedicated hardware architecture. In fact, the concept was first published as *spatial kd-trees* or *skd-trees* in [OMSD87], and this (somewhat less descriptive) name has been used more prominently in recent publications.

Similar to loose octrees [Tha00], skd-trees are in fact loose kd-trees, where child cells may be overlapping, but they completely contain the primitives sorted into them. In skd-trees, instead of cutting the cell into two non-overlapping cells sharing some triangles, we decompose it to two possibly overlapping cells, where every triangle belongs strictly to one cell, namely the one that completely contains it (Figure 11). The price is that we have to store the position for two cutting planes: the maximum of the left cell and the minimum of the right cell. Additionally, the minima and the maxima of both cells can be stored, resulting in tighter bounding volumes. In kd-tree construction parlance, this amounts to cutting off empty space. During traversal, we need to compute intersections with all planes, but on a GPU with four-channel ALUs this comes for free if ray packeting is not used.

In [SKSS06] the skd-tree traversal has been implemented on general graphics hardware in a single pass, using a stack traversal algorithm (Figure 12).

In [HHS06] several variants have been proposed to combine skd-trees with bounding volume hierarchies, and fast construction algorithms have been proposed. However, these are not targeted at GPUs.

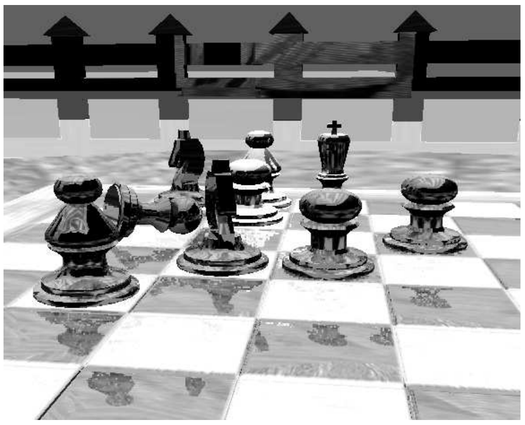

Figure 12: *A scene rendered with textures and reflections (15000 triangles, 10 FPS at 512×512 resolution on an NV8800 GPU) [SKSS06].*

2.3.5. Hierarchical bounding boxes

Geometry images [GGH02] can be used for fast bounding box hierarchy [TS05] construction as has been pointed out in [CHCH06]. Here a threaded bounding box hierarchy was used which does not rely on conditional execution (another feature that is poorly supported by the GPU) to determine the next node in a traversal. The method threads a bounding box hierarchy with a pair of pointers per node, indicating the next node to consider, given that the ray either intersects or misses the node's bounding box. These threads allow the GPU to stream through the hierarchy without maintaining a stack. As there are no conditional operations, the utilization of the parallel GPU processors are high, but the resulting traversal order might not be optimal, which decreases performance.

Other efficient hierarchical bounding box approaches have been proposed in [Chr05, BWSF06]. Günther et al. [GPSS07] exploited the CUDA programming environment that is more appropriate for general purpose computation like ray tracing in the original geometry than the incremental rendering pipeline.

3. Environment mapping

Environment mapping [BN76] is an old approximation technique that substitutes ray tracing by looking up values from a map. The steps of environment mapping are shown by Figure 13.

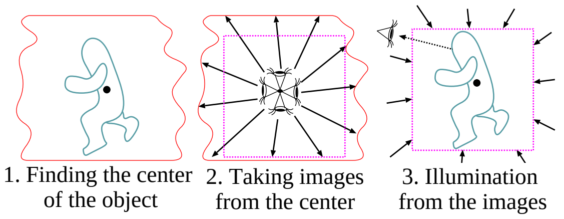
1. Finding the center of the object
2. Taking images from the center
3. Illumination from the images

Figure 13: *Steps of environment mapping*

The environment of a reflective object is rendered from a point, called *reference point*, that is in the vicinity of the reflector, defining the camera windows as six faces of a cube. Then, during the rendering of the reflective object, the radiance of the reflected ray is looked up from these images using only the direction of the ray but ignoring its origin and neglecting self reflections. In other words, environment mapping assumes that the reflected points are very (infinitely) far, and thus the hit points of the rays become independent of the reflection points, i.e. the ray origins.

Environment mapping has been originally proposed to render ideal mirrors in local illumination frameworks, then has been extended to approximate general secondary rays without expensive ray-tracing [Gre84, RTJ94, Wil01]. The idea can also be used for glossy and diffuse materials if the map is pre-filtered, i.e. it is convolved with the angular variation of the product of the BRDF and the cosine of the incident angle [KVHS00, KM00, RH01, HSC*05]. Environment mapping has also become a standard technique of *image based lighting* [MH84, Deb98].

A fundamental problem of environment mapping is that the environment map is the correct representation of the direction dependent illumination only at its reference point. For other points, accurate results can only be expected if the distance of the point of interest from the reference point is negligible compared to the distance from the surrounding geometry (Figure 14). Classical environment mapping cannot present correct *parallax* when the eye point moves, which makes the user feel that the reflective object is not properly embedded in its environment. However, if the object moves, the environment map can be re-generated which provides *motion parallax*.

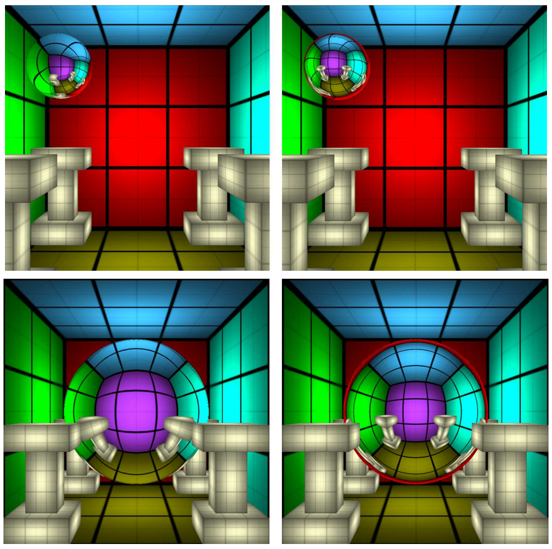
Figure 14: *Ray tracing (left) compared to environment mapping having the reference point in the middle of the room (right). Note that environment mapping becomes very inaccurate if the reflective surface is far from the reference point and is close to the reflected environment.*

In order to increase the accuracy of environment mapping, Cabral et al. [CON99] and Hakura et al. [HS01] used multiple environment maps, and for each shaded point they looked up that map where the shaded point is the closest to the reference point of the environment map. However, the transitions between different environment maps are difficult to control and may lead to discontinuities. One elegant way to solve the discontinuity problem is to approximate the hit point by tracing the ray in a proxy geometry, to look up the hit point from the environment maps having centers closest to the hit point, and blending the colors based on the distances [ML03]. In this paper the proxy geometry was the height map of an urban model.

3.1. Environment mapping on the GPU

GPUs can be effectively used in environment map generation and rendering reflective objects. Environment map generation is a conventional rendering pass where the rasterization hardware and the z-buffer can easily identify points that are visible from the reference point through the faces of a cube.

When the reflective or refractive object is rendered, the fragment shader computes the reflection or refraction direction from the view direction and the interpolated normal, and looks up the prepared cube map texture with these directions to get the incident radiance. The incident radiance is weighted with the Fresnel function according to equation 1.

3.2. Localized environment mapping

In order to provide more accurate environment reflections and the missing parallax effect, the geometry of the scene should be taken into account, and the ray tracing process should be executed or at least approximated. A simple approximation is to use some *proxy geometry* [Bre02, Bjo04] (e.g. a sphere, a billboard rectangle or a cube) of the environment, which is intersected by the reflection ray to obtain a point, whose direction is used in the environment map lookup (Figure 15). For a fixed and simple proxy geometry, the ray intersection calculation can be executed by the pixel shader of the GPU. However, the assumption of a simple and constant environment geometry creates visible artifacts that make the proxy geometry apparent during animation. Hargreaves [Har04] applied further simplifications and assumed that the environment is an infinite hemisphere, the object is always at a fixed distance from the base plane of the hemisphere, and reflection rays are always perpendicular to the base plane. These assumptions make even ray tracing unnecessary, but significantly limit the applicability of the algorithm (the target application was a bike racing game).

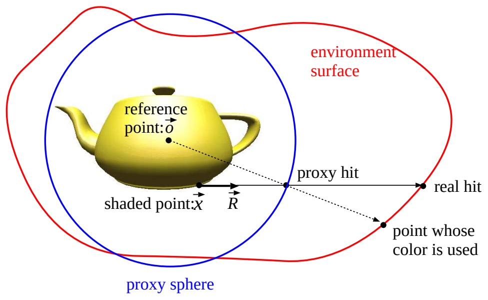

Figure 15: *Application of a proxy sphere when tracing reflection ray of origin $\vec{x}$ and direction $\vec{R}$. The environment map is looked up in the direction of the proxy hit.*

Popescu et. al [PMDS06] proposed a method that used displacement mapped rectangles as the proxy geometry.

We note that environment map localization efforts occurred even in the area of CPU ray tracing. Reinhard [RTJ94] proposed the reflection rays to be traced only in the neighborhood (i.e. inside a cube, which separates the surfaces handled explicitly from those of replaced by the environment map). In this method, environment mapping is only used if ray tracing in the neighborhood reports no intersection.

3.3. Image based rendering

Classical environment mapping looks up the environment map with the ray direction but ignoring the ray origin. Localization means the incorporation of the ray origin into this process. Making images dependent on the viewer's position is also important in *image based rendering* [LH96]. Image based rendering can be explained as combining and warping images taken from different camera positions in order to get the image that could be seen from a new camera position. The reconstruction may be based on coarse geometric information [GGSC96, DBY98] or on per pixel depth values [PCD*97b, SMS01, ESK03]. Lischinski and Rappoport [LR98] used layered light fields to render fuzzy reflections on glossy objects and layered depth images to ray trace sharp reflections. Heidrich et al. [HLCS99] and Yu et al. [YYM05] took two light fields to simulate accurate reflections, one for the radiance of the environment, and another for mapping viewing rays striking the object to outgoing reflection rays.

Pre-computing refractive paths is also strongly related to image based rendering. Having these paths, real-time effects can be generated easily [HLL07]. Refracted paths can be stored in a compact form using, for example, spherical harmonics [GLD06].

4. Ray tracing in the sampled geometry

Instead of storing the original geometry, the scene can also be represented in a sampled form in textures. A conventional texture map encodes a color (i.e. radiance) function over a 2D surface in the 3D space, thus a single texel represents a point of a 2D surface, and the texture coordinate pair navigates in this 2D subset. In order to allow the representation of points of 3D point sets, we can either use 3D textures or store additional coordinates in the texture data that complements the texture coordinate pair. One alternative is to store all three coordinates in the texture data, which is the basic idea of *geometry images* [GGH02]. On the other hand, if the texture coordinate pair is also directly utilized to identify the represented point, a single additional coordinate is enough. This "third coordinate" can be a distance from a reference surface, when it is called the "height", or from a reference point, when it is called the "depth".

Adding a per-texel depth or height to smooth surfaces is also essential in *displacement mapping* [PHL91, Don05, HEGD04, SKU08] and in the realistic rendering of *billboards* [Sch97, PMDS06, MJW07].

The oldest application of the idea of using distances from a given point as a sampled representation of the scene is the *depth map* of shadow algorithms [Wil78]. A shadow map samples just that part of the geometry which is seen from the light position through a window. To consider all directions, a *cube map* should be used instead of a single depth image. Storing modeling space Euclidean distance in texels instead of the screen space z coordinates [Pat95, SKALP05], the cube map becomes similar to a sphere map, which has the advantage that cube map boundaries need not be handled in a special way when interpolating between two directions. These cube maps are called *distance maps*.

The depth map generated from the camera — i.e. assuming that the reference point is the eye — is also sufficient in many applications if rays leaving the visible frustum can be ignored [Wym05b, WD06b, SP07].

Sampling can be used simultaneously for color and geometry information. The computation of these maps is very similar to that of classical environment maps. The only difference is that not only the color, but the distance is also calculated and stored in additional maps.

A major limitation of single layer maps is that they represent just those surfaces in these maps which are directly visible from the reference point, thus those ray hits which are occluded from the reference point cannot be accurately computed. If occluded surfaces are also needed, a texel should store multiple distances or, alternatively, we should use multiple layers as suggested by *layered depth impostors* [DSSD99]. The set of layers representing the scene could be obtained by *depth peeling* [Eve01, LWX06, Thi08] in the general case. However, we may often apply an approximation that limits the number of layers to two, representing front faces and back faces, respectively.

Instead of using multiple or layered cube maps, another solution of the occlusion problem is the application of *multi-perspective* or *non-pin-hole* cameras [HWSG06, MPS05]. In the extreme case, a multiperspective camera is assigned to each object, which provides a dense sampling of the space of rays intersecting this object. The ray space is five dimensional, which can be reduced to four dimensions if rays start far from the object. In this case the ray space can be parameterized by two points on a bounding sphere or on a cube. Having the ray samples, the intersections are computed during preprocessing and the results are stored in a high-dimensional texture. In the GPU such a high-dimensional texture should be "flattened", i.e. stored as a tiled 2D texture. During real-time processing a ray is traced by finding the closest sample rays and interpolating the hits, which is equivalent to a texture lookup [LFY*06]. The problems are that high dimensional textures require a lot of storage space and this approach works only for rigid objects. We note that this idea was used first in displacement mapping [WWT*04].

4.1. Ray tracing distance maps

In this section the ray tracing algorithm in a single distance map layer is discussed, using the notations of Figure 16. Let us assume that center $\vec{o}$ of our coordinate system is the reference point and we are interested in the illumination of point $\vec{x}$ from direction $\vec{R}$.

