Reflections in Real-Time Applications

Pascal Dario Hann*
Technical University Vienna

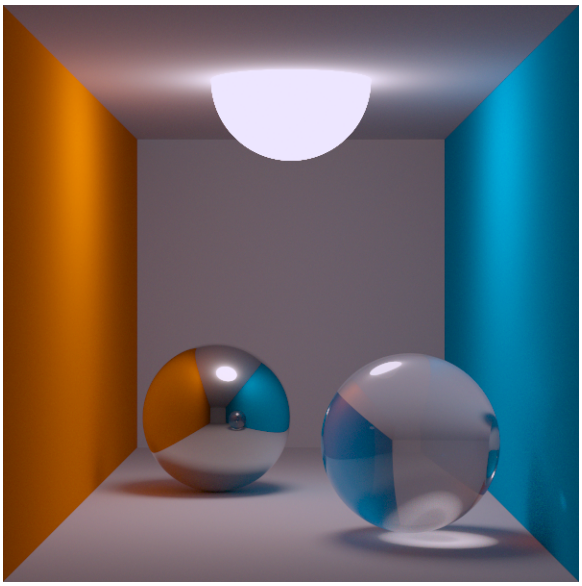

Figure 1: A rendered scene showing different effects of light reflection, like mirror reflections, indirect illumination and soft shadows. Reprinted from [Wikimedia 2017]

Abstract

The following Report provides a brief summary of a variety of techniques used to incorporate reflections into real-time applications. The techniques focused on are: Planar Reflection, Environment Mapping, Precomputed Radiance, Virtual Point Lights, voxel-based algorithms, Screen Space Reflections and real-time ray-tracing with NVIDIAS RTX technology.

Keywords: reflections, computer-graphics, real-time

1 Introduction

The process of light hitting a surface and getting bounced back is what is called ”reflection”. This interaction follows a set of rules given by the ”rendering equation” [Kajiya 1986], which provides a formula to calculate the emitted radiance of a given point in space. The variables that need to be taken into account are the incoming radiance, the radiance being emitted by the surface the point lies on and the material its made of, however the self-emitted radiance plays no role in reflection. We can see in figure 2 how a surface’s material affects it’s reflective property.

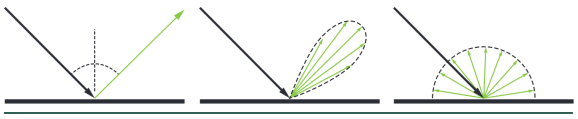

Figure 2: Mirror, glossy, and diffuse reflection. Left: the incoming light is reflected in a single direction off a mirrored surface. Middle: the surface is polished, such as brass, reflecting light near the reflection direction and giving a glossy appearance. Right: the material is diffuse or matte, such as plaster, and incoming light scatters in all directions. Adapted from [Haines and Akenine-Möller 2019]

As illustrated, perfectly smooth surfaces reflect a single incoming ray of light into a single outgoing one. The rougher the material gets the more the incoming light gets scattered in different outgoing directions. The human eye perceives the sharp ”specular” reflections created by smooth surfaces as mirrored scenery. The scattered ”diffuse” reflections created by rough materials are often forgotten when talking about reflections but the indirect illumination these provide make up an essential part of a real scenes total illumination, greatly affecting it’s look. Therefore, emulating reflection in computer applications is vital in order to achieve realistic scenes. This report summarises some of the techniques employed for this task and compares their different advantages and disadvantages.

*e-mail: e01633018@student.tuwien.ac.at

2 Ray-Tracing

The intuitive approach to calculate reflections in computer applications is to simulate the physical behavior of light by shooting light rays from the light-sources into the scene and following them along every bounce they make. This method of following a ray is called ”ray-tracing”.
Numerous terms describing forms of ray-tracing are used nowadays and it should be noted that no global standard terminology exists. Many terms like ”ray-tracing”, ”ray-marching”, ”path-tracing” and so on are often used interchangeably or with different meanings throughout the literature. To build a common ground for the sake of this report, the terminology used will be summarized here: ”ray casting” is the process of finding the closest intersection of a ray with scene geometry. A ray can be cast, for example from a light-source, and is followed until it hits it's first intersection point. ”ray-tracing” [Cook et al. 1984] is the action of recursively casting rays. When a ray is cast and hits it's first intersection, a new ray is cast from that point to the next intersection and so on until a termination criteria is met. The direction a new ray is cast from an intersection point can be set differently, correspondent to what one wants to achieve through ray-tracing. For example, in context of reflection computing, a ray might be recast along a reflection vector calculated from the incoming ray and the surface normal of the hit-point. Using ray-tracing, rays can be shot from a light-source to calculate the radiance they contribute to each hit-point.
As explained in the introduction, depending on a surface's material light-rays that hit it would need to be reflected into multiple new rays that again would need to be traced themselves. In addition an infinite amount of light-rays would need to be traced to simulate real world illumination. The calculation of this complex light transport would need an infinite amount of time which not only renders it unsuitable for real-time applications but for any computer application at all. In order to still use ray-tracing for the computation of reflections in such applications limitations need to be set. For instance, rays that only affect parts of the scene outside of the cameras frustum do not need to considered. To address this, instead of tracing rays from the light-sources through the scene, the process can be reversed, tracing rays from the camera back to the light-sources. When a light-source is hit by a ray, light is computed backwards through every hit-point the ray encountered contributing the initial light of the light-source plus the reflected light by the hit-points.
A method called ”path-tracing” [Kajiya 1986] addresses the problem of generating multiple rays at hit-points, by only allowing one single ray to be recast and in a useful direction on top, resulting in the name-giving path from start to finish of the traced ray. Blending multiple paths for a pixel gives an accurate estimation of the pixel's radiance with improving quality the more paths are traced. Modern ray-tracers often use more than one ray or path per pixel and use a randomised ”monte carlo” algorithm with a probability density function(PDF) to determine ”useful” directions for recast rays. When the PDFs used are non-uniform this is called ”importance-sampling” and when using number-theoretic methods rather than random number generators the algorithm is called ”quasi-monte carlo”. When dealing with scene non-boundary-based scene representations where computing ray intersections gets more complex, ”ray-marching” can be used. It follows a ray in steps, ”marching” along its direction. The steps can be regular or of differing size. Please note that the terminology detailed here is oriented on the one given in [Haines and Shirley 2019].
Even with limitations ray-tracing stays a costly process. The number of rays per pixel than can be traced even on modern hardware is very limited and when dealing with a small frame budget less costly techniques are needed. Some of these will be discussed in the following sections.

3 Planar Reflection

One common approach to produce reflections in computer applications, besides ray-tracing, is rendering the scene another or even multiple times from a different point of view to gain the light information of the environment that needs to be reflected. The technique making most direct use of this strategy is called ”Planar Reflection” [Diefenbach 1996]. Imagine a scene where the camera looks at a reflective plane. To achieve accurate reflections on this surface one simply needs to reflect the camera at the plane to get a new camera looking through the surface area the original camera looks at. The reflective vector of any given point on that area corresponds to a matching pixel captured by the reflected camera. In addition only a clipping plane aligned with the reflective surface needs to be applied to the reflected camera to solely capture objects in front of it. Now the new camera can be used to capture the reflections of the plane, as illustrated in figure 3.