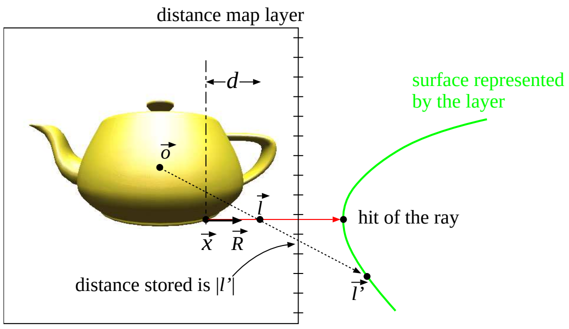

Figure 16: *Tracing a ray from reflection point $\vec{x}$ at direction $\vec{R}$. When we have a hit point approximation $\vec{l}$ on the ray, the distance and the radiance of point $\vec{l}'$ will be fetched from the cube map of reference point $\vec{o}$.*

The illuminating point is thus on the ray of equation $\vec{x}+\vec{R}d$, where d is the ray parameter. The accuracy of an arbitrary approximation d can be checked by reading the distance of the environment surface stored with the direction of $\vec{l}=\vec{x}+\vec{R}d$ in the cube map ($|\vec{l}'|$) and comparing it with the distance of approximating point $\vec{l}$ on the ray ($|\vec{l}|$). If $|\vec{l}|\approx|\vec{l}'|$, then we have found the intersection. If the point on the ray is in front of the surface, that is $|\vec{l}|<|\vec{l}'|$, the current approximation is an *undershooting*. On the other hand, the case when point $\vec{l}$ is behind the surface ($|\vec{l}|>|\vec{l}'|$) is called *overshooting*. Ray parameter d of the ray hit can be found by a simple approximation or by an iterative process.

4.1.1. Parallax correction

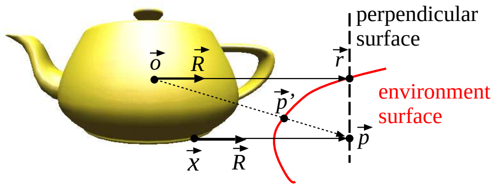

Figure 17: *Parallax correction assumes that the surface is perpendicular to the ray and replaces approximation $\vec{r}$ of the environment mapping by the intersection of the ray of origin $\vec{x}$ and direction $\vec{R}$ with this perpendicular surface.*

Classical environment mapping would look up the illumination selected by direction $\vec{R}$, that is, it would use the radiance of point $\vec{r}$ (Figure 17). This can be considered as the first guess for the ray hit. To find a better second guess, we assume that the environment surface at $\vec{r}$ is perpendicular to ray direction $\vec{R}$. In case of perpendicular surface, the ray would hit point $\vec{p}$ with ray parameter d_p:

$$d_p=|\vec{r}|-\vec{R}\cdot\vec{x}. \tag{4}$$

If we used the direction of point $\vec{p}$ to lookup the environment map, we would obtain the radiance of point $\vec{p}'$, which is in the direction of $\vec{p}$ but is on the surface. If the accuracy is not sufficient, the same step can be repeated, resulting in an iterative process [Wym05b, WD06b, SP07].

4.1.2. Linear search

The possible intersection points are on the half-line of the ray, thus the intersection can be found by marching on the ray, i.e. checking points $\vec{l}=\vec{x}+\vec{R}d$ generated with an increasing sequence of parameter d and detecting the first pair of subsequent points where one point is an overshooting while the other is an undershooting [Pat95].

Linear search by rasterizing a line segment

The complete search can be implemented by drawing a line and letting the fragment program check just one texel [KBW06]. However, in this case it is problematic to find the first intersection from the multiple intersections since fragments are processed independently. In their implementation, Krüger et al. solved this problem by rendering each ray into a separate row of a texture, and found the first hits by additional texture processing passes, which complicates the algorithm, reduces the flexibility, and prohibits early ray termination.

Linear search in a single fragment shader program

On GPUs supporting dynamic branching, the complete ray marching process can also be executed by a single fragment program [UPSK07]. This program can not only identify the first pair of overshooting and undershooting samples, but can also refine the hit point by a secant or a binary search.

The definition of the increments of ray parameter d needs special consideration because now the geometry is sampled and it is not worth checking the same sample many times while ignoring other samples. Unfortunately, making uniform steps on the ray does not guarantee that the texture space is uniformly sampled. As we get farther from the reference point, unit-length steps on the ray correspond to smaller steps in texture space. This problem can be solved by marching on line segment $\vec{r}'(t)=\vec{s}(1-t)+\vec{e}t$ where $\vec{s}$ and $\vec{e}$ are the projections of the start of the ray ($\vec{r}(0)$) and the end of the ray ($\vec{r}(\infty)$) onto a unit sphere, respectively.

The correspondence between ray parameter d and the parameter of the line segment t is:

$$d(t)=\frac{|\vec{x}|}{|\vec{R}|}\frac{t}{1-t}.$$

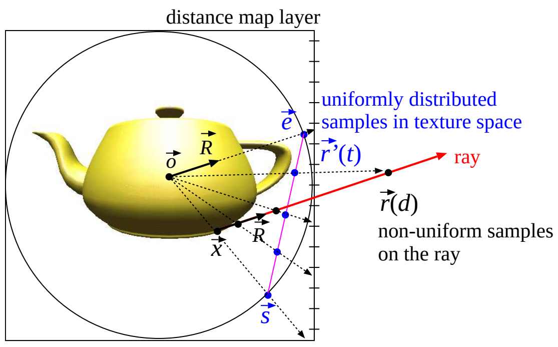

Figure 18: *Cube map texels that need to be visited when marching on ray $\vec{r}(d) = \vec{x} + \vec{R}d$ can be obtained either by non-uniform marching on ray $\vec{r}(d)$ or uniform marching on line segment $\vec{r}'(t)$ of projected points $\vec{s}$ and $\vec{e}$.*

4.1.3. Secant search

Secant search can be started when there are already two guesses of the intersection point, provided, for example, by a linear search [UPSK07], by taking the start and the end of the ray [SKAL05], or by pairing the end of the ray with the result of the parallax correction [SKALP05]. Let us denote the ray parameters of the two guesses by d_p and d_l, respectively. The corresponding two points on the ray are $\vec{p}$ and $\vec{l}$, and the two points on the surface are $\vec{p}'$ and $\vec{l}'$, respectively (Figure 19).

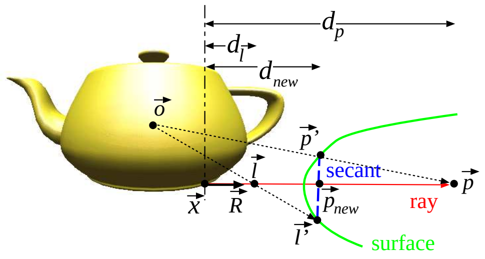

Figure 19: *Refinement by a secant step. The new approximation $\vec{p}_{new}$ is the intersection of the ray and the line segment of overshooting approximation $\vec{p}'$ and undershooting approximation $\vec{l}'$.*

Computing the ray parameter at the intersection of the ray and line segment $\vec{p}'$ and $\vec{l}'$, we obtain:

$$d_{new} = d_l + (d_p - d_l)\frac{1 - |\vec{l}|/|\vec{l}'|}{|\vec{p}|/|\vec{p}'| - |\vec{l}|/|\vec{l}'|}. \quad (5)$$

The point specified by this new ray parameter gets closer to the real intersection point. If a single secant step does not provide accurate enough results, then d_{new} can replace one of the previous approximations d_l or d_p, and we can proceed with the same iteration step. If we keep always one overshooting and one undershooting approximations, the method is equivalent to the false position root finding algorithm.

4.1.4. Binary search

Hu et al. [HQ07] and Oliveira et al. [OB07] used binary search steps to refine the approximations obtained by a linear search similarly to popular displacement mapping algorithms. Binary search simply halves the interval of the ray parameter:

$$d_{new} = \frac{d_l + d_p}{2}.$$

Since it does not use as much information as the secant search its convergence is slower.

Finally we note that two level search methods that start with a linear search and continue with either a binary or a secant search are also used in displacement mapping [POC05, BT04, SKU08] and in volume ray casting [HSS*05].

4.2. Single reflections or refractions using the sampled geometry

In order to render a scene with an object specularly reflecting its environment, we need to generate the depth or distance map of the environment of the specular object. This requires the rendering of all objects but the reflective object six times from the reference point, which is put close to the center of the reflective object. Then non-specular objects are rendered from the camera position in a normal way. Finally, the specular object is sent through the pipeline, setting the fragment shader to compute the reflected ray, and to approximate the hit point as a cube map texel. Having identified the texel corresponding to the hit point, its radiance is read from the cube map and is weighted by the Fresnel function.

Figure 20 compares images rendered by the distance map based ray tracing with standard environment mapping and classic ray tracing. Note that for such scenes where the environment is convex from the reference point of the environment map, and there are larger planar surfaces, the secant search algorithm converges very quickly. In fact, even the parallax correction is usually accurate enough, and iteration is needed only close to edges and corners.

Figure 21 shows a difficult case where a small box of chessboard pattern makes the environment surface concave and of high variation. Note that the convergence is still pretty fast, but the converged image is not exactly what we expect. We can observe that the green edge of the small box is visible in a larger portion of the reflection image. This phenomenon is due to the fact that a part of the wall is not visible from the reference point of the environment map, but is expected to show up in the reflection. In such cases the algorithm can go only to the edge of the box and substitutes the reflection of the occluded points by the blurred image of the edge.

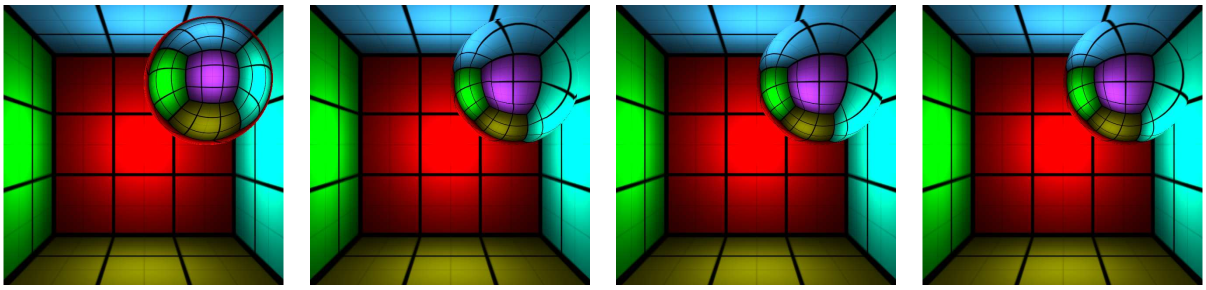

Figure 20: *Comparison of environment map reflections with distance map based and classical ray tracing, placing the reference point at the center of the room and moving a reflective sphere to the corner. The FPS values are measured with* 700×700 *resolution on an NV6800GT.*

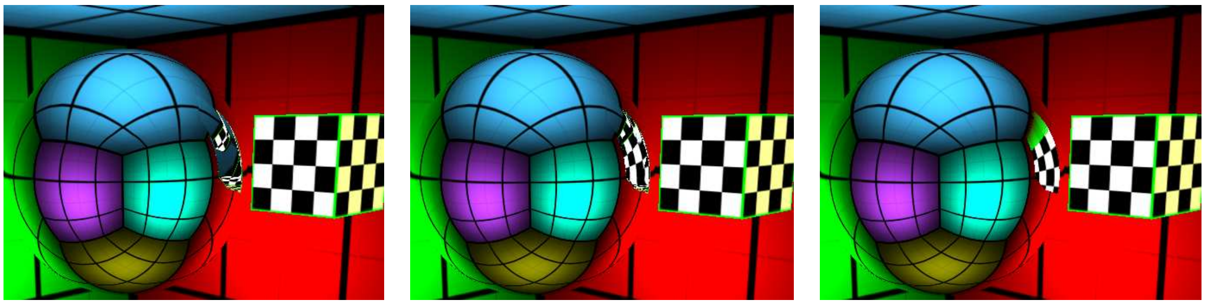

Figure 21: *A more difficult case when the room contains a box that makes the scene strongly concave and is responsible for view dependent occlusions. Note that the green edge of the box with chessboard pattern is visible in a larger portion of the reflection image, which replaces the reflection of the wall that is not visible from the reference point of the environment map, but is expected to show up in the reflection.*

The methods developed for reflections can also be used to simulate refracted rays, just the direction computation should be changed from the law of reflection to the Snellius-Descartes law of refraction.

4.3. Inter-object multiple reflections and refractions

Cube map based methods computing single reflections can straightforwardly be used to obtain multiple specular inter-reflections of different objects if each of them has its own cube map [NC02]. Suppose that the cube maps of specular objects are generated one by one. When the cube map of a particular object is generated, other objects are rendered with their own shader programs. A diffuse object is rendered with the reflected radiance of the direct light sources, and a specular object is processed by a fragment shader that looks up its cube map in the direction of the hit point of the reflection (or refraction) ray. When the first object is processed the cube maps of other objects are not yet initialized, so the cube map of the first object will be valid only where diffuse surfaces are visible. However, during the generation of the cube map for the second reflective object, the color reflected off the first object is already available, thus *diffuse surface – first reflective object – second reflective object* paths are correctly generated. At the end of the first round of the cube map generation process a later generated cube map will contain the reflection of other reflectors processed earlier, but not vice versa. Repeating the cube map generation process again, all cube maps will store double reflections and later rendered cube maps also represent triple reflections of earlier processed objects. Cube map generation cycles should be repeated until the required reflection depth is reached.

If we have a dynamic scene when cube maps are periodically re-generated anyway, the calculation of higher order reflections is not more expensive computationally than rendering single reflections. In each frame cube maps are updated using other objects' cube maps independently of whether they have already been refreshed in this frame or only in the previous frame (Figure 22). The reflection of a reflected image might come from the previous frame — i.e. the latency for degree *n* inter-reflection will be *n* frames — but this delay is not noticeable at interactive frame rates.

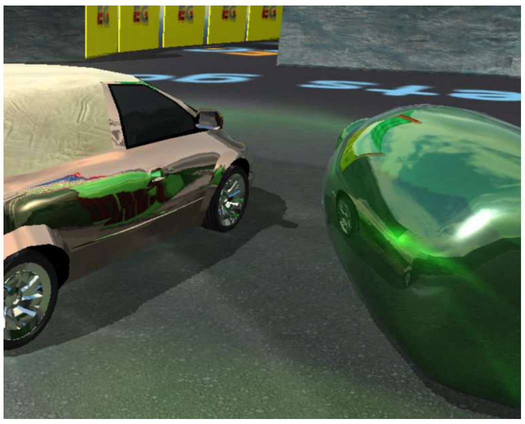

Figure 22: *Inter-object reflections in a car game. Note the reflection of the reflective car on the beer bottles, and vice versa.*

4.4. Self refractions

Light may get reflected or refracted on an ideal reflector or refractor several times. If the hit point of a ray is again on the specular surface, then reflected or refracted rays need to be computed and ray tracing should be continued, repeating the same algorithm recursively. Generally, the computation of the reflected or refracted ray requires the normal vector at the hit surface, the Fresnel function, and also the index of refraction in case of refractions, thus the hit point identification should also be associated with the indexing of these parameters.

However, the special case of *self refractions* is a little easier since if objects do not intersect, then the light must meet the surface of the same object again after entering it. It means that for homogeneous material objects, the material data is already available and the hit point should only be searched on the given refractive object. Self refraction computations can be further simplified by considering only *double refractions*, i.e. where the light enters and exits the object, and ignoring *total internal reflection* where the light would be unable to leave the inside of the object.

Wyman [Wym05a] proposed a front-face/back-face double refraction algorithm. During pre-computation this method calculates the distance of the back of the object at the direction of the normal vector at each vertex. The on-line part of rendering a refractive object consists of two passes. In the first pass, normals and depths of back facing surfaces are rendered into a texture. In the second pass front faces are drawn. The fragment shader reads the pre-computed distance in the direction of the normal and obtains the distance in the view direction from the texture, and linearly interpolates these distances according to the angle between the view and normal directions and the angle between the refracted and the normal directions. Translating the processed point by the distance obtained by this interpolation into the refraction direction, we get an approximation of the location of the second refraction. Projecting this point into the texture of rendered normal vectors, the normal vector of the second refraction is fetched and is used to compute the refracted direction of the second refraction.

To handle total internal reflection, Wyman's original algorithm has been improved by [DW07]. Recently Hu and Qin [HQ07] have developed an algorithm that is similar to Wyman's approach and uses a binary search to determine ray–object intersections with a depth map. Oliveira et al. [OB07] used both linear and binary search steps to find the intersection in a depth image.

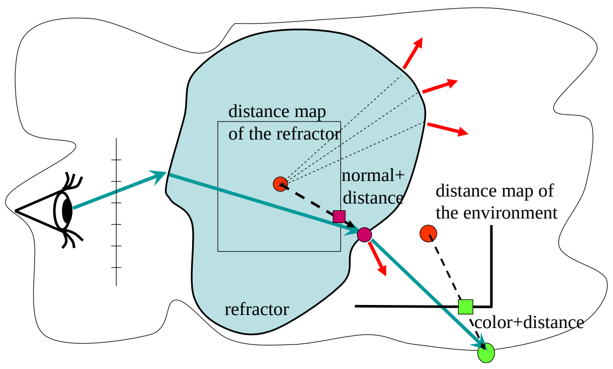

Figure 23: *Computation of multiple refractions on a single object storing the object's normals in one distance map and the color of the environment in another distance map.*

The approximate ray tracing algorithm [SKALP05] generates a cube map for each refracting object during the preprocessing phase. These refractor cube maps store the normal vector and the distance from the reference point. When the refractor is rendered from the camera, the fragment shader computes the refracted ray, finds its hit point with the sampled geometry of the refractor surface searching the cube map, and looks up the normal vector in the direction of the hit point from that map (Figures 23, 24, and 25). This method can also solve multiple reflections on the internal side of the surface if the refractor is not strongly concave, i.e. all surface points can be seen from its center point. Consequently, this method is appropriate for simulating total internal reflections. When the ray exits the object, another ray tracing phase is started that searches the distance cube map of the environment geometry.

Krüger et al. used layered depth maps generated from the

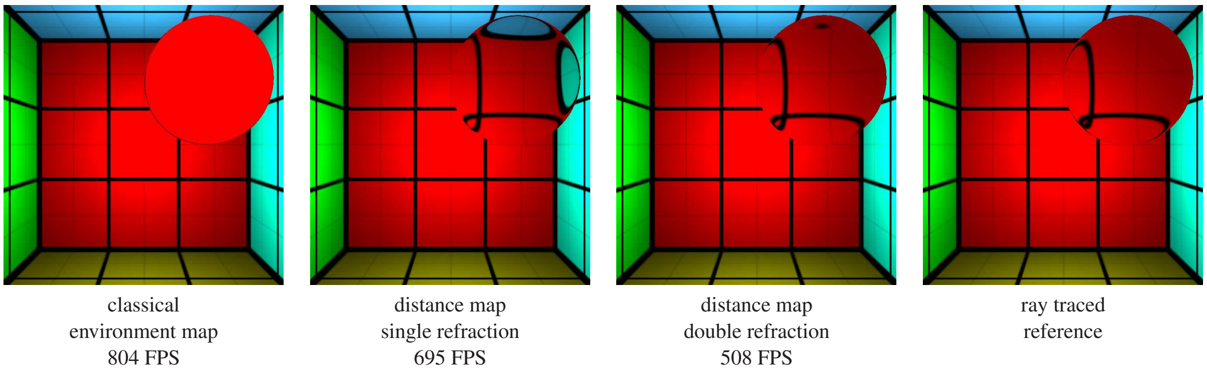

Figure 24: *Comparison of single and double refractions of a sphere having refraction index* $\nu = 1.1$*, computed with environment mapping, approximate ray tracing, and classical ray tracing. The FPS values are measured with* 700×700 *resolution on an NV6800GT.*

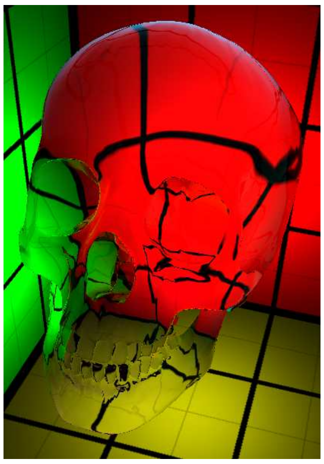

Figure 25: *Reflective and refractive glass skull* ($\nu = 1.3$) *of 61000 triangles rendered at 130 FPS on an NV6800 GT.*

camera to compute multiple refractions [KBW06], thus this method could handle more general objects than other methods.

4.5. General multiple reflections and refractions

Unfortunately, the methods of self refractions cannot be easily adapted to self reflections since the depth image seen from the surface of a concave reflective object may have a large variation and self occlusions are also likely to happen. Thus, distance map based methods may create artifacts. On the other hand, reflections are easier to predict by human observers than refractions, making accuracy more important. Finally, a ray refracting into a solid object surely hits the surface of the same object, which simplifies the search for this hit, but this is not the case for reflections.

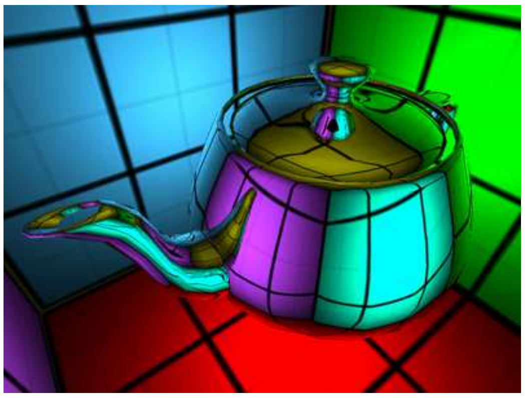

Figure 26: *Double reflections on a teapot.*

For self reflections we need to solve the general case and trace rays processing the sampled geometry multiple times once on each level of reflection. We cannot decompose the scene to the reflector and to its environment, but the complete geometry must be stored in distance maps. On the other hand, we cannot ignore those points that are occluded from the reference point, which means that a distance map texel should represent a set of points that are at the same direction from the reference point.

The method proposed in [UPSK07] uses layered distance maps and a combined linear and secant search to guarantee both speed and robustness (Figures 26, 27, and 28). A single layer of these layered distance maps is a cube map, where a texel contains the material properties, the distance, and the normal vector of the point that is in the texel's direction. The

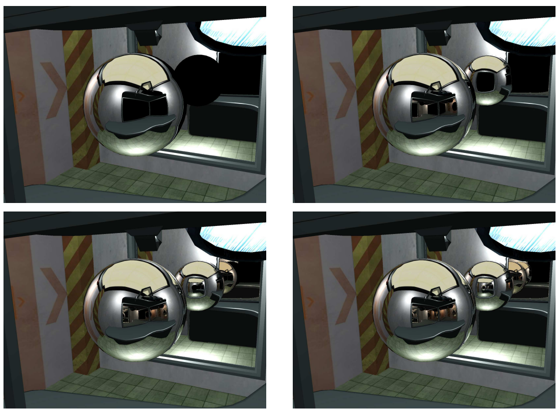

Figure 27: *Multiple reflections on a specular sphere and a mirror, incrementing the maximum recursion from one up to four.*

material property is the reflected radiance for diffuse surfaces and the Fresnel factor at perpendicular illumination for specular reflectors. For refractors, the index of refraction is also stored. With these data a complete recursive ray tracing algorithm can be implemented by a fragment shader, which computes intersections with the sampled geometry stored in cube maps.

5. Geometry Transformation

Searching the reflected point after finding the reflection point (Figure 2) is not the only way to render reflections, but searches can also be organized differently. For example, we can start at the reflected points, i.e. at the vertices of the reflected environment, and search for the reflection points, i.e. identify those points that reflect the input points. This means searching on or projecting onto the reflector surface rather than searching on the reflected environment surface.

The comparative advantages of searching on the reflected environment surface or searching on the reflector depend on the geometric properties of these surfaces. The algorithm that looks for the reflected point on the environment surface can handle arbitrary reflector surfaces and is particularly efficient if the environment surface does not have large distance variations. On the other hand, the algorithm that searches the reflection point on the reflector surface can cope with environment surfaces of large depth variations, but is limited to simple reflectors that are either concave or convex.

The first attempts focused on planar reflectors. *Multi-pass rendering* [DB97,MBGN00] used a modified projection matrix to project a reflected object through the plane of each reflector, and mapped its image onto the reflector plane. Reflections are implemented by first rendering the scene without the mirrored surfaces. A second rendering pass is performed for each reflected viewpoint, and the resulting image is applied to the reflector. This process can be repeated recursively in scenes with multiple reflectors. Stencil operations are arranged so the reflected images are masked by the stencil buffer. Pre-computed radiance maps [BHWL99] can be used to avoid the projection step.

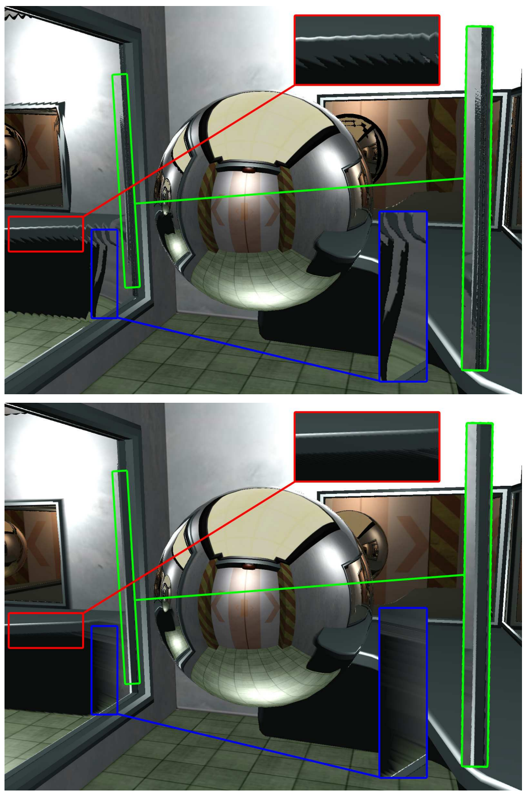

Figure 28: *Sampling artifacts caused by the low numbers of linear and secant search steps (upper image) and their elimination when they are increased (lower image). Thin objects shown in the green and red frames require fine linear steps. The aliasing of the reflection shown in the blue frame is caused by the limitation of distance map layers to three. Since this area is not represented in the layers, not even the secant search can compute accurate reflections.*

For curved reflectors, the situation is more complicated. An interesting solution was proposed by Ofek and Rappoport [OR98]. Their method warps the surrounding scene geometry such that it appears as a correct virtual image when drawn on the reflector. For each vertex to be reflected, an efficient data structure (the *explosion map*) accelerates the search for a triangle which is used to perform the reflection. Since this method transforms all scene geometry and usually requires fine tessellation, it can be either computationally expensive or exhibit accuracy problems. An analytic approach has also been developed [CA00], using a preprocessing step based on path perturbation theory. When objects move, the preprocessing step needs to be repeated.

In the Master Thesis of Schmidt [Sch03] geometry transformation was used for both planar and curved refractions. He noted that multi-pass rendering developed for reflections is only approximately correct for refractions, and also proposed a method to handle refractions on curved surfaces.

Guy and Soler [GS04] used geometry transformation to realistically render gem stones, which have planar faces. A light path in a gem stone contains two refractions and arbitrary number of internal reflections in between. Planar reflection transforms triangles to triangles, thus triangles can be exactly handled by geometry transformation, but refraction distorts geometry. This method "linearized" the refraction of the eye ray, i.e. approximated the refraction by a linear transformation similar to reflections. On the other hand, the refraction of the ray connecting the light source was computed by the fragment shader with per pixel accuracy.