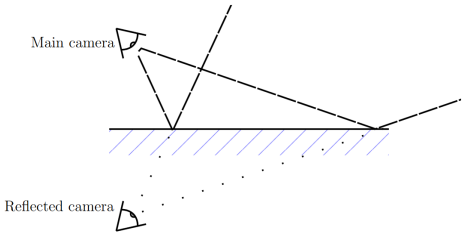

Figure 3: Diagram showing the setup for Planar Reflection. A secondary camera is placed in the scene, reflected about the plane. The viewing frustums of both cameras are shown as dashed lines. Reprinted from [Toft 2017]

This principle can be applied in different ways, one being a texture mapping approach, where the calculated reflections are stored in a special texture which can then be applied to the reflector during rendering. This method is rather straight-forward but requires a separate texture for every reflector in a given scene. Another way is to make use of the so called ”stencil buffer” and a technique called ”stencil testing” as described in [Diefenbach 1996]. In essence, stencil testing allows labeling pixels during rendering, storing that information in the stencil buffer. This way pixels can be masked during further rendering passes, allowing to apply reflections generated by Planar Reflection to be rendered directly to the frame buffer by applying a mask around the corresponding reflector. This eliminates the need for extra texture resources. Furthermore, stencil testing can also help with another problem multi-pass approaches pose: reflections of reflections. As the reflections are gathered from a separate rendering-pass, other reflectors in the scene whose reflections have not been calculated during that pass will not show up in the result. This can only be solved with recursively rerendering the scene requiring a multitude of rendering passes increasing with the number of reflectors and depth of interreflection targeted. Masking reflective surfaces with stencil testing allows rendering only those during these recursive rendering passes increasing the process' efficiency. Still this approach gets impractical very fast when the number of reflectors in a scene gets too big. Therefore, Planar Reflection is best used for a limited amount of mirror-like surfaces in a scene, where accurate reflection is crucial. The technique is not well suited for computing the usually high number of diffuse reflectors in a common scene and should be combined with others better fitting to this task. In addition, this method only works well for planar objects, as more complex objects would need to be split up into planar surfaces to perform Planar Reflection for each of them. This would increase the number of needed rendering passes drastically, in the worst case of a curved object, a pass for every pixel would be

required. Thus another method for calculating reflections found for non-planar objects needs to be used. One such technique, similar in essence to Planar Reflection, will be discussed in the next section: ”Environment Mapping”.

4 Environment Mapping

”Environment Mapping” [Blinn and Newell 1976] [Miller and Hoffman 1984] describes the approach to store the environment of a reflective object in a special texture, the so called ”environment map”, so it can then be sampled, as seen in figure 4, during rendering to create reflections on said object. For this the 3D environment of the object needs to be projected onto a 2D texture to be able to sample it with the reflection vector of each point of the object during rendering.

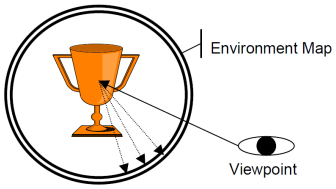

Figure 4: Sampling process of Environment Mapping techniques. Reprinted from [Kilgard 1999b]

This sampling process offers a more efficient approach compared to ray tracing techniques while still producing realistic results.

4.1 Spherical Environment Mapping

There are different approaches to environment mapping, mainly differing in the shape of the environment map. One approach uses a spherical shape, sampling reflections from a single image warped onto that shape, acting as the environment map. See figure 5 for a visual representation of a spherical environment map.

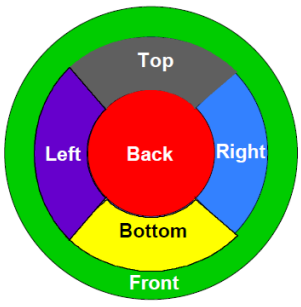

Figure 5: Spherical environment map - the entire outer ring represents one single point in space. Reprinted from [Kilgard 1999b]

This approach makes it easy to sample reflections from the map with the reflection vector of a given point in space, however, a rectangular 2D texture needs to be warped onto a spherical shape for this process, resulting in distortions. The most noticeable one being at the back of the object, as the point exactly at the back of the object is represented by the entire outer ring of the spherical environment map. In addition, ”Spherical Environment Mapping” techniques depend on the view-point as the reflections sampled of the pre-defined texture on the environment map will remain the same regardless of view-point. Even if they were not, the distortions on the back of an object mapped this way, already make this approach somewhat view-point dependent.

4.2 Paraboloid Environment Mapping

Another approach to environment mapping is the so called ”Dual Paraboloid Environment Mapping” technique, presented by [Heidrich and Seidel 1998]. This method uses two images instead of one and warps them onto two parabolas, one in front of the object, facing the viewer and one on the back, oriented in viewing direction. This way the technique achieves a view-independent environment map in addition to increased image quality compared to sphere mapping since two images are used instead of one. The downside of this process lies within the increase of needed memory space compared to. Furthermore, dual paraboloid environment mapping also requires image warping which lowers the effective resolution of the used images and is a costly calculation to perform.

4.3 Cubical Environment Mapping

The last form of environment mapping that shall be discussed here is ”Cubical Environment Mapping” [Greene 1986] [Kilgard 1999b]. As the name suggests this technique relies on a cube shape, using six images mapped to it’s inside as it’s environment map, also referred to as ”cube map”. See figure 6 for visual representations of a cube maps.

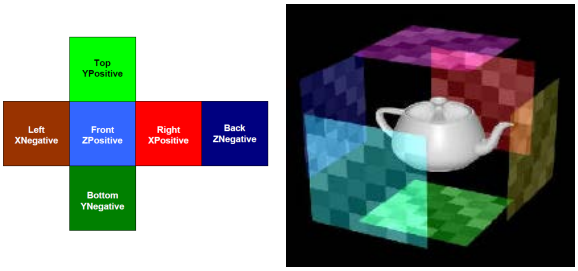

Figure 6: Cubical Environment Mapping - Each face of the cube map is aligned with either the positive or negative direction of one of the three axis of 3D space. Adapted from [Kilgard 1999b]

Synthesizing the environment information is more straight-forward than with the other two map shapes: one only needs to take six images from the origin of the cube through each of its faces. No image warping operations are required, which makes cube maps easier to generate than other forms, while also utilizing the full image resolution provided by the taken renders of the environment [NVIDIA 1999]. This balances the need of storing six images compared to just one or two like with the other two techniques, since the same effective resolution can be achieved with lower resolution images of the environment.
The sampling process, mapping the three dimensional reflection vector to the appropriate two dimensional texture coordinates on the right image of the environment map, is a bit more demanding than the sampling process of the other two environment mapping methods as it involves some conditional jumps to determine which

of the six images to sample. However, with the use of hardware support introduced with NVIDIAs GeForce 256 GPU in 1999 [Kilgard 1999b] and the accompanying updates for both the OpenGL and the Direct3D API - industry standards to this day - cubical environment mapping can now be implemented very efficiently. Cube Maps are like dual paraboloid environment maps view-point independent and can be generated very easily, even in real-time if necessary, since no image warping is involved.
Environment mapping in general shares the issue of interreflection with Planar Reflection. As explained in section [3], this effect can only be achieved with recursive generation an application of the environment maps, which requires multiple render passes increasing with the number of reflectors affected.

4.4 Environment Map Linking

Using separate environment maps for every object in a scene and regenerating all of them each frame is very costly depending on the number of objects. Since environment mapping methods relying on a cube or two parabolas as shapes are view-point independent they do not need to be updated every frame as long as the environment they map does not change. In addition instead of using separate environment maps for every object, maps can be generated only for distinct parts of a scene and objects can be linked dynamically to them during runtime, as explained in [Sbastien Lagarde 2012], drastically decreasing the number of maps needed. Different linking strategies will be detailed in the following.

4.4.1 Object-based

A possible approach is to simply link each reflective object to its nearest environment map. Static objects can be linked before runtime while dynamic objects need to be linked dynamically. This technique has three main disadvantages, however. Firstly, visible seams are possible at the verge between environment maps as a sudden change from one captured environment to another happens at these positions. This problem is especially visible when the environment changes at a high frequency between environment maps. The second problem is that objects that lie within multiple environments, must be split so they can be linked to the respective environment map, adding structural constraints to an application. Walls, floors and ceilings are frequently prone to this problem. The third problem is linking dynamic reflective objects. A jump between environment maps can be visible when a dynamic object switches environment map. The first and third problem can be somewhat compensated by blending adjacent environment maps in a weighted manner, but because there is still a relatively big number of environment maps in use with this method, and blending needs to be performed individually for objects in different positions this is a rather costly solution and it does not solve those problems completely.

4.4.2 Zone-based

Another way is to divide a level into different zones which are typically wider than local environment areas used in the object-based strategy. Each zone then gets assigned one single environment map, which all objects are linked to when the viewer enters that zone. Resulting out of their global positioning, zone environment maps do only contain global environmental information without the local details.
This strategy produces no seams because all objects share the same environment map at every given time, however, jumps can be noticed when the camera crosses from one zone to another. Weighted blending between the zone maps can once again help to solve this problem and because this does not have to be done individually for every object in a different position, but only for adjacent zones, the number of environment maps that need to be blended is relatively small, turning this solution into a valid option in this scenario. The main disadvantage here is the lack of detail in the reflections as all objects are always linked to only one global environment map. Local environmental details, like a local light for instance, are not reflected.