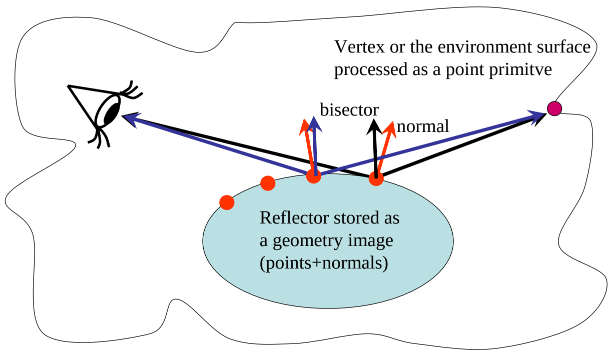

Figure 29: *Searching the geometry image of the refractor for a sample point where the normal vector and the directions toward the processed vertex and toward the eye meet the reflection law, i.e. where the angle enclosed by the normal and the bisector of the two directions is minimal.*

More recently, in [EMD*05], the reflected vertices are paired with reflection points by first localizing the reflection point on each reflector. The localization process finds a point on the reflector surface where the normal vector and the directions from this point toward the camera and the processed point meet the reflection law of geometric optics (Figure 29). The reflector is represented by two floating point textures, the first one stores the 3D coordinates of its sample points, the second represents the surface normals at the sample points (this representation is also called the *geometry image* [GGH02]). The vertices of the environment are sent down the rendering pipeline as point primitives, which generate a single fragment, and the fragment shader program searches the geometry image and locates that sample point where the length of optical path is minimal, i.e. where the associated normal vector and the directions from the sample point toward the camera and toward the processed vertex satisfy the reflection law of geometric optics. Defining a bisector between the directions from the sample point toward the camera and toward the processed vertex, the search where the law of reflection is satisfied becomes equivalent to an optimization that locates the minimum of the angle between the bisector and the normal vector. Unfortunately, if the reflective surface is neither convex nor everywhere concave there might be several minima, which limits the application of such approaches to simple reflector geometry (Figure 5).

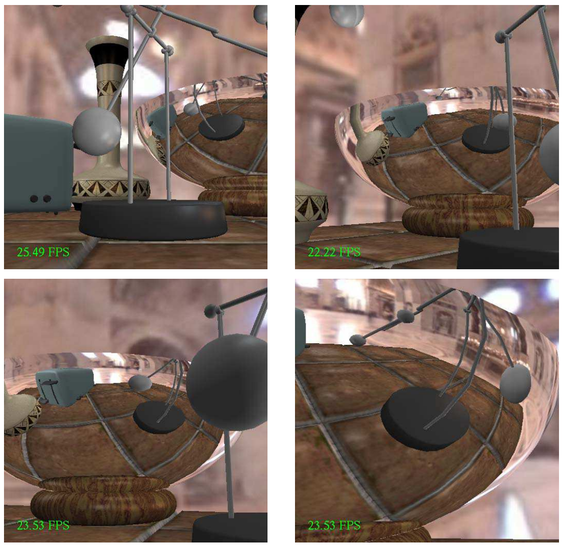

Figure 30: *Reflections on a spherical bowl with geometry transformation [EMD*05].*

Later, in [EMDT06], this technique has been improved by rendering and blending a virtual object for every reflector and for every other object reflected in the reflector. For each vertex of each virtual object, a reflection point is found on the reflector's surface, and used to find the reflected virtual vertex. In [RHS06] the same strategy is used, with a different refinement criterion, keeping geometric bounds on the reflected position for robustness.

6. Caustics

Caustics show up as beautiful patterns on diffuse surfaces, formed by light paths originating at light sources and visiting mirrors or refracting surfaces. Caustics are the concentration of light, which can "burn". The name caustic, in fact, comes from the Latin "causticus" derived from Greek "kaustikos", which means "burning". These indirect effects have a significant impact on the final image [Jen01, TS00, WS03].

Light paths starting at the light sources and visiting specular reflectors and refractors until they arrive at diffuse surfaces need to be simulated to create caustic effects. Theoretically such paths can be built starting the path at the light source and following the direction of the light (light or photon tracing), or starting at the receiver surfaces and going opposite to the normal light (visibility ray tracing). If light sources are small, then the probability that visibility ray tracing finds them is negligible, thus visibility ray tracing is inefficient to render general caustics. Effective caustic generation algorithms have two phases [Arv86]. The first phase identifies the terminal hits of light paths using photon tracing, and the second renders an image for the camera also taking into account the extra illumination of the photons.

6.1. Photon tracing phase

In the first phase photons are followed as they leave the light source, get specularly reflected, and finally arrive at diffuse surfaces generating caustic hits. In GPU based caustic generation algorithms, the simulation of these paths requires ray tracing implemented on the GPU. Using the introduced taxonomy of GPU based ray tracing, this algorithm can process either the original geometry or the sampled geometry.

Hybrid caustic generation algorithms also exist that execute photon tracing on the CPU and filter photon hits [LC04, CSKSN05] or splat photons [LP03] with the aid of the GPU.

6.1.1. Photon tracing in the original geometry

In [PDC*03], Purcell et. al presented a photon mapping algorithm capable of running entirely on GPUs. In their implementation photon tracing uses a breadth-first search algorithm to distribute photons. They store the photons in a grid-based photon map that is constructed directly on the graphics hardware using one of two methods. In the first one, the fragment shaders of a multipass technique directly sort the photons into a compact grid. The second proposed method uses a single rendering pass that combines a vertex program and the stencil buffer to direct photons to their grid cells, producing an approximate photon map.

As an application of their GPU based kd-tree construction in CUDA Zhou et al. also presented a photon mapping method that worked with the original triangle meshes [ZHWG08].

6.1.2. Photon tracing in the sampled geometry

The rasterization hardware and GPU algorithms developed for specular reflections and refractions can also be exploited in caustic generation by exchanging the roles of the eye and a light source.

In the first phase, called the *light pass*, the caustic generator object is rendered from the point of view of the light source, and the terminal hits of caustic paths are determined by a ray tracing algorithm that searches the sampled geometry and is implemented by the fragment shader

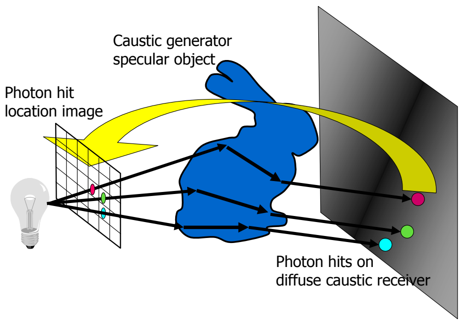
Caustic generator specular object
Photon hit location image
Photon hits on diffuse caustic receiver

Figure 31: *The light pass renders into the photon hit location image where each pixel stores the location of the terminal hit of a photon trajectory that goes through this particular pixel.*

[SKALP05, WD06b, SP07]. In this phase the view plane is placed between the light and the caustic generator object (Figure 31). Following the light path going through the pixel of the image and visiting specular surfaces, the final location of the caustic photon on a diffuse surface can be determined, and this information is stored in the pixel. The resulting image may be called as the *photon hit location image*. Note that this step is very similar to the generation of depth images for shadow maps [Wil78].

6.2. Storing photon hits

Since discrete photon hits should finally be combined to a continuous caustic pattern, a photon hit should affect not only a surface point, but also a surface neighborhood where the power of the photon is distributed. The distribution of photon power may result in light leaks, which can be reduced — but not completely eliminated — if the neighborhood also depends on whether its points are visible from the caustic generator object. Finally, the computation of the reflection of the caustic illumination requires the local BRDF, thus we should find an easy way to reference the surface BRDF with the location of the hit.

There are several alternatives to represent the location of a photon hit (Figure 32):

3D grid [PDC*03]: Similarly to classical photon mapping photon hits can be stored independently of the surfaces in a regular or adaptive 3D grid. When the irradiance of a point is needed, hits that are close to the point are obtained. If the surface normal is also stored with the photon hits, and only those hits are taken into account which have similar normal vectors as the considered surface point, then light leaks can be minimized and photons arriving at back faces can be ignored. Purcell et al. [PDC*03] developed fragment programs for locating the nearest photons in the grid, which makes it possible to compute an estimate of the radiance at any surface location in the scene.

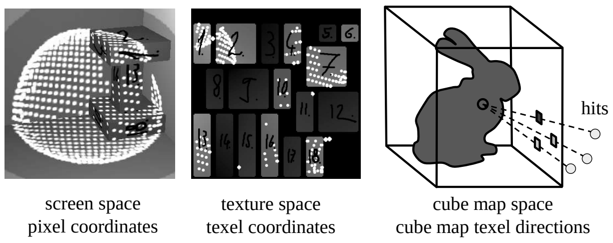

Figure 32: *Three alternatives of storing photon hit locations. Screen space methods store pixel coordinates, texture space methods texture coordinates. Cube map space methods store the direction of the photon hit with respect to the center of a cube map associated with the caustic generator (the bunny in the figure).*

Texture space [GD01,SKALP05,CSKSN05]: Considering that the reflected radiance caused by a photon hit is the product of the BRDF and the power of the photon, the representation of the photon hit should identify the surface point and its BRDF. A natural identification is the pair of texture coordinates of that surface point which is hit by the ray. A pixel of the photon hit location image stores two texture coordinates of the hit position and the luminance of the power of the photon. The photon power is computed from the power of the light source and the solid angle subtended by the pixel of photon hit location image. Since the texture space neighborhood of a point visible from the caustic generator may also include occluded points, light leaks might show up.

Screen or image space [LC04, WD06b]: A point in the three-dimensional space can be identified by the pixel coordinates and the depth when rendered from the point of view of the camera. This screen space location can also be written into the photon hit location image. If photon hits are represented in image space, photons can be splat directly onto the image of the diffuse caustic receivers without additional transformations. However, the BRDF of the surface point cannot be easily looked up with this representation, and we should modulate the rendered color with the caustic light, which is only an approximation. This method is limited to cases when not only the caustics, but also its generator object is visible from the camera, and caustics that are seen through reflectors or refractions might be missing. This method is also prone to creating light leaks.

Ray space [IDN02, EAMJ05, KBW06]: Instead of the hit point, the ray after the last specular reflection or refraction can also be stored. When caustic patterns are projected onto the receiver surfaces, the first hit of this ray needs to be found to finalize the location of the hit, which is complicated. Thus these methods either ignore visibility [IDN02, EAMJ05] or do not apply filtering [KBW06]. Methods working without filtering require many photons to be traced, but provide per-pixel accuracy and do not create light leak artifacts.

Shadow map space [SP07]: In the coordinate system of the shadow map, where the light source is in the origin, a point is identified by the direction in which it is visible from the light source and the computed depth. An advantage of this approach is that rendering from the light's point of view is needed by shadow mapping anyway. The drawbacks are the possibility of light leaks and that caustics coming from the outside of the light's frustum are omitted. Shah and Pattanaik [SP07] estimated the intersection point between a single refracted light ray and a receiver using the undistorted image of the receiver as seen from the light source, much in the way shadow map generation works. Then, starting from an initial guess for the intersection point, its position is corrected by iteratively moving this point along the refracted ray in light-screen-space. As a consequence, the quality of this method strongly depends on the initial guess as well as on the size and the shape of the receiver.

Cube map space [UPSK08]: The coordinate system of the distance map used to trace rays leaving the caustic generator automatically provides an appropriate representation. A point is identified by the direction in which it is visible from the reference point, i.e. the texel of the cube map, and also by the distance from the reference point. An appropriate neighborhood for filtering is defined by those points that are projected onto neighboring texels taking the reference point as the center of projection, and having similar distances from the reference point as stored in the distance map. Note that this approach is similar to classical shadow tests and successfully reduces light leaks.

6.3. Photon hit filtering and projection

The photon hit location image contains finite photon hits, from which a continuous signal needs to be reconstructed on the receiver surfaces.

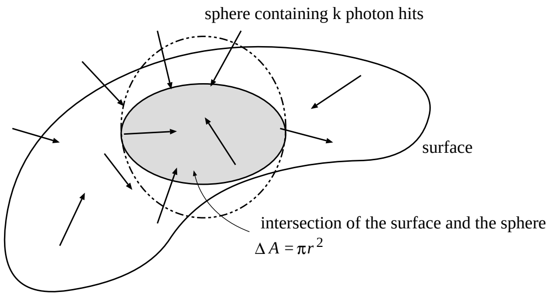
sphere containing k photon hits
surface
intersection of the surface and the sphere
$\Delta A = \pi r^2$

Figure 33: *The adaptive reconstruction scheme of the original photon mapping algorithm.*

The original photon mapping algorithm [Jen96] executed this filtering step during the rendering from the camera (*final*

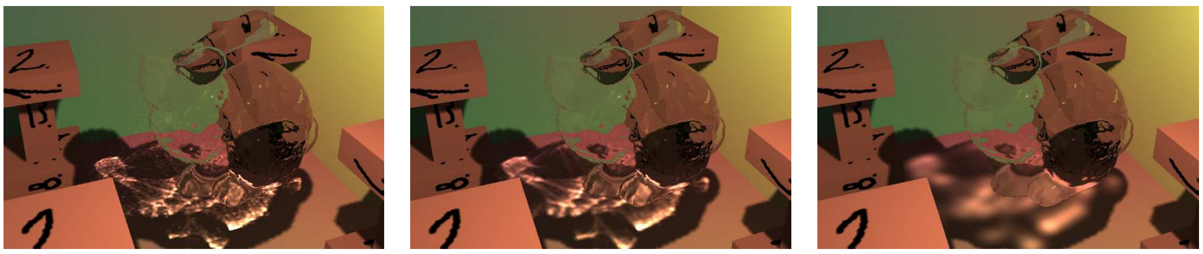

Figure 34: *The problems of splatting. The left image was rendered with too small splatting filter, the right one with too large. To obtain the middle image, the splatting size was manually optimized.*

gathering) and used an adaptive kernel size for reconstruction, which is small where the photon density is high and big where the photon density is small. Actually, the original photon mapping method expanded a sphere until it contained k photons (Figure 33), and approximated the surface area where the average of photon reflections is computed as the area of the main circle of this sphere.

Purcell at al. [PDC*03] used a variation of this adaptive strategy, which is called the k nearest neighbor grid or the *kNN-grid*, and is based on Elias's algorithm to locate the nearest neighbors [Cle79]. A recent method [ZHWG08] implemented a more advanced neighborhood search as well as all other steps of photon mapping in CUDA.

On the other hand, it is simpler to execute the filtering step in the same coordinate system where the photon hits are stored and not directly in camera space. Thus, GPU algorithms often introduce a separate filtering pass between photon tracing and final gathering. The filtering pass creates a texture, called *caustic intensity map*, which is projected onto the surfaces during final gathering. There are two methods to find the area affected by a photon hit: photon splatting and caustic triangles.

6.3.1. Photon splatting

Photon splatting draws a textured semi-transparent quad around the photon hit, and is very popular in GPU based caustic algorithms [SKALP05, SP07, WD06b]. Photon splatting is equivalent to always using the same radius for the filter, but averaging more photons at high density regions. To find a uniformly good splat size is not always possible due to the very uneven distribution of the photon hits (Figure 34). On the other hand, splatting does not take into account the orientation of the receiver surface, which results in unrealistically strong caustics when the surface is lit from grazing angles. The reconstruction problems of splatting can be reduced by adaptive size control [WD06a] or by hierarchical methods [Wym08], but these approaches either always work with the largest splat size or with multiple hierarchical levels, which reduces their rendering speed.

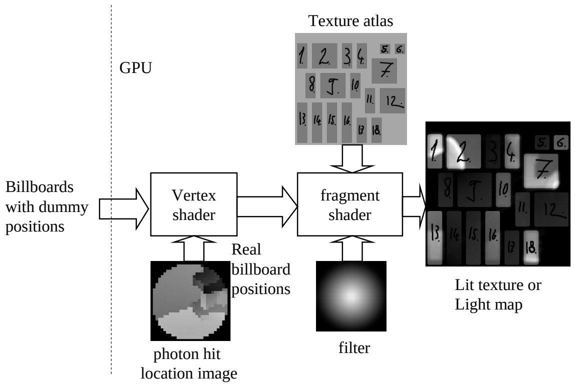

Figure 35: *Photon hit filtering pass assuming that the photon hit location image stores texture coordinates.*

To execute spatting on the GPU, as many small quadrilaterals or point sprites are sent down the pipeline as photons the photon hit location image has (Figure 35). The size of these quadrilaterals controls the support of the blurring filter. We can avoid the expensive GPU to CPU transfer of the photon hit location image, if the vertex shader modulates the location of points by the content of the photon hit location image. This operation requires at least Shader Model 3 GPUs that allow the vertex shader to access textures.

Rectangles with the corrected position are rasterized, and their fragments are processed by the fragment shader. In order to splat the photon on the render target, a splatting filter texture is associated with the rectangle and additive alpha blending is turned on to compute the total contribution of different photon hits (Figure 36).

Splatting can work directly in screen space or in other spaces where the photon hits are stored. Figure 35 shows the case when splatting is executed in texture space, resulting in a light map or a modified texture. These can be mapped onto the surfaces by an additional pass.

We note that photon splatting is also popular in render-

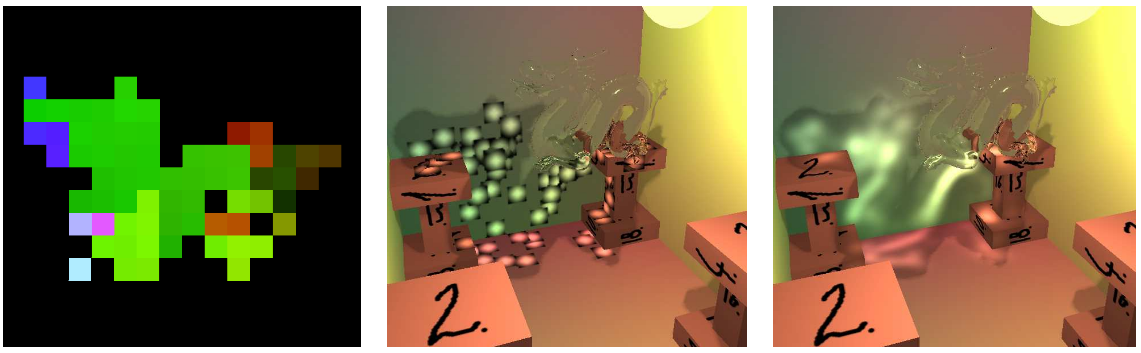

Figure 36: *A low resolution photon hit location image (left), a room with photon hits but without blending (middle), and a room with blending enabled (right).*

ing diffuse and glossy indirect illumination [LP03, DS06, HHK*07].

6.3.2. Caustic triangles

Neighboring rays emitted by the light source form a beam. *Beam tracing* [Wat90] computes the reflection or refraction of these beams defined by corner rays. A beam represents a 3D volume where points may be affected by caustics, thus to find caustic receivers, these volumes are intersected by the surfaces [IDN02,EAMJ05]. If a beam is defined by three rays, an intersection can be approximated by a triangle.

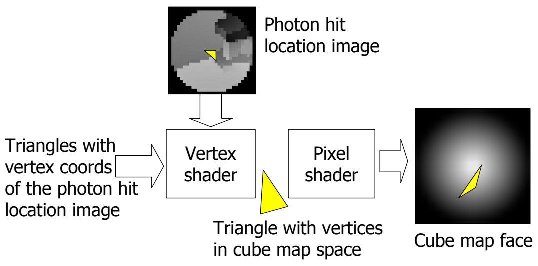
Photon hit location image
Triangles with vertex coords of the photon hit location image
Vertex shader
Pixel shader
Triangle with vertices in cube map space
Cube map face

Figure 37: *Rendering a caustic triangle into the light cube map.*

Thus, for caustic reconstruction we can take three neighboring photon hits and assume that they form a *caustic triangle* [UPSK08]. These triangles are additively blended. Figure 37 shows the case when the reconstruction results in a light cube map (Figure 38). Unlike splatting-based methods the caustic triangles algorithm does not exhibit problems depending on the splatting filter since it provides continuous patterns without overblurring even at moderate resolutions. However, sharp triangle boundaries may be disturbing so we should filter the caustic pattern before projection.

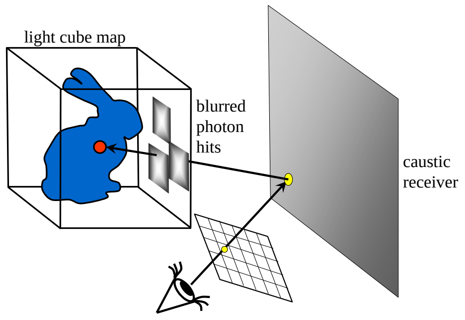

Figure 38: *Light projection pass assuming that the photon hit location image stores cube map texel directions. When a point visible from the camera is illuminated, its distance to the reference point of the light cube map is computed and compared to the distance stored at the texel corresponding to the direction of this point. If the two distances are similar, then lighting from caustics affects this point, and the intensity is read from the texel, which was computed by photon hit filtering in the previous pass.*

6.3.3. Projection of caustic patterns

If photon hits are stored and filtering is executed in a space other than the screen space, in the camera pass the illumination caused by caustics should be projected onto the surfaces visible from the real camera (Figure 38). This requires one additional pass but it becomes straightforward to compute also the reflections and refractions of caustic patterns (Figures 39 and 40).

If photon hits are stored in texture space and filtering is executed there, then the final projection is automatically executed by a normal texturing or light mapping step [SKALP05].

If photon hits are defined and filtered in cube map space (Figure 38) or in shadow map space [SP07, WD06b,

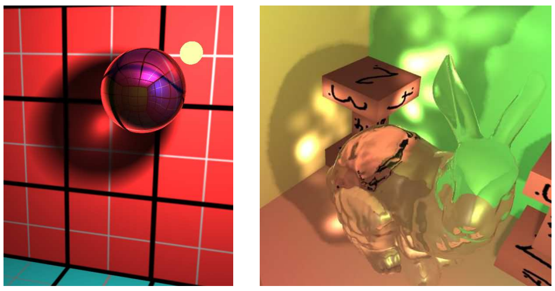

Figure 39: *Real-time caustics caused by glass objects (*$\nu = 1.3$*). A* 64×64 *resolution photon hit location image is obtained in each frame, which is fed back to the vertex shader. The method runs at 182 FPS on an NV6800GT [SKALP05].*

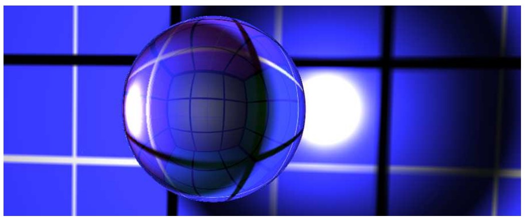

Figure 40: *Caustics seen through the refractor object.*

UPSK08], then the projection gets very similar to shadow mapping algorithms. The point processed during camera pass is transformed into the coordinate system of the cube map and the direction from the cube map center to the processed point is obtained. The cube map texels store distance values. The distance associated with the direction is fetched and is compared to the distance of the processed point. If these values are similar, then the processed point is visible from the point of view of the caustic generator, so no object prohibits the processed point to receive the extra illumination of the caustics, that is stored in the cube map texel. Although this approach requires one additional texture lookup in the shader, it reduces light leaks caused by splatting.

If photon hits are stored in ray space, the three neighbors form a *caustic beam* or *illumination volume* [NN94], and this beam is intersected with the caustic receiver surfaces [IDN02, EAMJ05]. Caustic patterns are displayed by drawing the intersection areas between all of the illumination volumes and the receiver surfaces, and by accumulating the intensities of light reaching the intersection area. Unfortunately, this method treated neither shadows nor the case of warped volumes which can occur in beam tracing. Ernst et al. [EAMJ05] solved these situations and also presented a caustic intensity interpolation scheme to reduce aliasing, which resulted in smoother caustics. However, this algorithm also ignored occlusions, so was unable to obtain shadows.

Krüger et al. [KBW06] presented a photon tracing algorithm that stored dense hits in ray space. These dense hits are not filtered but are projected onto the surfaces one by one providing per pixel accuracy and eliminating light leaks that might be caused by filtering. The projection is executed by rendering a line in image space for each photon, and letting the fragment program check whether the point of the line is similar to the point represented by the current pixel, which indicates an intersection. To find the first intersection each ray is rendered into a separate row of a texture, and the first hit is selected by additional texture processing passes.

6.4. Simplified and pre-computation aided caustic algorithms

Underwater caustic generation has received significant attention. The early work by Jos Stam [Sta96] pre-computed underwater caustic textures and mapped them onto objects in the scene. Although this technique is extremely fast, the caustics produced are not correct given the shape of the water surface and the receiver geometry. Later, Trendall and Stewart [TS00] computed refractive caustics by performing numerical integration for the caustic intensities on a flat receiver surface. Their method cannot support arbitrary receiver geometry and cannot be easily extended to handle shadows.

The algorithm presented by Daniel Sanchez-Crespo in [SC03] applied aggressive assumptions on good candidates for caustic paths and computed them with visibility ray tracing. The method has very low computational cost, and produces something that, while being totally incorrect from a physical standpoint, very closely resembles a caustic look and behavior. The first assumption is that caustics are caused by the sun that is directly above the refractive water surface. The second assumption is that the ocean floor is located at a constant, shallow depth. Then, their three-pass algorithm works as follows. The first pass renders the ocean floor as a regular textured quad. Then, in a second pass, the same floor is painted again using a fine mesh, which is lit per-vertex using their caustic generator. For each vertex in the fine mesh, a ray is shot vertically. The ray ocean surface intersection is obtained, the bent ray is computed with the Snellius law, and the “sky/sun” texture is read with the direction of the bent ray. The third and final pass renders the ocean waves using environment mapping to get a sky reflection on them.

Wand and Straßer [WS03] developed an interactive caustic rendering technique by explicitly sampling points on the caustic-forming object. The receiver geometry is rendered by considering the caustic intensity contribution from each of the sample points without visibility tests. The intensity of a sample point is obtained by looking up a cube map representing the direct light sources in the reflection direction. The authors presented results using reflective caustic-forming objects, but refractive caustics can also be achieved with this technique. However, the explicit sampling hinders the scalability of the algorithm since the amount of computation is directly proportional to the number of sample points.

6.5. Specular effects in varying materials

Methods surveyed so far assumed that the materials are homogeneous, i.e. the refraction index inside an object is constant, so is the refraction index of the air. This assumption allows to replace light paths by line segments between surfaces. However, in inhomogeneous materials the light follows a curved path [Gro95], which makes refraction and caustic computation more complicated [IZT*07]. Such non-linear ray tracing [WSE04] methods can also visualize relativistic effects.

7. Conclusions

In this STAR we reviewed different approaches aiming at rendering specular effects on the GPU. We can conclude that although GPUs originally did not aim at tracing incoherent secondary rays needed for such effects, they are versatile enough for this purpose as well. If the GPU is controlled by a graphics API, it may access textures while processing a vertex or a fragment, thus these approaches require the representation of the scene radiance and geometry in textures. There are many different possibilities to encode geometry in textures, which lead to a large selection of algorithms of GPU based specular effects.