4.4.3 Global and local without overlapping

Adding onto zone-based environment mapping, another approach introduces a small number of local environment maps on top of the global zone maps, preferably in areas with high local environmental detail. All objects are linked to the zone map of the zone the camera is positioned in per default, but when it comes in range of a local environment map all objects are linked to that one instead. The local maps are placed with enough space between them so the viewer must always pass through an area outside of any local environment map's influence, returning all reflective objects linked to the global zone map, before it can enter another local environment map's influence volume again. This way seams are prevented and the number of environment maps is still small enough to blend them to counteract sudden jumps at changing points in the level. With this method a higher level of detail than the one produced by pure zone-based environment mapping alone can be achieved, while still benefiting from it's advantages. However, it also poses a new problem: because every object is linked to a local environment map once the camera enters its influence, distant objects also falsely reflect the local environment. This effect gets countered at least to some degree by the blending with the global zone maps.

4.4.4 Global and local overlapping

Allowing multiple environment maps, global and local, to overlap each other's influence volume creates yet another approach. Here environment maps only affect things inside of their reach, which means this method can only be used with a deferred rendering architecture which is explained in chapter 9 of [Pharr and Fernando 2005] - or something similar, because one would have to split all objects multiple times for all the areas of influence and their overlapping parts too in a traditional rendering architecture. It also means multiple different maps can be active at the same time like with the object-based method sharing the possibility of visual seams in reflected images at the edge of influence areas. Having the possibility of multiple environment maps affecting different objects brings more accurate reflections, but, as explained, comes with a restriction in rendering architecture.

4.4.5 Point of interest based

The last environment map linking strategy that shall be mentioned here was presented in [Sbastien Lagarde 2012]. It uses multiple local environment maps like the object-based method but introduces a point of interest, which can be the camera, a player, or something else depending on context. In contrast to the object-based approach, this strategy does not link objects to the environment map nearest to them but instead blends together the maps nearest to the point of interest - weighing them the nearer they are - and links all viewable reflective objects to the resulting environment map. This produces no seams because only one environment map is applied to all objects while still retaining detailed local information about

the environment. While sudden jumps can be visible when the least contributing map - meaning the one farthest away from the point of interest - changes, this can be avoided by cleverly placing the local environment maps with this problem in mind. Point of interest based environment mapping shares the problem of distant objects reflecting inaccurate environment information with the combined global and local approach without overlapping. Lagarde describes several methods to fight this obstacle in [Lagarde 2012]. One approach is also good practise for any of the linking strategies involving local environment maps mentioned, namely implementing the option for objects to override the system with an environment map specific to that object, whose center should be aligned with the origin of that map. This way special objects, like for example perfect mirrors, on which the shortcomings of a linking method would be especially noticeable.

4.5 Parallax Corrected Environment Mapping

A general negative effect of having only a set amount of environment maps in central locations instead of one map per reflective surface is posed by the sampling process, as it still uses the reflection vector calculated from the view vector - from camera to object - for querying the environment map. This assumes the map's origin still alignes with the reflection point, which, however, it most certainly does not in this scenario, leading to retrieval of the wrong location of the environment, as can be seen in figure 7. This false assumption creates a displacement of the reflected environment, which is called parallax effect. This is not only a problem when environment map and mapped object are not aligned but also when environment mapping large surfaces, like a floor or a wall, because a reflection vector calculated somewhere away from the center of that object does not point at the same geometry as it does when moved to the center, which is essentially what happens when sampling an environment map. The environment mapped in the environment map would have to be infinitely far away to not produce this problem as Chris Brennan explains in [Brennan 2002].

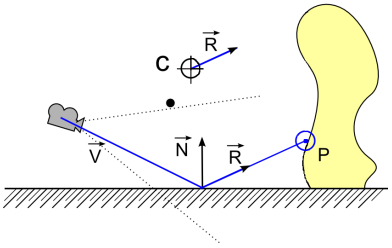

Figure 7: The Reflection vector "R" does not point at the same point "P" of the geometry when moved to the environment map's origin "C" after calculating it with the view vector "V" from the camera. Adapted from [Lagarde 2012]

In order to correct this error, information about the actual geometry of the environment surrounding the environment map is required to determine where a given reflection vector intersects with that geometry. Knowing that point, the vector from the origin of any environment map to that position can be used as the new parallax corrected reflection vector, as illustrated in figure 9.
Since determining which object in a scene intersects a reflected ray is not an easy task and would involve some sort of ray-tracing or similar method which would be costly, an easier, more cost-efficient approach is to approximate the geometry. The hallway of a building for instance could be approximated by a box-shape or some other primitive that matches the environment, like a sphere or a cube. This way the reflection vector must intersect with the approximated geometry at some point which therefore can be calculated efficiently.

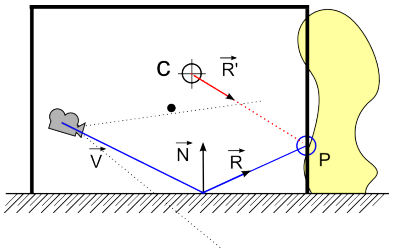

Figure 8: Instead of simply using the reflection vector "R", the intersection with the geometry's approximation "P" is calculated and the vector from the environment map's origin "C" to P is used as new parallax corrected reflection vector. Reprinted From [Lagarde 2012]

The quality of the corrected reflections obtained this way varies with the accuracy of the approximation. The more complex an environment's geometry is the harder it gets to approximate with a primitive shape, resulting in less precise reflections. While more complex approximations can be chosen, this comes with an increased demand on hardware.
Toft describes a way to implement parallax correction using the depth buffer, to get the information about the scenes geometry needed, in [Toft 2017]. The depth buffer stores the distance of geometry along a camera's z-axis in a texture, where every pixel holds the value for exactly one point of that geometry. Using ray-marching [2] this technique can produce quite precise results, again at the price of performance. This method has similarities to a technique called "screen space reflection" which also makes use of the depth buffer and will be highlighted later in [7.2].

Like Planar Reflection, environment mapping can be used to accurately simulate diffuse as well as specular reflections in dynamic scenes. In contrast to Planar Reflection, its better suited for complex or rounded objects as for big planes. Interreflection is difficult to achieve with both techniques and environment mapping requires an extra amount of texture storage. Linking strategies for limiting the amount of maps used and different scene approximations varying in accuracy for Parrallax Correction each trade quality against performance.

5 Precomputed Radiance

Methods that can calculate reflected light accurately offline exist, so one intuitive approach is to simply precalculate scenes beforehand and store lighting information in textures, which can then be applied during real-time rendering. Ray-tracing techniques, path-tracing for example can be used or another method called "radiosity" [Goral et al. 1984], a recursive algorithm that makes use of the "finite element method" and is inspired by thermal engineering techniques. As reflections are precomputed here, this technique comes, however, not only with the cumbersome and time-consuming pre-runtime calculation steps and a high amount of texture storage needed, but on top of that, scenes rendered this way have to be completely static and do not support any dynamic objects or changes in light-sources, leaving the only interactivity still possible: dynamically changing the view-port in walk-through applications.

5.1 Precomputed Radiance Transfer

Somewhat better results concerning interactivity can be achieved with "precomputed radiance transfer" techniques [Sloan et al. 2002], which simplify the rendering equation [1] by making the following assumptions: Firstly, like the radiosity technique all reflectors in the scene are assumed to have perfect "lambertian" diffuse materials. This simplifies the rendering equation because such materials scatter light equally in all outgoing directions. Secondly, all objects in the scene do not emit any light, removing that part of the rendering equation as well. Thirdly, light-sources are assumed to be infinitely far away which makes the incoming light direction at a given point independent of its position breaking down the rendering equation further. The next step is to break the simplified equation down into two parts: the lighting function which basically describes the incoming light and the transfer function which in return represents how a given surface reacts to that incoming light. Both these functions are then projected into a linear combination of base functions which is done in a precomputation step and stored afterwards in an appropriate form like "spherical harmonics", so that the approximated rendering equation resulting can be calculated efficiently as a simple dot product during real-time rendering. An understandable overview of this technique is given in [Slomp et al. 2006]. Precomputed radiance transfer can normally only calculate diffuse reflections, however glossy ones can be achieved with some additions. The technique supports any number of light-sources without any increase in calculation and interreflection. When spherical harmonics are used as base functions even some dynamic positional changes of the light-sources are possible, however, dynamic scenes stay impossible to calculate with this technique and a time consuming precomputation step is needed.