We established several main categories, depending on whether the original or the sampled geometry is intersected by the rays, and on whether reflection or reflected points are identified (Table 1):

- The simplest methods for computing reflections have been the environment map based approaches, but, due to the parallax approximations involved, they are only suitable for small specular objects put in large, distant environment, or when accuracy is not important. If a separate environment map is maintained for every reflector, then this method can even be used for recursive reflections/refractions.
- Proxy geometries can mimic the missing parallax effects in an approximate form, with no tradeoff between precision and speed. Although the application of proxy geometries might generate reflections that are very different from the real phenomena, the results are pleasing to the human eye since there are no significant sampling artifacts.
- Image based rendering methods also offer solutions for reflections and refractions, but they do not fit well to usual game and scene graph management software.
- Methods working with the sampled geometry of the environment are usually very fast and just a little more difficult to implement than environment mapping. Since they are based on the concept of processing an object and its dynamic texture, these algorithms can be seamlessly integrated into game engines and scene graph management software, and can benefit from the fast visibility algorithms developed for static [BWW01] and dynamic scenes [WB05]. For example, a distance map is like an environment map, while a caustic cube map is like a shadow map. If a game engine supports environment maps and shadow maps, then it can also support distance map based reflection/refraction and caustic rendering (Figures 1, 41–43). Inter-object multiple reflections are obtained easily if every specular object has its own distance map, and it is also possible to extend the method for self refractions using an additional map of normals per refractor. An algorithm searching on the environment surface is useful for handling arbitrary reflector surfaces and is particularly efficient if the environment surface does not have large distance variations. However, sampling problems may occur if the environment includes thin objects that are close to the reflector surface, or the environment surface exhibits view dependent occlusions when seen from the point of view of the specular objects. The accuracy of the methods can be tuned by increasing the resolution of the cube map and the number of search steps used to find the intersection. However, the precision is limited by the occlusion problem.
- Multi-layer maps storing the sampled geometry can handle view dependent occlusions and even self reflections and refractions. In theory, these methods offer per-pixel accuracy if the number of layers, the resolution of the maps, and the number of search steps are high enough. In practice, however, less accurate results are traded for high frame rates.
- Reflection mapping algorithms for planar reflectors have been around to simulate recursive planar reflections for a long time. They transform the scene from the reflected viewpoint, which has been the dominant way of simulating polished floors in games. These methods are exact, but are limited to planar reflectors. The application of similar techniques to refractions in just an approximation, since refraction even on planar surfaces is a non-linear geometry transformation, unlike reflection.
- Geometry transformation to handle reflection on curved reflectors has been a more difficult task, as a search on the reflector surface is needed to find the correct reflection point for every scene vertex. Algorithms searching on the reflector surface can cope with reflected surfaces of large depth variations, but are limited to simple reflectors that are either concave or convex. Geometry transformation methods are more complicated to implement than environment mapping, and their extension to multiple reflections is not as straightforward. The accuracy and the speed of geometry transformation methods can also be controlled by modifying the tessellation level of the environment surfaces and the resolution of the geometry images representing the reflectors.
- Methods working with the original geometry are close to

CPU based ray tracing algorithms and inherit their flexibility, generality, and accuracy. These methods are "exact" i.e. can provide results with floating point accuracy. However, the performance of GPU implementations is not very appealing. Despite the fact that the floating point processing power of GPUs is more than one order of magnitude higher than that of a CPU, only recent GPU solutions can outperform CPU ray tracers in speed. The reason is that these algorithms utilize GPU resources in a fundamentally different way than the incremental rendering pipeline, for which GPUs are not optimized to. It is likely that the performance of such algorithms would further improve on newer hardware and APIs that are optimized also for GPGPU computations. The introduction of CUDA is an important step forward. We can conclude that, until GPUs evolve as mentioned, GPU ray tracers should be kept for very accurate rendering simulations or when the number of specular objects gets high in the scene.

All methods belonging to the discussed categories are able to render single reflections, but just a few of them are good for self reflections or refractions. With respect to self refractions, depending on the degree of desired accuracy, the implementor can choose from approximate methods [Wym05a, DW07] or use algorithms that search in a distance map or height-map representation of the object [SKALP05,HQ07,OB07]. Approximate ray tracing in multi-layer sampled geometry [KBW06,UPSK07], and ray tracing of the original geometry are appropriate for both self refractions and reflections.

Speed can be considered from two different aspects. We can take specific scenes and measure the frame rates on a specific GPU (such results are shown by the captions of figures in this paper), which gives a feeling how fast these approaches are. Considering current hardware and scenes of complexity shown by these figures, algorithms using the sampled geometry are at least an order of magnitude faster than ray tracing in the original geometry.

On the other hand, we can study the complexity of these methods, which indicates the asymptotic behavior when the scene complexity grows. Ray tracing in the original geometry is algorithmically equivalent to classic ray tracing algorithms. Denoting the number of triangles by N, naive ray tracing requires $O(N)$ time to trace a ray. Paying the super-linear cost of building a space partitioning scheme, the tracing cost can be sub-linear [SKM98, Hav01, SKHBS02, HP03]. Examining average case behavior and assuming that objects are uniformly distributed, in the optimal case these schemes can be constructed in $O(N \log N)$ time, and a ray can be shot in $O(1)$ time if its origin is already located. The location of the origin needs $O(\log N)$ time in hierarchical structures. We have to emphasize that these formulae apply for the average case, in the worst case, the presented algorithms are not better than the naive implementation. Tracing a specular path of length R needs R rays and thus $O(R)$ time.

Ray tracing in the sampled geometry requires the creation and look up of environment and distance maps, which are obtained by rasterization. Rasterization based image synthesis runs in $O(N)$ time without preprocessing. However, taking advantage of the latest scene graph management approaches and visibility algorithms ([BWW01] and [WB05] are just two examples from the long list of such methods), these methods can also be speeded up similarly to ray tracing after super-linear preprocessing. If inter-object specular effects are also demanded, we need an environment or distance map for each reflector/refractor. Thus, the construction cost of these maps is proportional to the number of specular objects (S in Table 1) and to the cost of the rasterization of all non-specular objects (D). However, when the map is available, the intersection point is obtained in constant time, independently of the geometric complexity of the scene.

A single multi-layer map can represent the whole scene, independently of the number of specular and non-specular objects. Multi-layer maps are constructed by depth peeling, which rasterizes scene objects several times, depending on the depth complexity (L) of the scene. The depth complexity equals to the number of intersection points a ray may produce. Searching a map of L layers has complexity $O(L)$.

Reflection mapping used for planar specular reflectors doubles the number of view points with each reflector, but similarly to multiple environment maps, the cost of additional reflections can be traded for increased latency in an animation sequence. Geometry transformation, which can render curved reflections, use a map for each reflector, so its construction cost is proportional to the number of reflectors. During rendering, the reflection point of each vertex belonging to a non-specular surface (D) is searched in each map, which results in $O(S \cdot D)$ complexity.

Summarizing the complexity observations, we can state that ray tracing in the original geometry and multi-layer maps are the winners if the scene consists of many specular objects and inter-object or self reflections are also needed. Environment map or distance map based techniques are good if there are only a few specular objects. Geometry transformation also has this limitation in addition to the disadvantage that the cost of tracing a reflection ray is proportional to the scene complexity.

In this paper we also reviewed GPU based caustic algorithms. These methods begin with the photon tracing phase, where photons are followed from the sources up to diffuse surfaces. This pass may exploit the GPU ray tracing approaches developed for reflections or refractions. Photon hits should be stored, filtered, and finally projected onto the visible surfaces. The coordinate system where the photon hits are stored and the additional information used during filtering may affect the scale of potential light leaks caused by filtering. For this, there are basically four options: nearest neighbors search, which is the most versatile, but also the most complex and expensive solution; splatting, where the

	Accuracy	Reflection/refraction	Complexity	Greatest limitation
Environment mapping with one map [BN76]	approximate	single	$O(D)$, $O(1)$	no view parallax
Environment mapping with one map per specular object [NC02]	approximate	single and inter-object multiple reflection/refraction	$O(S \cdot D)$, $O(1)$	no view parallax
Proxy geometry with one map per specular object [Bre02, Bjo04]	approximate	single and inter-object multiple reflection/refraction	$O(S \cdot D)$, $O(1)$	accuracy
One single layer distance map per specular object [SKALP05]	tradeoff	single and inter-object multiple reflection/refraction	$O(S \cdot D)$, $O(1)$	occlusion artifacts
One single layer distance map and refractor map per refractor object [Wym05a, SKALP05, HQ07]	tradeoff	multiple (double) self refractions and inter-object multiple reflection/refraction	$O(S + S \cdot D)$, $O(1)$	occlusion artifacts
Sampled geometry with one multi layer map [KBW06, UPSK07]	tradeoff	all	$O(N \cdot L)$, $O(L)$	sampling artifacts
Reflection mapping on planar reflectors [DB97, MBGN00]	exact	multiple reflections	$O(1)$, $O(S \cdot 2^R \cdot N)$	planar reflector
Geometry transformation on curved reflectors [EMDT06, RHS06]	tradeoff	single reflections	$O(S)$, $O(N)$	high tessellation
Ray tracing in original geometry with regular grid [PBMH02]	exact	all	$O(N^2)$, $O(R)$	speed, storage
Ray tracing in original geometry with hierarchical space partitioning [HSHH07, PGSS07, GPSS07] [ZHWG08]	exact	all	$O(N \log N)$, $O(R \cdot \log N)$	speed, storage
Ray tracing in original geometry with ray hierarchy [Sze06, RAH07]	exact	all	$O(1)$, $O(R \cdot N)$	speed

Table 1: *Comparison of algorithms computing specular reflections and refractions. In the Accuracy column "approximate" means that the method has some deterministic error that cannot be controlled. "Tradeoff", on the other hand, means that there is the possibility to find a tradeoff between accuracy and speed. The column entitled Complexity contains the costs of preprocessing and of computing the color where a specular object is visible, respectively. Preprocessing should be repeated if the scene changes. The notations are as follows: N is the number of triangles, S is the number of specular objects, D is the number of triangles of non-specular objects. L is the depth complexity of the scene, (i.e. the maximum number of hits a ray may produce), R is the recursion depth. These formulae correspond to average case complexity in case of ray tracing in the original geometry. The worst case complexity of tracing a ray would be at least* $O(N \cdot R)$.

size of the splats must be manually tuned; beam-based approaches, which ignore occlusions and thus they are unable to produce shadows; and the projection of caustic triangles.

8. Acknowledgement

This work has been supported by the National Office for Research and Technology, Hewlett-Packard, OTKA K-719922 (Hungary), TIN2007-67120 and TIN2007-68066-C04-01 from the Spanish Government, and by the GameTools FP6 (IST-2-004363) project.

References

[AK87] Arvo J., Kirk D.: Fast ray tracing by ray classification. In *Computer Graphics (SIGGRAPH '87 Proceedings)* (1987), pp. 55–64.

[AK89] Arvo J., Kirk D.: A survey of ray tracing acceleration techniques. In *An Introduction to Ray Tracing*, Glassner A. S., (Ed.). Academic Press, London, 1989, pp. 201–262.

[Ama84] Amanatides J.: Ray tracing with cones. In *Computer Graphics (SIGGRAPH '84 Proceedings)* (1984), pp. 129–135.

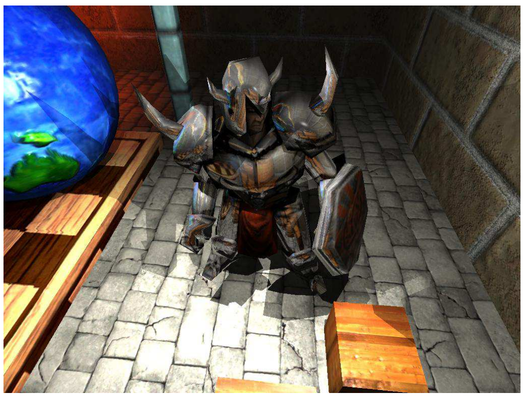

Figure 41: *A knight in reflective armor in textured environment illuminated by dynamic lights (130 FPS) [SKALP05].*

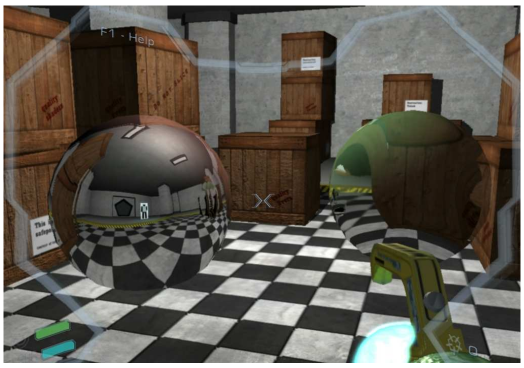

Figure 42: *Reflective and refractive spheres in PentaG (http://www.gebauz.com/)*

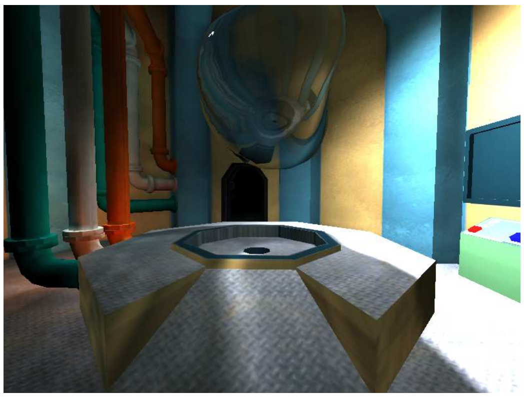

Figure 43: *Deforming glass bubble generates reflections, refractions, and caustics in Space Station (http://www.gametools.org)*

[Arv86] ARVO J.: Backward ray tracing. In *SIGGRAPH '86 Developments in Ray Tracing* (1986).

[Arv91] ARVO J.: Linear-time voxel walking for octrees. *Ray Tracing News 1*, 2 (1991). available under anonymous ftp from weedeater.math.yale.edu.

[AW87] AMANATIDES J., WOO A.: A fast voxel traversal algorithm for ray tracing. In *Proceedings of Eurographics '87* (1987), pp. 3–10.

[BHWL99] BASTOS R., HOFF K., WYNN W., LASTRA A.: Increased photorealism for interactive architectural walkthroughs. In *SI3D 99: Proceedings of the 1999 symposium on Interactive 3D graphics* (1999), pp. 183–190.

[Bjo04] BJORKE K.: Image-based lighting. In *GPU Gems*, Fernando R., (Ed.). NVidia, 2004, pp. 307–322.

[Bly06] BLYTHE D.: The Direct3D 10 system. In *SIGGRAPH 2006 Proceedings* (2006), pp. 724–734.

[BN76] BLINN J. F., NEWELL M. E.: Texture and reflection in computer generated images. *Communications of the ACM 19*, 10 (1976), 542–547.

[Bre02] BRENNAN C.: Accurate environment mapped reflections and refractions by adjusting for object distance. In *Shader X* (2002), Engel W., (Ed.), Woodware.

[BT04] BRAWLEY Z., TATARCHUK N.: Parallax occlusion mapping: Self-shadowing, perspective-correct bump mapping using reverse height map tracing. In *ShaderX 3*, Engel W., (Ed.). Charles River Media, Cambridge, MA, 2004.

[BWSF06] BENTHIN C., WALD I., SCHERBAUM M., FRIEDRICH H.: Ray tracing on the cell processor. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (2006).

[BWW01] BITTNER J., WONKA P., WIMMER M.: Visibility preprocessing for urban scenes using line space subdivision. In *Pacific Graphics* (2001), pp. 276–284.

[CA00] CHEN M., ARVO J.: Perturbation methods for interactive specular reflections. *IEEE Transactions on Visualization and Computer Graphics 6*, 3 (2000), 253–264.

[CHCH06] CARR N., HOBEROCK J., CRANE K., HART J.: Fast GPU ray tracing of dynamic meshes using geometry images. In *Graphics Interface 2006* (2006), pp. 203–209.

[CHH02] CARR N., HALL J., HART J.: The ray engine. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (2002), pp. 37–46.

[Chr05] CHRISTEN M.: Ray tracing on GPU, 2005. Master's thesis, Univ. of Applied Sciences Basel (FHBB).

[Cle79] CLEARY J. G.: Analysis of an algorithm for finding nearest neighbors in euclidean space. *ACM Transactions on Mathematical Software (TOMS) 5*, 2 (1979), 183–192.

[CON99] CABRAL B., OLANO M., NEMEC P.: Reflection space image based rendering. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques* (1999), pp. 613–620.

[CSKSN05] CZUCZOR S., SZIRMAY-KALOS L., SZÉCSI L., NEUMANN L.: Photon map gathering on the GPU. In *Eurographics short papers* (2005), pp. 117–120.

[CT81] COOK R., TORRANCE K.: A reflectance model for computer graphics. *Computer Graphics 15*, 3 (1981), 7–24.

[DB97] DIEFENBACH P., BADLER N.: Multi-pass pipeline rendering: realism for dynamic environments. In *SI3D 97: Proceedings of the 1997 symposium on Interactive 3D graphics* (1997), pp. 59–68.

[DBY98] DEBEVEC P., BORSHUKOV G., YU Y.: Efficient view-dependent image-based rendering with projective texture-mapping. In *9th Eurographics Rendering Workshop* (1998), pp. 105–116.

[Deb98] DEBEVEC P.: Rendering synthetic objects into real scenes: Bridging traditional and image-based graphics with global illumination and high dynamic range photography. In *SIGGRAPH '98* (1998), pp. 189–198.

[Don05] DONELLY W.: Per-pixel displacement mapping with distance functions. In *GPU Gems 2*, Parr M., (Ed.). Addison-Wesley, 2005, pp. 123–136.

[DS06] DACHSBACHER C., STAMMINGER M.: Splatting indirect illumination. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games* (2006), pp. 93–110.

[DSSD99] DECORET X., SCHAUFLER G., SILLION F., DORSEY J.: Improved image-based impostors for accelerated rendering. *Computer Graphics Forum, (Eurographics '99) 18*, 3 (1999), 207–218.

[DW07] DAVIS S., WYMAN C.: Interactive refractions with total internal reflections. In *Proceedings of Graphics Interface* (2007), pp. 185–190.

[EAMJ05] ERNST M., AKENINE-MOLLER T., JENSEN H. W.: Interactive rendering of caustics using interpolated warped volumes. In *Proceedings Graphics Interface* (2005), pp. 87–96.

[EMD*05] ESTALELLA P., MARTIN I., DRETTAKIS G., TOST D., DEVILLIERS O., CAZALS F.: Accurate interactive specular reflections on curved objects. In *Proceedings of Vision, Modeling, and Visualization 2005* (2005).

[EMDT06] ESTALELLA P., MARTIN I., DRETTAKIS G., TOST D.: A GPU-driven algorithm for accurate interactive specular reflections on curved objects. In *Proceedings of the 2006 Eurographics Symposium on Rendering* (2006), pp. 313–318.

[ESK03] EVERS-SENNE J., KOCH R.: Image based interactive rendering with view dependent geometry. *Computer Graphics Forum (Eurographics '03) 22*, 3 (2003), 573–582.

[Eve01] EVERITT C.: *Interactive order-independent transparency.* Tech. rep., NVIDIA Corporation, 2001.

[EVG04] ERNST M., VOGELGSANG C., GREINER G.: Stack implementation on programmable graphics hardware. In *Proceedings of Vision, Modeling, and Visualization* (2004), pp. 255–262.

[FS05] FOLEY T., SUGERMAN J.: Kd-tree acceleration structures for a GPU raytracer. In *Proceedings of Graphics Hardware 2005* (2005), pp. 15–22.

[FTK86] FUJIMOTO A., TAKAYUKI T., KANSEI I.: Arts: Accelerated ray-tracing system. *IEEE Computer Graphics and Applications 6*, 4 (1986), 16–26.

[GD01] GRANIER X., DRETTAKIS G.: Incremental updates for rapid glossy global illumination. *Computer Graphics Forum (Eurographics '01) 20*, 3 (2001), 268–277.

[GGH02] GU X., GORTLER S. J., HOPPE H.: Geometry images. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques* (2002), pp. 355–361.

[GGSC96] GORTLER S., GRZESZCZUK R., SZELISKI R., COHEN M.: The lumigraph. In *SIGGRAPH 96* (1996), pp. 43–56.

[Gla84] GLASSNER A.: Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications 4*, 10 (1984), 15–22.

[Gla89] GLASSNER A.: *An Introduction to Ray Tracing*. Academic Press, London, 1989.

[GLD06] GÉNEVAUX O., LARUE F., DISCHLER J.-M.: Interactive refraction on complex static geometry using spherical harmonics. In *SI3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games* (2006), pp. 145–152.

[GPSS07] GUENTHER J., POPOV S., SEIDEL H.-P., SLUSALLEK P.: Realtime ray tracing on GPU with BVH-based packet traversal. In *Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing 2007* (2007), pp. 113–118.

[Gre84] GREENE N.: Environment mapping and other applications of world projections. *IEEE Computer Graphics and Applications 6*, 11 (1984), 21–29.

[Gro95] GROELLER E.: Nonlinear ray tracing: visualizing strange worlds. *The Visual Computer 11*, 5 (1995), 263–274.

[GS04] GUY S., SOLER C.: Graphics gems revisited: fast and physically-based rendering of gemstones. *ACM Transactions on Graphics (Proceedings of SIGGRAPH'04)* (2004), 231–238.

[GWS04] GÜNTHER J., WALD I., SLUSALLEK P.: Real-time caustics using distributed photon mapping. In *Proceedings of the 15th Eurographics Symposium on Rendering* (2004), pp. 111–121.

[Har04] HARGREAVES S.: Hemisphere lighting with radiosity maps. In *ShaderX 2: Shader Programming Tips and Tricks with DirectX 9*, Engel W., (Ed.). Wordware, 2004, pp. 113–121.

[Hav01] HAVRAN V.: *Heuristic Ray Shooting Algorithms*. Czech Technical University, Ph.D. dissertation, 2001.

[HEGD04] HIRCHE J., EHLERT A., GUTHE S., DOGGETT M.: Hardware accelerated per-pixel displacement mapping. In *Proceedings of Graphics Interface* (2004), pp. 153–158.

[HHK*07] HERZOG R., HAVRAN V., KINUWAKI S., MYSZKOWSKI K., SEIDEL H.-P.: Global illumination using photon ray splatting. *Computer Graphics Forum (Eurographics'07) 26*, 3 (2007), 503–513.

[HHS06] HAVRAN V., HERZOG R., SEIDEL H.-P.: On the fast construction of spatial data structures for ray tracing. In *Proceedings of IEEE Symposium on Interactive Ray Tracing 2006* (2006), pp. 71–80.

[HLCS99] HEIDRICH W., LENSCH H., COHEN M., SEIDEL H.-P.: Light field techniques for reflections and refractions. In *Eurographics Rendering Workshop* (1999), pp. 187–196.

[HLL07] HUI K. C., LEE A. H. C., LAI Y. H.: Accelerating refractive rendering of transparent objects. *Computer Graphics Forum 26*, 1 (2007), 24–33.

[HP03] HAVRAN V., PURGATHOFER W.: On comparing ray shooting algorithms. *Computers & Graphics 27*, 4 (2003), 593–604.

[HQ07] HU W., QIN K.: Interactive approximate rendering of reflections, refractions, and caustics. *IEEE TVCG 13*, 1 (2007), 46–57.

[HS01] HAKURA Z. S., SNYDER J. M.: Realistic reflections and refractions on graphics hardware with hybrid rendering and layered environment maps. In *Proceedings of the Eurographics Rendering Workshop* (2001), pp. 289–300.

[HSC*05] HENSLEY J., SCHEUERMANN T., COOMBE G., SINGH M., LASTRA A.: Fast summed-area table generation and its applications. *Computer Graphics Forum 24*, 3 (2005), 547–555.

[HSHH07] HORN D. R., SUGERMAN J., HOUSTON M., HANRAHAN P.: Interactive k-d tree GPU raytracing. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games* (2007), pp. 167–174.

[HSS*05] HADWIGER M., SIGG C., SCHARSACH H., BÜHLER K., GROSS M.: Real-time ray-casting and advanced shading of discrete isosurfaces. *Computer Graphics Forum (Eurographics '05) 22*, 3 (2005), 303–312.

[HTSG91] HE X., TORRANCE K., SILLION F., GREENBERG D.: A comprehensive physical model for light reflection. *Computer Graphics 25*, 4 (1991), 175–186.

[HWSG06] HOU X., WEI L.-Y., SHUM H.-Y., GUO B.: Real-time multi-perspective rendering on graphics hardware. In *Eurographics Symposium on Rendering* (2006), pp. 93–102.

[IDN02] IWASAKI K., DOBASHI Y., NISHITA T.: An efficient method for rendering underwater optical effects using graphics hardware. *Computer Graphics Forum 21*, 4 (2002), 701–711.

[IWRP06] IZE T., WALD I., ROBERTSON C., PARKER S.: An Evaluation of Parallel Grid Construction for Ray Tracing Dynamic Scenes. *Interactive Ray Tracing 2006, IEEE Symposium on* (2006), 47–55.

[IZT*07] IHRKE I., ZIEGLER G., TEVS A., THEOBALT C., MAGNOR M., SEIDEL H.-P.: Eikonal rendering: Efficient light transport in refractive objects. *ACM Transactions on Graphics (SIGGRAPH '07)* (2007).

[Jen96] JENSEN H. W.: Global illumination using photon maps. In *Rendering Techniques '96* (1996), pp. 21–30.

[Jen01] JENSEN H. W.: *Realistic Image Synthesis Using Photon Mapping*. AK Peters, 2001.

[KBW06] KRÜGER J., BÜRGER K., WESTERMANN R.: Interactive screen-space accurate photon tracing on GPUs. In *Eurographics Symposium on Rendering* (2006), pp. 319–329.

[Ken07] KENSLER A.: A state table algorithm for direct ray tracing of CSG models. In *Symposium on Interactive Ray Tracing 2007, Poster* (2007).

[KL04] KARLSSON F., LJUNGSTEDT C. J.: *Ray tracing fully implemented on programmable graphics hardware*. Master's Thesis, Chalmers University of Technology, 2004.

[KM00] KAUTZ J., MCCOOL M.: Approximation of glossy reflection with prefiltered environment maps. In *Proceedings of Graphics Interface* (2000), pp. 119–126.

[KSK01] KELEMEN C., SZIRMAY-KALOS L.: A microfacet based coupled specular-matte BRDF model with importance sampling. In *Eurographics 2001, Short papers, Manchester* (2001), pp. 25–34.

[KVHS00] KAUTZ J., VÁZQUEZ P., HEIDRICH W., SEIDEL H.-P.: A unified approach to prefiltered environment maps. In *11th Eurographics Workshop on Rendering* (2000), pp. 185–196.

[LC04] LARSEN B. D., CHRISTENSEN N.: Simulating photon mapping for real-time applications. In *Eurographics Symposium on Rendering* (2004), pp. 123–131.

[LFY*06] LI S., FAN Z., YIN X., MUELLER K., KAUFMAN A., GU X.: Real-time reflection using ray tracing with geometry field. In *Eurographics 06, Short papers* (2006), pp. 29–32.

[LH96] LEVOY M., HANRAHAN P.: Light field rendering. In *SIGGRAPH 96* (1996), pp. 31–42.

[LP03] LAVIGNOTTE F., PAULIN M.: Scalable photon splatting for global illumination. In *GRAPHITE'03* (2003), pp. 203–211.

[LR98] LISCHINSKI D., RAPPOPORT A.: Image-based rendering for non-diffuse synthetic scenes. In *Eurographics Rendering Workshop* (1998), pp. 301–314.

[LSK05] LAZÁNYI I., SZIRMAY-KALOS L.: Fresnel term approximations for metals. In *WSCG 2005, Short Papers* (2005), pp. 77–80.

[LWX06] LI B., WEI L.-Y., XU Y.-Q.: *Multi-Layer Depth Peeling via Fragment Sort*. Tech. Rep. MSR-TR-2006-81, Microsoft Research, 2006.

[MBGN00] MCREYNOLDS T., BLYTHE D., GRANTHAM B., NELSON S.: Advanced graphics programming techniques using opengl, 2000. Course notes, ACM SIGGRAPH 2000, Course 32.

[MGK05] MESETH J., GUTHE M., KLEIN R.: Interactive fragment tracing. *The Visual Computer 21*, 8-10 (2005), 591–600.

[MH84] MILLER G. S., HOFFMAN C. R.: Illumination and reflection maps: Simulated objects in simulated and real environment. In *SIGGRAPH '84* (1984).

[MJW07] MANTLER S., JESCHKE S., WIMMER M.: *Displacement Mapped Billboard Clouds*. Tech. Rep. TR-186-2-07-01, Institute of Computer Graphics, TU Vienna, Jan. 2007.

[ML03] MEYER A., LOSCOS C.: Real-time reflection on moving vehicles in urban environments. In *VRST '03: Proceedings of the ACM symposium on Virtual reality software and technology* (2003), pp. 32–40.

[MPS05] MEI C., POPESCU V., SACKS E.: The occlusion camera. *Computer Graphics Forum (Eurographics '05) 24*, 3 (2005), 335–342.

[MT97] MÖLLER T., TRUMBORE B.: Fast, minimum storage ray-triangle intersection. *Journal of Graphics Tools 2*, 1 (1997), 21–28.

[NC02] NIELSEN K., CHRISTENSEN N.: Real-time recursive specular reflections on planar and curved surfaces using graphics hardware. *Journal of WSCG 10*, 3 (2002), 91–98.

[NN94] NISHITA T., NAKAMAE E.: Method of displaying optical effects within water using accumulation buffer. In *SIGGRAPH'94 Proceedings* (1994).