6 Virtual Point Lights

A family of "virtual point light" algorithms makes use of an idea originating from Alexander Kellers technique "instant radiosity" [Keller 1997] to overcome the limitations of complex precomputation for computing diffuse reflections in real-time. The essential approach is to cast a finite number of rays into the scene and generate virtual point lights (VPLs) where these hit the geometry. These virtual point lights simulate the indirect light being reflected by the reflectors in the scene and can be rendered easily with traditional rendering methods. In the original instant radiosity method an image with shadows, usually in the form of a "shadow map", is rendered for every VPL separately and the final result is obtained by summing these up with an accumulation buffer. The number of particles to use for VPL generation can be varied with the computational power at hand, making this technique easily scalable. The algorithm described by Keller uses the following rule to limit the number of bounces made by the rays cast into the scene: Since the diffuse reflectivity of real scenes only differs minimally from the mean scene reflectivity "p" - a number between 1 and 0 - the radiant density of the scene can be approximated by stating that p multiplied with the number of starting rays "N" yields the number of rays "pN" that will be reflected and cast into the scene again, while the rest will be absorbed. Of course, in the next step the new number of total rays will be multiplied with p again, decreasing it further. This process is iterated through until no more particles are left and at every bounce a ray makes, a new VPL is created, limiting the total number of VPLs "M" to:

$$M < \sum_{j=0}^{\infty} p^j N = \frac{1}{1-p} N$$

The initial rays are cast into the scene in a random, also called "monte carlo" fashion, beginning at the light-source or light-sources. Subsequent cast directions for reflected rays are obtained by following the rules of diffuse light scattering. The exact mathematics can be read in [Keller 1997]. To account for the different colors of the objects in the scene, the particles are assigned the color of the light-source they are coming from initially, which is then attenuated with the color of every object that is hit. The VPLs generated get the current color of the ray they come from. This way every hit surface contributes to the color of the simulated indirect light. The technique of casting rays into a scene and bouncing them from objects they hit is a form of ray-tracing. The advantage of this form is that while the other forms start at the view-point sending rays into the scene and recursively calculate the radiant power of every point they hit with the ray reflected of that point, the ray-tracing technique employed in instant radiosity does not need to concern itself with a point it hits and generates a corresponding VPL for beyond that interaction. Because of this. Keller also describes a way to use instant radiosity efficiently with interactive view-point changing walk-through applications: instead of casting all rays anew in every frame, only one is, the images created from the VPLs generated by that one traced ray are accumulated and that image is then stored. "N" images - depending of the computational power available - are stored with their time of generation and then accumulated for the current rendered frame. When a new path is completed the oldest image is replaced by the new one, implicitly performing temporal antialiasing.
So in theory instant radiosity can be used to compute multi-bounce diffuse reflection in real-time without precomputation even for dynamic scenes, but the technique comes with its own set of problems: Because of the monte carlo ray-tracing and limited number of possible VPLs, dynamic objects can show awkward behavior due to a lack of temporal coherence in the positioning of the VPLs and the fact that temporal antialiasing can not be done the same way as described for walk-through applications with dynamically moving objects, because older generated frames would not be accurate for newly positioned objects. In addition, also because the number of particles has to be limited and the cast directions are not dependant on the view-point with classic instant radiosity, it can happen that a portion of the generated VPLs is placed somewhere so they do contribute to parts of the scene the viewer can see, while too little VPLs are generated in viewable reach, resulting in poor visuals. To solve this issue a variation of the original technique called "bidirectional instant radiosity" was presented in [Segovia et al. 2006], which introduces a second type of VPLs that are cast not from a light-source but from the camera. This process called "reverse instant radiosity" makes use of the fact that light propagates linearly, to make sure reverse VPLs are placed at points in the scene that can actually illuminate geometry seen by the camera, by allowing cast particles from the camera to only bounce off once. The problem with these inverse VPLs is that their density and radiant power need to be calculated separately. To get radiant power of an inverse VPL several random rays have to be cast from the inverse VPL, then some traditional VPLs need to be created to finally cast "shadow rays" between those two for the resulting radiance. See figure 9 for an illustration of this algorithm.

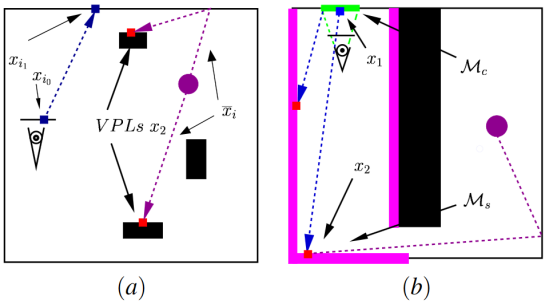

Figure 9: a) Standard instant radiosity (IR). (b) Reverse IR. Green points "x1"("Mc") are points visible from the camera. Magenta points "x2"("Ms") are points visible from "Mc" and the only possible VPL locations for the given point of view. Allowing only one single bounce gives "Ms". Connecting "Ms" to the light sources gives the outgoing radiance field of the sampled VPLs. Reprinted from [Segovia et al. 2006]

This is a costly operation and should be avoided if possible, so a method is proposed which creates an even number of inverse and traditional VPLs, determines which of them have most impact on the viewable area, selects only those and uses them for the final render. This method produces more robust results in situations where traditional instant radiosity could not, but it also adds to the cost of the algorithm.
To increase efficiency [Laine et al. 2007] presented another derivation of instant radiosity: "incremental instant radiosity" . This technique is based on reusing VPLs to save costly re-generation. To reuse VPLs, every frame the validity of every VPL is determined by their directional distribution measured from the light-source they originated from and if they are still directly visible by the originating light-source.

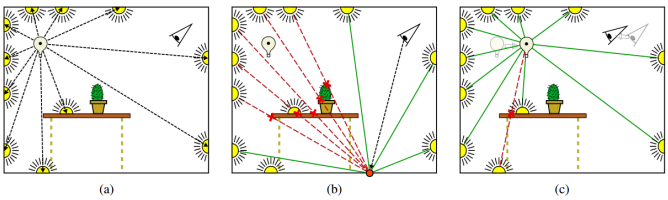

Figure 10: VPL Validation - a) Paths are traced from a light-source to generate VPLs at reflection/absorption points. b) A given points radiant power is accumulated from all VPLs during rendering. c) The light-source has moved, instead of recreating all VPLs, old ones are validated. Only one VPL can no longer be seen by the light-source and has to be recreated, the others can be reused. Potential camera movement does not affect this step. From [2007]

Ideally the directional distribution of VPLs from their light-source should be the same as the directional intensity distribution of that source which would result in every VPL representing an equal fraction of the sources outgoing power. Incremental instant radiosity tries to stay as close to ideal VPL distribution as possible by removing invalid ones and replacing them with new ones. Costly re-generation is only performed for new VPLs, reusing previously generated, still valid VPLs. Incremental instant radiosity might have to be combined with bidirectional instant radiosity to account for the visibility problems mentioned before. Overall this technique can provide a substantial boost in efficiency for instant radiosity but it depends largely on the light and view-port movement in an application. When these factors change too fast, no or near to zero VPLs can be reused, nullifying the benefits of this approach.
[Walter et al. 2005] developed a different way of saving computational power that can be applied to instant radiosity by clustering VPLs into "light cuts". A global "light-tree" - a binary tree containing all VPLs - is first generated. The leafs of the tree each represent exactly one VPL and every node of the tree can be used as a cluster for its children. The tree is built by grouping VPLs together based on their spatial proximity and directional orientation as similarity criteria. Instead of accounting for every VPL in a scene each cluster or "light cut" gets assigned a single representative VPL which then is used during rendering. Clustering the VPLs has the added benefit that it counters the temporal incoherence produced by VPL redistribution explained earlier to some extent. Which clusters shall be used is chosen dynamically each frame by a specified error criteria, shifting the speed limitations of instant radiosity approaches from the amount of VPLs in a scene to the amount of VPLs used per pixel, improving scalability. This technique can greatly reduce the number of VPLs that have to be accounted for during rendering, increasing efficiency of an instant radiosity algorithm.
Best frame-rates for truly interactive dynamic scenes in real-time applications with VPL approaches have been achieved with algorithms that limit VPL generation to image-space. [Nichols et al. 2009] presented a hierarchical approach for so called "image-space radiosity". They use "reflective shadow maps"(RSM) [Dachsbacher and Stamminger 2005] for their VPL sampling process. Shadow maps in general are essentially depth buffers rendered from a light-source. Reflective Shadow Maps in addition store the reflected diffuse light of the light-source the map is generated for, at every pixel of the shadow map. Building onto the splatting technique from [Dachsbacher and Stamminger 2006] to sample the RSMs for VPLs, Nichols image space radiosity algorithm uses a stencil buffer to mark parts of image-space with high frequency details to perform a rendering on multiple resolutions: finer for high frequent parts of the scene and more roughly for low frequent ones. This image-sapce approach reduces the cost of rendering significantly and is combined with hierarchical VPL clustering similar to the light cuts technique to further increase performance and counteract temporal inconsistency. All efforts to save computational power together make for a relatively fast, efficient rendering technique for diffuse reflections that can be used effectively in interactive, dynamic, real-time applications. The biggest disadvantage of only sampling image-space VPLs from the RSM is obviously that off-screen information is not taken into account, so renders created this way are just a rough approximation of the whole scene.
Bottom line, VPL-based techniques offer a method to achieve multi-bounce diffuse reflections in dynamic real-time environments. Since the number of rays that can be traced is limited by the frame budget of an application, the VPLs simulation is only a rough approximation of the reflections in a scene. It is not accurate enough to achieve specular reflections. Furthermore as described, temporal inconsistency is a problem with VPL-based algorithms and temporal antialiasing methods need to be employed.

7 Real-Time Ray-Tracing

Some techniques that employ ray-tracing in some form in real-time were already mentioned up to this point. A method was discussed that used ray-marching for intersection testing against a depth buffer for parrallax corrected environment mapping [4.5] and Virtual Point Light algorithms [6] also use ray-tracing for VPL generation. The techniques explored in the following subsections further focus on ray-tracing to achieve reflections. In general one of the biggest problems when using ray-tracing is intersection testing. With traditional scene representations, namely polygons subdivided

in primitives, triangles most commonly, determining where a traced ray hits scene geometry is a non-trivial task. The computation that is needed to calculate this information are one of the biggest limitations of ray-tracing algorithms performance. To solve this problem different scene representations have been explored with the goal to make intersection testing more efficient.

7.1 Voxel-based algorithms

One family of algorithms relies on"voxels" as scene representation to efficiently test ray intersections against. A voxel can be described as a three dimensional pixel. Like pixels are ordered in 2D grids to form an image, 3D scenes can be divided into a 3D grid of regularly sized voxels, as one can be seen in figure 11, implicitly defining their spatial position. A voxel can hold different information: "binary" voxels can be represented by a single bit, where 0 means the voxel does not contain any geometry and 1 means it does. "Multi-value" voxels can store additional information like color, material properties or normals, much like 2D textures. Furthermore, "boundary" and "solid voxels" have to be differentiated, where the first only store object surface information, the latter also saves interior object properties.

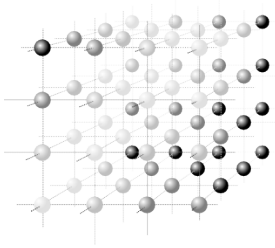

Figure 11: Voxel-grid containing color information. Reprinted from [Wikimedia 2015]

With voxel-based scene representations rays only need to be tested against scene geometry that lies within a voxel the ray traverses reducing intersection testing complexity. Prior to that, however, techniques have to be found to generate such a representation from a polygonal one, because most of the time scenes will not be modeled with voxels, resulting from industry modeling application functionality. This has to be done efficiently in real-time if voxel-based algorithms shall be used for dynamic scenes and should not involve time consuming, cumbersome, precomputational steps. Different strategies were developed to tackle this problem: An early approach was to use different settings for the near and far plane of an octagonal camera to sample different slices from the geometry that needs to be "voxelized" [Fang and Chen 2000]. Building onto that approach [Crane et al. 2007] used a geometry shader to intersect the scenes geometry with multiple regularly spaced planes to construct a 3D voxel-grid. [Schwarz and Seidel 2010] introduced custom "3D rasterizers" to leverage gpu power better than the conventional 2D rasterizer. Slicing algorithms unnecessarily sample the empty space around non-boundary voxels. An alternative approach based on "depth peeling" [Li et al. 2007] splits the scene into layers and renders the scene multiple times iterating through the layers starting at the camera and traversing through to the farthest layer from the camera. In every iteration, the current frame is compared to the depth buffer of the previous render, checking if a voxel lies in the current layer, by measuring how far away corresponding parts of the geometry are. The depth complexity of a scene is usually lower than the slices of a volume, resulting in less, but still multiple needed rendering passes. [Eisemann and Décoret 2006] observed that the conventional rendering pipeline already has to define a grid - the rendered image - and must visit every primitive of a scene during rasterization. Using that information they combined the depth buffer and a RGBA texture to voxelize a scene using simple bit-wise blending. This allowed for fast binary voxelization using only a single rendering pass and the authors later presented how to use the same method for solid voxelization [Eisemann and Décoret 2008]. [Thiedemann et al. 2011] propose a technique to achieve real-time multi-bounce diffuse reflection for dynamic scenes by using voxel-grids to represent scene geometry. Their voxelization method follows the same principle as depth peeling, but instead of dividing an object into multiple layers they store the world-space position of every point on the object in a "texture-atlas", also called "sprite-sheet". This is a special texture that can store 3D world-space coordinates by placing texels in specific locations of the texture. They can compute this texture with the information of a single render-passes depth-buffer. Then a special "voxelization-camera" is used to define the region that shall be voxelized. For each texel of the atlas texture a corresponding vertex is placed in the camera's frustum, projecting the stored world-space coordinates into the voxel-grids coordinate system. Each vertex corresponds to a voxel in the final voxel-grid. The voxel-grids resolution directly depends on the resolution of the atlas texture and because every valid texel - every atlas texture also has invalid texels, resulting from the mapping of the 3D world-space positions - corresponds to one placed voxel and one drawn vertex during voxelization, the resolution of the atlas texture also directly limits performance. This technique is a bounding voxelization algorithm as preferred for reflections, because interior voxels can be ignored for light reflection. To avoid revoxelization every frame, static objects of a scene are voxelized only once and just dynamic objects are revoxelized at runtime.
To compute the indirect illumination, resulting from reflected light, at a given point in space, several rays are shot into the hemisphere along the point's normal. The voxels along the ray's path are traversed until a voxel representing geometry is hit. The hit-point is then "back-projected" into a reflective shadow map of the scene's nearest light-source: the texel corresponding to the 3D position of the hit-point is sampled from the RSM. That way the reflected direct diffuse lighting on the hit-point is determined, which in return is used as indirect light on the original point where the rays where shot from, creating single-bounce diffuse reflection effects. To account for blockers between light-sources and hit-points, the texel queried from the RSM is only used if its position is not farther away from the hit-point than a specific error threshold. Otherwise direct lighting at the hit-point is assumed as 0, meaning it does not reflect any light on incident surfaces. As performance depends on how long a ray is traced through voxel-space, a limit length for shot rays is introduced, implicitly defining a sphere, with the limit length as its radius, around a given point, in which incident geometry can contribute to the indirect illumination of that point. Resulting from that constraint, only near field indirect lighting can be computed by this technique. See figure 12 for an illustration of this algorithm.