[NVI07] NVIDIA: http://developer.nvidia.com/cuda. In *The CUDA Homepage* (2007).

[OB07] OLIVEIRA M. M., BRAUWERS M.: Real-time refraction through deformable objects. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games* (2007), ACM Press, pp. 89–96.

[OLG*05] OWENS J., LUEBKE D., GOVINDARAJU N., HARRIS M., KRÜGER J., LEFOHN A., PURCELL T.: A Survey of General-Purpose Computation on Graphics Hardware. In *EG2005-STAR* (2005), pp. 21–51.

[OM87] OHTA M., MAEKAWA M.: Ray coherence theorem and constant time ray tracing algorithm. In *Computer Graphics 1987. Proc. CG International '87* (1987), Kunii T. L., (Ed.), pp. 303–314.

[OMSD87] OOI B., MCDONELL K., SACKS-DAVIS R.: Spatial kd-tree: An indexing mechanism for spatial databases. *IEEE COMPSAC 87* (1987).

[OR98] OFEK E., RAPPOPORT A.: Interactive reflections on curved objects. In *Proceedings of SIGGRAPH* (1998), pp. 333–342.

[Pat95] PATOW G.: Accurate reflections through a z-buffered environment map. In *Proceedings of Sociedad Chilena de Ciencias de la Computacion* (1995).

[PBMH02] PURCELL T., BUCK I., MARK W., HANRAHAN P.: Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics 21*, 3 (2002), 703–712.

[PCD97a] PATOW G., CASTIGLIONI J. L., DELRIEUX C. A.: Environment mapped refraction models for low cost scan-line rendering. *Computer Networks and ISDN Systems 29*, 14 (1997), 1727–1736.

[PCD*97b] PULLI K., COHEN M., DUNCHAMP T., HOPPE H., SHAPIRO L., STUETZLE W.: View-based rendering: Visualizing real objects from scanned range and color data. In *8th Eurographics Rendering Workshop* (1997), pp. 23–34.

[PDC*03] PURCELL T., DONNER C., CAMMARANO M., JENSEN H. W., HANRAHAN P.: Photon mapping on programmable graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware* (2003), pp. 41–50.

[PGSS06] POPOV S., GÜNTHER J., SEIDEL H.-P., SLUSALLEK P.: Experiences with streaming construction of SAH KD-trees. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (2006), pp. 89–94.

[PGSS07] POPOV S., GÜNTHER J., SEIDEL H.-P., SLUSALLEK P.: Stackless KD-tree traversal for high performance GPU ray tracing. *Computer Graphics Forum (Eurographics '07) 26*, 3 (2007), 415–424.

[PHL91] PATTERSON J. W., HOGGAR S. G., LOGIE J. R.: Inverse displacement mapping. *Computer Graphics Forum 10*, 2 (1991), 129–139.

[PMDS06] POPESCU V., MEI C., DAUBLE J., SACKS E.: Reflected-scene impostors for realistic reflections at interactive rates. *Computer Graphics Forum (Eurographics'2006) 25*, 3 (2006), 313–322.

[POC05] POLICARPO F., OLIVEIRA M. M., COMBA J.: Real-time relief mapping on arbitrary polygonal surfaces.

In *ACM SIGGRAPH 2005 Symposium on Interactive 3D Graphics and Games* (2005), pp. 155–162.

[PSS*06] PABST H., SPRINGER J., SCHOLLMEYER A., LENHARDT R., LESSIG C., FROEHLICH B.: Ray Casting of Trimmed NURBS Surfaces on the GPU. *Interactive Ray Tracing 2006, IEEE Symposium on* (2006), 151–160.

[RAH07] ROGER D., ASSARSSON U., HOLZSCHUCH N.: Whitted ray-tracing for dynamic scenes using a ray-space hierarchy on the GPU. In *Eurographics Symposium on Rendering* (2007), pp. 99–110.

[Ral07] RALOVICH K.: Implementing and analyzing a GPU ray tracer. In *Central European Seminar of Computer Graphics* (2007).

[RH01] RAMAMOORTHI R., HANRAHAN P.: An efficient representation for irrandiance environment maps. *SIGGRAPH 2001* (2001), 497–500.

[RHS06] ROGER D., HOLZSCHUCH N., SILLION F.: Accurate specular reflections in real-time. *Computer Graphics Forum (Eurographics'2006) 25*, 3 (2006), 293–302.

[RSH05] RESHETOV A., SOUPIKOV A., HURLEY J.: Multi-level ray tracing algorithm. *ACM Trans. Graph. 24*, 3 (2005), 1176–1185.

[RTJ94] REINHARD E., TIJSSEN L. U., JANSEN W.: Environment mapping for efficient sampling of the diffuse interreflection. In *Photorealistic Rendering Techniques*. Springer, 1994, pp. 410–422.

[SC03] SANCHEZ-CRESPO D.: Inexpensive underwater caustics using Cg, 2003. http://www.gamasutra.com/features/20030903/crespo_01.shtml.

[Sch93] SCHLICK C.: A customizable reflectance model for everyday rendering. In *Fourth Eurographics Workshop on Rendering* (1993), pp. 73–83.

[Sch97] SCHAUFLER G.: Nailboards: A rendering primitive for image caching in dynamic scenes. In *Eurographics Workshop on Rendering* (1997), pp. 151–162.

[Sch03] SCHMIDT C.: Simulating refraction using geometric transforms, 2003. Master's Thesis, University of Utah.

[SGS06] STOLL C., GUMHOLD S., SEIDEL H.: Incremental Raycasting of Piecewise Quadratic Surfaces on the GPU. *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (2006), 141–150.

[SKAL05] SZIRMAY-KALOS L., ASZÓDI B., LAZÁNYI I.: Ray-tracing effects without tracing rays. In *ShaderX 4: Lighting & Rendering*, Engel W., (Ed.). Charles River Media, 2005.

[SKALP05] SZIRMAY-KALOS L., ASZÓDI B., LAZÁNYI I., PREMECZ M.: Approximate ray-tracing on the GPU with distance impostors. *Computer Graphics Forum (Eurographics '05) 24*, 3 (2005), 695–704.

[SKHBS02] SZIRMAY-KALOS L., HAVRAN V., BENEDEK B., SZÉCSI L.: On the efficiency of ray-shooting acceleration schemes. In *Proc. Spring Conference on Computer Graphics (SCCG)* (2002), pp. 97–106.

[SKM98] SZIRMAY-KALOS L., MÁRTON G.: Worst-case versus average-case complexity of ray-shooting. *Journal of Computing 61*, 2 (1998), 103–131.

[SKSS06] SZIRMAY-KALOS L., SZÉCSI L., SBERT M.: GPUGI: Global illumination effects on the GPU. In *EG2006 Tutorial* (2006).

[SKSS08] SZIRMAY-KALOS L., SZÉCSI L., SBERT M.: *GPU-Based Techniques for Global Illumination Effects*. Morgan and Claypool Publishers, San Rafael, USA, 2008.

[SKU08] SZIRMAY-KALOS L., UMENHOFFER T.: Displacement mapping on the GPU - State of the Art. *Computer Graphics Forum 27*, 1 (2008).

[SMS01] SCHIRMACHER H., MING L., SEIDEL H.-P.: On-the-fly processing of generalized lumigraphs. *Eurographics 20*, 3 (2001), 165–173.

[SP07] SHAH M., PATTANAIK S.: Caustics mapping: An image-space technique for real-time caustics. *IEEE Transactions on Visualization and Computer Graphics (TVCG) 13*, 2 (2007), 272–280.

[SSK07] SHEVTSOV M., SOUPIKOV A., KAPUSTIN A.: Ray-Triangle Intersection Algorithm for Modern CPU Architectures. In *Proceedings of International Confence on Computer Graphics and Vision* (2007).

[Sta96] STAM J.: Random caustics: natural textures and wave theory revisited. In *SIGGRAPH '96: ACM SIGGRAPH 96 Visual Proceedings* (1996), p. 150.

[Sze06] SZECSI L.: The hierarchical ray-engine. In *WSCG Full Paper Proceedings* (2006), pp. 249–256.

[Tha00] THATCHER U.: Loose octrees. In *Game Programming Gems* (2000), Charles River Media.

[Thi08] THIBIEROZ N.: Robust order-independent transparency via reverse depth peeling in Direct3D 10. In *ShaderX 6: Advanced Rendering Techniques*, Engel W., (Ed.). Charles River Media, 2008, pp. 211–226.

[TS00] TRENDALL C., STEWART A.: General calculations using graphics hardware, with application to interactive caustics. In *Rendering Techniques 2000* (2000), pp. 287–298.

[TS05] THRANE N., SIMONSEN L.: A comparison of acceleration structures for GPU assisted ray tracing, 2005. Master's thesis, Univ. of Aarhus, Denmark.

[UPSK07] UMENHOFFER T., PATOW G., SZIRMAY-KALOS L.: Robust multiple specular reflections and refractions. In *GPU Gems 3*, Nguyen H., (Ed.). Addison-Wesley, 2007, pp. 387–407.

[UPSK08] UMENHOFFER T., PATOW G., SZIRMAY-KALOS L.: Caustic triangles on the GPU. In *Computer Graphics International, 2008* (2008), pp. 222–228.

[Wal04] WALD I.: *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University, 2004. Available at http://www.mpi-sb.mpg.de/~wald/PhD/.

[Wat90] WATT M.: Light-water interaction using backward beam tracing. In *SIGGRAPH '90 Proceedings* (1990), pp. 377–385.

[WB05] WIMMER M., BITTNER J.: Hardware occlusion queries made useful. In *GPU Gems 2*. Addison-Wesley, 2005, pp. 91–108.

[WBS03] WALD I., BENTHIN C., SLUSSALEK P.: Interactive global illumination in complex and highly occluded environments. In *14th Eurographics Symposium on Rendering* (2003), pp. 74–81.

[WD06a] WYMAN C., DACHSBACHER C.: *Improving Image-Space Caustics Via Variable-Sized Splatting*. Tech. Rep. UICS-06-02, University of Utah, 2006.

[WD06b] WYMAN C., DAVIS S.: Interactive image-space techniques for approximating caustics. In *Proceedings of ACM Symposium on Interactive 3D Graphics and Games* (2006).

[Whi80] WHITTED T.: An improved illumination model for shaded display. *Communications of the ACM 23*, 6 (1980), 343–349.

[WIK*06] WALD I., IZE T., KENSLER A., KNOLL A., PARKER S.: Ray tracing animated scenes using coherent grid traversal. *ACM Transactions on Graphics (TOG) 25*, 3 (2006), 485–493.

[Wil78] WILLIAMS L.: Casting curved shadows on curved surfaces. In *Computer Graphics (SIGGRAPH '78 Proceedings)* (1978), pp. 270–274.

[Wil01] WILKIE A.: *Photon Tracing for Complex Environments*. PhD thesis, Institute of Computer Graphics, Vienna University of Technology, 2001.

[WK06] WÄCHTER C., KELLER A.: Instant ray tracing: The bounding interval hierarchy. In *EuroGraphics Symposium on Rendering* (2006), pp. 139–149.

[WKB*02] WALD I., KOLLIG T., BENTHIN C., KELLER A., SLUSSALEK P.: Interactive global illumination using fast ray tracing. In *13th Eurographics Workshop on Rendering* (2002), pp. 15–24.

[WMG*07] WALD I., MARK W., GÜNTHER J., BOULOS S., IZE T., HUNT W., PARKER S., SHIRLEY P.: State of the art in ray tracing animated scenes. *Eurographics 2007 State of the Art Reports* (2007).

[WMK04] WOOD A., MCCANE B., KING S.: Ray tracing arbitrary objects on the GPU. In *Proceedings of Image and Vision Computing New Zealand* (2004), pp. 327–332.

[WMS06] WOOP S., MARMITT G., SLUSALLEK P.: B-KD Trees for Hardware Accelerated Ray Tracing of Dynamic Scenes. In *Proceedings of Graphics Hardware* (2006), pp. 67–77.

[WS03] WAND M., STRASSER W.: Real-time caustics. *Computer Graphics Forum (Eurographics '03) 22*, 3 (2003), 611–620.

[WSE04] WEISKOPF D., SCHAFHITZEL T., ERTL T.: GPU-based nonlinear ray tracing. *Computer Graphics Forum (Eurographics '04) 23*, 3 (2004), 625–633.

[WSS05] WOOP S., SCHMITTLER J., SLUSALLEK P.: RPU: a programmable ray processing unit for realtime ray tracing. *ACM Transactions on Graphics 24*, 3 (2005), 434–444.

[WTP01] WILKIE A., TOBLER R. F., PURGATHOFER W.: Combined rendering of polarization and fluorescence effects. In *Proceedings of the 12th Eurographics Workshop on Rendering Techniques* (London, UK, 2001), Springer-Verlag, pp. 197–204.

[WW08] WEIDLICH A., WILKIE A.: Realistic rendering of birefringency in uniaxial crystals. *ACM Transactions on Graphics 27*, 1 (2008), 1–12.

[WWT*04] WANG L., WANG X., TONG X., LIN S., HU S., GUO B., H.-Y. S.: View-dependent displacement mapping. *ACM Transactions on Graphics 22*, 3 (2004), 334–339.

[Wym05a] WYMAN C.: An approximate image-space approach for interactive refraction. *ACM Transactions on Graphics 24*, 3 (2005), 1050–1053.

[Wym05b] WYMAN C.: Interactive image-space refraction of nearby geometry. In *GRAPHITE'05* (2005), pp. 205–211.

[Wym08] WYMAN C.: Hierarchical caustic maps. In *SI3D '08: Proceedings of the 2008 symposium on Interactive 3D graphics and games* (New York, NY, USA, 2008), ACM, pp. 163–171.

[YYM05] YU J., YANG J., MCMILLAN L.: Real-time reflection mapping with parallax. In *Proceedings of the Symposium on Interactive 3D Graphics and Games* (2005), pp. 133–138.

[ZHWG08] ZHOU K., HOU Q., WANG R., GUO B.: *Real-Time KD-Tree Construction on Graphics Hardware*. Tech. rep., Microsoft Research Asia, 2008.