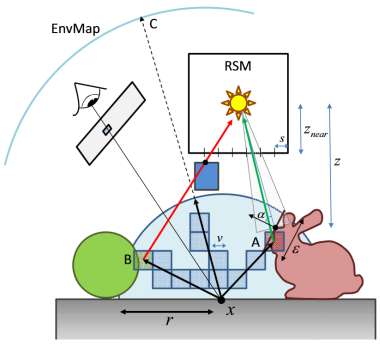

Figure 12: Near-field indirect light: To compute the indirect light at a point "x", several rays are shot in the upper hemisphere. The voxels along each ray are traversed until a hit-point is found. The hit-point is then back-projected into the RSM to obtain the direct radiance at this point. Indirect light is gathered from hit point "A", because it is visible from the light source: The distance from hit-point A to the RSM pixel position is smaller than the threshold. Hit-point "B" is in shadow of the direct light, because it's position is further away from the position stored in the RSM pixel than allowed by the threshold. If no intersection point is found within the search radius "r", the radiance can optionally be read from an environment map, to simulate directional occlusion (point "C") Reprinted from [Thiedemann et al. 2011]

With the technique from [Thiedemann et al. 2011] convincing diffuse single bounce reflection can be achieved in real-time applications with fully dynamic scenes. However, the reflection are limited to near-field geometry and noise and flickering artifacts plus temporal incoherence issues are present problems with this approach because the number of rays that can be used in real-time for computing reflections is limited and they are shot in a random fashion. Another voxel-based algorithm developed by [Crassin et al. 2011] replaces the voxel-grid scene representation used by the previous method with a hierarchical structure: a "sparse octree" [Laine and Karras 2010]. An octree is a tree data structure where every node has eight children. In computer graphics the root node usually represents a whole scene and it's children one eighth of it each and so on. "Sparse" octree nodes are only subdivided if they contain any information, in this scenario geometry, saving memory. The voxelization method used in this specific technique directly builds the voxel-octree without using any regular voxel-grids. To create it, the polygonal source geometry of the scene is rasterized three times, once along every 3D axis, using the traditional rendering pipeline. The view-port resolution for these renders is set to the maximum subdivision target for the octree and at least one fragment shader thread for each potential leaf node. These threads each traverse the tree from the node to its leafs in parallel, subdividing it if geometry is found in a current node, until the resolution is exhausted when the targeted maximum tree subdivision is reached. Like the previous method, only dynamic and semi-dynamic objects are revoxelized every frame while static parts of the scene get voxelized just once. The main clue of using a voxel-octree is that every node represents the light reflection behavior of its children. Complex calculations and interpolations are employed to store directional distribution functions, occlusion information in form of a percentage value of blocked rays, material information and other miscellaneous data appropriate in every node of the tree. The [Crassin et al. 2011] technique also makes use of ray-tracing to compute reflections, but in contrast to the previous approach they rely on the spatial and directional coherence of rays shot from a point in space into its facing hemisphere, to adapt their ray-tracing approach.

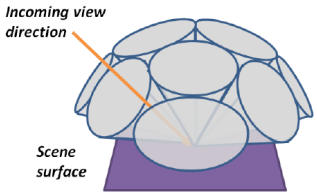

Figure 13: Because of the spatial and temporal coherence of rays shot from a point in space, they can be bundled into ray cones. Adapted from [Crassin et al. 2011]

Instead of tracing single rays, bundles of spatially and directional coherent rays are summarized into cones, as illustrated in figure 13, and only these are than traced, while making clever use of the hierarchical octree scene approximation. The principle is to step along a traced cones axis and gather samples from the octree at the level of subdivision correspondent to the current cone radius. The authors call this technique "voxel cone tracing". To achieve reflections with this voxel cone tracing algorithm, the incoming direct radiance from light-sources in the scene must be distributed correctly in the octree first. To get there, as a first step, a map similar to a reflective shadow map is rendered from a light-source. Every pixel of that map basically corresponds to one photon of light that needs to be bounced into the scene. The maps resolution should be set higher than the octree's maximal subdivision, to ensure that the photons can be splatted on the leafs of the tree without any visual gaps. After achieving accurate lighting information on the lowest level of the octree this way, the distribution to the higher levels is performed in a second step, the exact details of can be read in [2011]. This process is performed only when a light-source moves, while voxel cone tracing can use the stored distributed values efficiently otherwise. See figure 14 for an illustration of the algorithm.

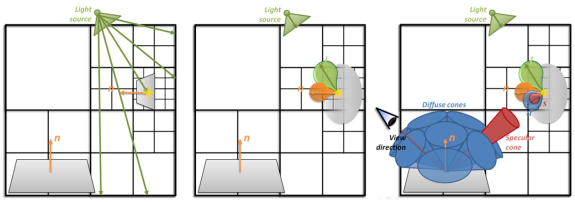

Figure 14: From left to right: Render from light-sources. Bake incoming radiance and direction into the lowest level of the octree - Distribute information to higher levels - use voxel cone tracing to compute reflections. Using a special cone along the reflected eye ray makes glossy reflection possible. Adapted from [2011]

This technique achieves convincing real-time multi-bounce reflection effects for truly dynamic scenes. Both diffuse and glossy reflections with indirect highlights are supported and the method's performance is nearly independent of the scenes geometric complexity because it operates entirely on the voxel-octree representation of it. Neither flickering artifacts nor noise is visible and the algorithm produces temporal coherent results, all due to tracing all cones subdividing a hemisphere, a point is facing, instead of single random shot rays. Although all this is possible, the algorithm is not as efficient as screen-space only techniques and because cone-tracing is not entirely accurate perfect specular mirror reflections are not possible.

7.2 Screen Space Reflection

Algorithms that operate primarily on screen-space have already been mentioned in this report: parallax-corrected environment maps that use screen-space depth information as a geometry approximation for correction [4.5] and image-space radiosity [6] were two examples of a family of techniques that use screen-space depth buffers as efficient scene representation. The family of screen-space algorithms in general uses the gained performance of operating on already provided - by traditional obstruction culling in standard rendering pipelines - depth information to employ forms of ray-tracing to achieve global reflection effects. [Sousa et al. 2011] proposed ray-marching the screen-space depth information for mirror-reflection in games.
[Wronski 2014] explains the problems of screen-space ray-tracing: Firstly, information outside the view-port is not available meaning that information is lacked if reflected rays exit the viewable area. Secondly, information on the back of objects is also not available because it can not be seen from the camera and thirdly, information about parts of the geometry that is obstructed by other geometry is also not available because, again, the camera can not see these parts. See figure 15 for an illustration of these problems.

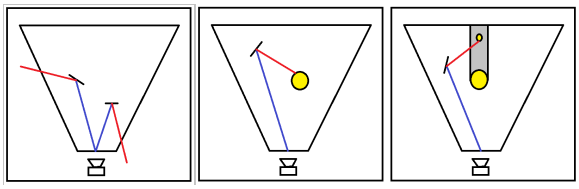

Figure 15: From left to right: rays are reflected outside the camera's frustum - information on the backside of objects when viewed from the camera is not available - information that is obstructed by an object is also not captured from the camera. Adapted from [Wronski 2014]

In general it is strongly adviseable to use some form of fall-back when ray-tracing screen-space reflection(SSR) to not only gain accuracy but also to avoid visual artifacts resulting from missing information.
SSR ray-marching methods, in style of [Sousa et al. 2011], is performed in 3D with a limit to trace length. For every step along the ray the point is projected into 2D screen-space to test if it is behind the depth duffer, getting classified as a hit-point if it is. Because of the projection from 3D to 2D several pixels of the depth buffer are skipped while others get sampled multiple times. In the worst-case of a ray being shot directly parallel to the viewing direction, the same pixel may be sampled for every ray-marching step until the maximum traced length is reached. This being inefficient, [McGuire and Mara 2014] proposed an algorithm performed directly on the 2D depth buffer based on the digital differential analyzer(DDA) algorithm, the direct evolution from Bresenham's 2D line rasterization.
One way to overcome the missing depth information because of obstruction is to add multiple layers to the screen-space depth buffer. Methods to acquire these have been touched upon in the this report's part on voxelization [7.1] since a layered depth buffer is basically the same thing as a voxel-grid representation if the layers are spaced regularly, as illustrated in figure 16. Mentioned techniques like slicing or depth peeling can be used to achieve layers, but [Mara et al. 2014] propose a different approach to create a depth buffer with 2 layers or how they call it, a two layered "geometry-buffer"(g-buffer).

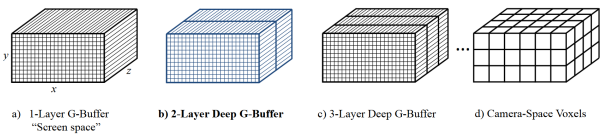

Figure 16: A continuum of data structures on the content of the homogeneous clip-space view frustum. Traditional G-buffers(a) have high "xy"-resolution, minimal "z"-resolution, and choose the closest surface to fill each voxel. Traditional voxels (d) have medium "xyz"-resolution and average surface properties within each voxel. Adapted from [Mara et al. 2014]

They use a single render-pass from the main camera to generate both layers simultaneously by using prediction techniques to approximate which parts of the geometry belongs in which layer. In the same paper the authors build onto a screen-space radiosity technique devised by [Soler et al. 2010], by sampling their layered g-buffer in a monte carlo fashion to compute fast approximate reflection effects, in similar fashion as reflective shadow maps. They use a confidence value to determine points which samples are not representative enough, because they lie on the back or side of an object and neither layer of the g-buffer could capture its depth information. For these points an interpolation between the screen-space radiosity and a fall-back solution like environment maps or voxel-representation is calculated. This method is more efficient for computing diffuse reflection effects than ray-tracing and can be combined with it to gain specular reflection effects as well.
In general SSR algorithms offer an efficient way to compute real-time multi-bounce reflections, but the lack of non-screen-space information can lead to visual artifacts and inaccurate visual effects. These techniques should therefore be used in combination with other reflection methods.

7.3 NVIDIA RTX Real-Time Ray-Tracing

In 2018 NVIDIA revealed their new line of "RTX" GPUs [NVDIA 2018] which are engineered specifically towards ray-tracing applications. Special ray-tracing cores are built into these GPUs providing hardware support suiting ray-tracing needs. On top of that Microsoft launched a new ray-tracing API "DirectX Raytracing" [Stich 2018], or DXR in short, simultaneously to leverage this new hardware-support for ray-tracing. DXR introduces a new ray-tracing pipeline with the option to program it through special ray-tracing shaders and functions. A "ray-generation" shader is used to start the pipeline, allowing for ray specification. "Intersection shaders" define the logic for intersecting rays with arbitrary geometry primitives. "Any-hit shaders", "closest-hit shaders" and "miss shaders" are executed for the respective situation. Furthermore the pipeline includes generation and traversal of an accelerated scene geometry representation, which can be defined by the driver of the GPU. Existing drivers use bounding volume hierarchies (BVLs) [Gunther et al. 2007] which not only allow for hierarchically bounding primitives but also guarantee bounded memory usage. DXR acceleration structures are divided into two levels: Bottom-level and top-level acceleration structures. Bottom-level structures contain geometry in the driver specified form, BVLs for instance. Top-level structures contain one or many low-level ones. While bottom-levels can compute ray intersections fast they are expensive to generate and with top-levels it is the other way around, so it is suggested to generate the bottom-levels with the least possible overlapping [Wyman and Marrs 2019a]. As with other geometry representations mentioned in this report like sparse voxel-octrees, it is possible to only regenerate the acceleration

structure for dynamic parts of the scene, saving performance. See figure 14 for a simplified view of the new DXR ray-tracing pipeline.

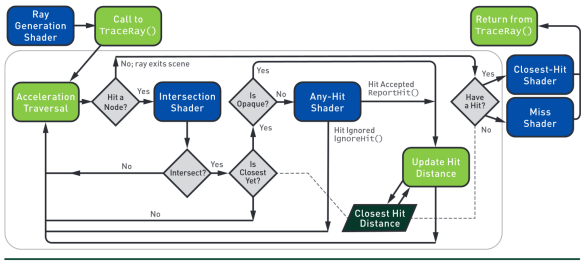

Figure 17: A simplified view of the new DirectX ray-tracing pipeline, including the five new shader stages (in blue): the ray generation, intersection, any-hit, closest-hit, and miss shaders. The complexity occurs in the traversal loop (the large gray outline, most of the figure), where rays are tested against bounding volume nodes and potential hits are identified and ordered to determine the closest hit. Reprinted from [Wyman and Marrs 2019b]

The RTX line of GPUs help with the generation of the acceleration structure and with the intersection testing against it. In addition the RTX cards support fast image denoising, leveraging AI technology presented in 2017 [NVIDIA a]. The need for denoising arises from the fact, that the number of rays used for real-time ray-tracing is very limited even with the new cards and DXR API. Applying reconstructive denoising filters to images rendered with a limited number of rays, a complete image noise-free can be calculated with much less rays than without. [Liu et al. 2019] successfully integrated DXR into the modern game engine "Unreal 4" and achieved real-time performance by leveraging the hardware-supported DXR ray-tracing pipeline combined with hardware-accelerated denoising.

With ray-tracing most accurate reflection effects like multi-bounce diffuse, glossy and specular reflection, caustics, occlusion and color bleeding can be achieved. Performance is still an issue with ray-tracing techniques and the number of rays per pixel than can be traced in real-time remains very limited. Modern algorithms combine ray-tracing with traditional rendering techniques to achieve best performance for now.

8 Conclusion

This report has summarized an overview of different techniques for computing reflections in real-time, discussing their different advantages and disadvantages. In conclusion it can be said that all methods have different strengths and weaknesses, so more often than not, it is advisable to use a combination of algorithms instead of just one, to try and harness their strengths and eliminate their weaknesses.

Acknowledgements

I thank Konstantin Lackner and Theresa Bodner, who both greatly helped me with the exposition of this report.

References

BLINN, J. F., AND NEWELL, M. E. 1976. Texture and reflection in computer generated images. *Communications of the ACM 19*, 10, 542–547.

BRENNAN, C., 2002. Accurate environment mapped reflections and refractions by adjusting for object distance. `http://developer.amd.com/wordpress/media/2012/10/ShaderX_CubeEnvironmentMapCorrection.pdf`. [Online; accessed 16-May-2019].

CAULFIELD, B., 2018. Whats the difference between ray tracing and rasterization? `https://blogs.nvidia.com/blog/2018/03/19/whats-difference-between-ray-tracing-rasterization/`. [Online; accessed 28-March-2019].

COOK, R. L., PORTER, T., AND CARPENTER, L. 1984. Distributed ray tracing. *SIGGRAPH Comput. Graph. 18*, 3 (Jan.), 137–145.

CRANE, K., LLAMAS, I., AND TARIQ, S. 2007. *Gpu Gems 3: Programming Techniques for High-performance Graphics and General-purpose Computation*, first ed. ch. 30, 633–675.

CRASSIN, C., NEYRET, F., SAINZ, M., GREEN, S., AND EISEMANN, E. 2011. Interactive indirect illumination using voxel cone tracing. *Computer Graphics Forum 30*, 7, 1921–1930.

DACHSBACHER, C., AND STAMMINGER, M. 2005. Reflective shadow maps. In *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games*, ACM, New York, NY, USA, I3D '05, 203–231.

DACHSBACHER, C., AND STAMMINGER, M. 2006. Splatting indirect illumination. In *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*, ACM, New York, NY, USA, I3D '06, 93–100.

DIEFENBACH, P. J. 1996. Pipeline rendering: interaction and realism through hardware-based multi-pass rendering.

EISEMANN, E., AND DÉCORET, X. 2006. Fast scene voxelization and applications. In *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*, ACM, New York, NY, USA, I3D '06, 71–78.

EISEMANN, E., AND DÉCORET, X. 2008. Single-pass gpu solid voxelization for real-time applications. In *Proceedings of Graphics Interface 2008*, Canadian Information Processing Society, Toronto, Ont., Canada, Canada, GI '08, 73–80.

FANG, S., AND CHEN, H. 2000. *Hardware Accelerated Voxelisation*. Springer London, London, 301–315.

GORAL, C. M., TORRANCE, K. E., GREENBERG, D. P., AND BATTAILE, B. 1984. Modeling the interaction of light between diffuse surfaces. *SIGGRAPH Comput. Graph. 18*, 3 (Jan.), 213–222.

GREENE, N. 1986. Environment mapping and other applications of world projections. *IEEE Computer Graphics and Applications 6*, 11, 21–29.

GUNTHER, J., POPOV, S., SEIDEL, H., AND SLUSALLEK, P. 2007. Realtime ray tracing on gpu with bvh-based packet traversal. In *2007 IEEE Symposium on Interactive Ray Tracing*, 113–118.

HAINES, E., AND AKENINE-MÖLLER, T., Eds. 2019. *Ray Tracing Gems*. Apress. [Online; accessed 23-May-2019].

HAINES, E., AND SHIRLEY, P. 2019. *Ray Tracing Terminology*. Apress, Berkeley, CA, 7–14.

HEIDRICH, W., AND SEIDEL, H.-P. 1998. View-independent environment maps. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, ACM, New York, NY, USA, HWWS '98, 39–ff.

HIRVONEN, A., SEPPÄLÄ, A., AIZENSHTEIN, M., AND SMAL, N. 2019. *Accurate Real-Time Specular Reflections with Radiance Caching*. Apress, Berkeley, CA, 571–607.

KAJIYA, J. T. 1986. The rendering equation. *SIGGRAPH Comput. Graph. 20*, 4 (Aug.), 143–150.

KELLER, A. 1997. Instant radiosity. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, SIGGRAPH '97, 49–56.

KILGARD, M. J. 1999. Improving shadows and reflections via the stencil buffer. *Advanced OpenGL Game Development*, 204–253.

KILGARD, M. 1999. Perfect reflections and specular lighting effects with cube environment mapping. *Technical Brief, nVidia Corp*.

LAGARDE, S., 2012. Image-based lighting approaches and parallax-corrected cubemap. `https://bit.ly/2M6VL7d`. [Online; accessed 28-March-2019].

LAINE, S., AND KARRAS, T. 2010. Efficient sparse voxel octrees. In *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ACM, New York, NY, USA, I3D '10, 55–63.

LAINE, S., SARANSAARI, H., KONTKANEN, J., LEHTINEN, J., AND AILA, T. 2007. Incremental instant radiosity for real-time indirect illumination. In *Proceedings of Eurographics Symposium on Rendering 2007*, Eurographics Association, 277–286.

LI, W., FAN, Z., WEI, X., AND KAUFMAN, A. 2007. *Gpu Gems 3: Programming Techniques for High-performance Graphics and General-purpose Computation*, first ed. ch. 47, 747–763.

LIU, E., LLAMAS, I., CAÑADA, J., AND KELLY, P. 2019. *Cinematic Rendering in UE4 with Real-Time Ray Tracing and Denoising*. Apress, Berkeley, CA, 289–319.

MARA, M., MCGUIRE, M., NOWROUZEZAHRAI, D., AND LUEBKE, D. 2014. Fast global illumination approximations on deep g-buffers. *NVIDIA Corporation 2*, 4.

MCGUIRE, M., AND MARA, M. 2014. Efficient GPU screen-space ray tracing. *Journal of Computer Graphics Techniques (JCGT) 3*, 4 (December), 73–85.

MILLER, G. S., AND HOFFMAN, C. 1984. Illumination and reflection maps. *Course Notes for Advanced Computer Graphics Animation, SIGGRAPH 84*.

NICHOLS, G., SHOPF, J., AND WYMAN, C. 2009. Hierarchical image-space radiosity for interactive global illumination. In *Proceedings of the Twentieth Eurographics Conference on Rendering*, Eurographics Association, Goslar Germany, Germany, EGSR'09, 1141–1149.

NVDIA, C., 2018. Nvidia rtx platform. `https://developer.nvidia.com/rtx`. [Online; accessed 22-May-2019].

NVIDIA. Nvidia optix ai-accelerated denoiser. `https://developer.nvidia.com/optix-denoiser`. [Online; accessed 23-May-2019].

NVIDIA, C. Vxgi technologie. `https://www.nvidia.de/object/vxgi-technology-de.html`. [Online; accessed 28-March-2019].

NVIDIA, 1999. Opengl cube map texturing. `https://www.nvidia.com/object/cube_map_ogl_tutorial.html`. [Online; accessed 28-March-2019].

PARKER, S. G., FRIEDRICH, H., LUEBKE, D., MORLEY, K., BIGLER, J., HOBEROCK, J., MCALLISTER, D., ROBISON, A., DIETRICH, A., HUMPHREYS, G., MCGUIRE, M., AND STICH, M. 2013. Gpu ray tracing. *Commun. ACM 56*, 5 (May), 93–101.

PHARR, M., AND FERNANDO, R. 2005. *Gpu Gems 2: Programming Techniques for High-performance Graphics and General-purpose Computation*, first ed. Addison-Wesley Professional, ch. 9, 143–165.

SCHWARZ, M., AND SEIDEL, H.-P. 2010. Fast parallel surface and solid voxelization on GPUs. *ACM Transactions on Graphics 29*, 6 (Proceedings of SIGGRAPH Asia 2010) (Dec.), 179:1–179:9.

SEGOVIA, B., IEHL, J. C., MITANCHEY, R., AND PÉROCHE, B. 2006. Bidirectional instant radiosity. In *Proceedings of the 17th Eurographics Conference on Rendering Techniques*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, EGSR '06, 389–397.

SLOAN, P.-P., KAUTZ, J., AND SNYDER, J. 2002. Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. *ACM Trans. Graph. 21*, 3 (July), 527–536.

SLOMP, M. P. B., OLIVEIRA NETO, M. M. D., AND PATRÍCIO, D. I. 2006. A gentle introduction to precomputed radiance transfer. *Revista de informática teórica e aplicada. Porto Alegre. Vol. 13, n. 2 (2006), p. 131-160*.

SOLER, C., HOEL, O., ROCHET, F., AND HOLZSCHUCH, N. 2010. A fast deferred shading pipeline for real time approximate indirect illumination.

SOUSA, T., SCHULZ, N., AND KASYAN, N., 2011. Secrets of cryengine3 graphics technology. `http://www.klayge.org/material/4_1/SSR/S2011_SecretsCryENGINE3Tech_0.pdf`. [Online; accessed 23-May-2019].

STICH, M., 2018. Introduction to nvidia rtx and directx ray tracing. `https://devblogs.nvidia.com/introduction-nvidia-rtx-directx-ray-tracing/`. [Online; accessed 28-March-2019].

SBASTIEN LAGARDE, A. Z., 2012. Siggraph 2012 and game connection 2012 talk : Local image-based lighting with parallax-corrected cubemap. `https://seblagarde.wordpress.com/2012/11/28/siggraph-2012-talk/`. [Online; accessed 28-March-2019].

THIEDEMANN, S., HENRICH, N., GROSCH, T., AND MÜLLER, S. 2011. Voxel-based global illumination. In *Symposium on Interactive 3D Graphics and Games*, ACM, New York, NY, USA, I3D '11, 103–110.

TOFT, A., 2017. Fast, accurate reflections withparallax-corrected cubemaps. `https://www.addymotion.com/files/dissertation-redacted.pdf`. [Online; accessed 16-May-2019].

WALTER, B., FERNANDEZ, S., ARBREE, A., BALA, K., DONIKIAN, M., AND GREENBERG, D. P. 2005. Lightcuts: A scalable approach to illumination. *ACM Trans. Graph. 24*, 3 (July), 1098–1107.

WIKIMEDIA, C., 2014. File:brdf diagram.png — wikimedia commons, the free media repository. https://commons.wikimedia.org/w/index.php?title=File:BRDF_Diagram.png&oldid=123484143. [Online; accessed 20-May-2019].

WIKIMEDIA, C., 2015. File:voxelgitter.png — wikimedia commons, the free media repository. https://commons.wikimedia.org/w/index.php?title=File:Voxelgitter.png&oldid=159805689. [Online; accessed 22-May-2019].

WIKIMEDIA, C., 2017. File:box - path tracing high.png — wikimedia commons, the free media repository. https://commons.wikimedia.org/w/index.php?title=File:Box_-_Path_Tracing_High.png&oldid=263559945. [Online; accessed 24-May-2019].

WRONSKI, B., 2014. The future of screenspace reflections. https://bartwronski.com/2014/01/25/the-future-of-screenspace-reflections/. [Online; accessed 28-March-2019].

WYMAN, C., AND MARRS, A. 2019. *Introduction to DirectX Raytracing*. Apress, Berkeley, CA, 21–47.

WYMAN, C., AND MARRS, A. 2019. *Introduction to DirectX Raytracing*. Apress, Berkeley, CA, 21–47.