CHALMERS
UNIVERSITY OF TECHNOLOGY

Generating Real Time Reflections by Ray Tracing Approximate Geometry

Master's thesis in Interaction Design and Technologies

JOHAN FREDRIKSSON, ADAM SCOTT

Department of Interaction Design and Technologies
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2016

Master's thesis 2016:123

Generating Real Time Reflections by Ray Tracing Approximate Geometry

JOHAN FREDRIKSSON, ADAM SCOTT

Department of Computer Science
Division of Interaction Design and Technologies
Chalmers University of Technology
Gothenburg, Sweden 2016

Generating Real Time Reflections by Ray Tracing Approximate Geometry
JOHAN FREDRIKSSON, ADAM SCOTT

Supervisor: Erik Sintorn, Computer Graphics
Examiner: Staffan Björk, Interaction Design and Technologies

Master's Thesis 2016:123
Department of Computer Science
Division of Interaction Design and Technologies
Chalmers University of Technology
SE-412 96 Gothenburg

Cover: BVH encapsulating approximate geometry created in the Frostbite engine.

Typeset in LaTeX
Gothenburg, Sweden 2016

Generating Real Time Reflections by Ray Tracing Approximated Geometry
Johan Fredriksson, Adam Scott Department of Interaction Design and Technologies
Chalmers University of Technology

Abstract

Rendering reflections in real time is important for realistic modern games. A common way to provide reflections is by using environment mapping. This method is not an accurate way to calculate reflections and it consumes a lot of memory space. We propose a method of ray tracing approximate geometry in real time. By using approximate geometry it can be kept simple and the size of the bounding volume hierarchy will have a lower memory impact. The ray tracing is performed on the GPU and the bounding volumes are pre built on the CPU using surface area heuristics. The triangle intersection data is pre calculated in order to keep the run time costs low. The method is implemented and tested on the Frostbite game engine.

Keywords: ray tracing, reflections, real time, acceleration structures

Acknowledgements

Big thanks to: Erik Sintorn, our supervisor for all help, feedback, and suggestions. Everybody at Ghost for everything, especially Andreas Brinck, Adam Carlson, Matthew Clarke, Lars Pensjö, David Skyfall and Martin Green for helping us with all our integration issues.
The Image Quality team at Frostbite, in particular Charles de Rousiers and Tomasz Stachowiak for the detailed explanation and help they offered us.
Ghost Games for allowing us to use their assets and resources and providing us with a great working environment.
We would also like to thank our friends and family for supporting and cheering us on during the Chalmers years. - Adam Scott and Johan Fredriksson

Contents

1
Introduction

In the video game industry the graphics are becoming more and more realistic. As the computing power of GPUs are growing, the possibilities of what can be rendered in real time grows with them. To provide realistic renderings an important aspect is the simulation of light. It is common to approximate or "cheat" in these simulations in order to achieve a viable calculation speed. An aspect of light simulation that is expensive to simulate is reflections. As light moves from a light source and bounces off materials, reflections are created. The color of the reflections vary depending on the materials it comes in contact with along the way.

1.1 Background

Reflections are everything that can be seen, except for direct light. It is a vital part of providing realistic effects in rendering. In order to create a realistic rendering, it is necessary that the entire setting feels real to the observer. Specular and glossy reflections are a big part of making a setting realistic, e.g. an observer expects to see reflections when looking into a mirror. Accurate reflections can be expensive to calculate and are normally only approximated. A widely used method to accurately simulate reflections is ray tracing, which is an expensive process. This thesis will focus on how to use ray tracing to calculate reflections in real time. Ray tracing is performed by tracing the path of the light through the individual pixels of the screen and simulating its interaction with virtual objects, as is shown in Figure 1.1. Ray tracing works well with dynamic lighting models as the light is not necessarily determined with pre computed light maps but can instead be calculated during run time. For accurate and life-like reflections ray tracing can become expensive to calculate, since more recursive rays will have to be used. This, in turn, demands incrementally more computing power to process. However, when utilizing ray tracing in real time, a higher frame rate can be achieved by exchanging reflection quality for performance.
The cost of ray tracing is dependent on the complexity of the environment, as the environment gets more detailed the rays needs to perform additional intersection checks. This creates a bottleneck in the rendering process when scenes becomes large and complex. The method described in this thesis is developed to suit the future needs for the game Need for Speed [2] using the Frostbite game engine [9]. The existing system for handling reflections uses screen space reflections (SSR) and image based lighting (IBL) through parallax-corrected cube maps which is further described by Lagarde[43]. This implementation faces three major obstacles for po-

tential future use cases. The first occurs when traveling at a high velocity in the game and the cube maps fails to update fast enough, which leads to the method failing to produce any reflections from the IBL system. The second, is that IBL fails to handle dynamic lighting in the environment. The third and final obstacle, is the discrepancy between the flat textured approximation for the scene around the cube map and the actual geometry. Cube maps are rendered from a single point in space, so for any other points within the cube the reflections are just approximations.

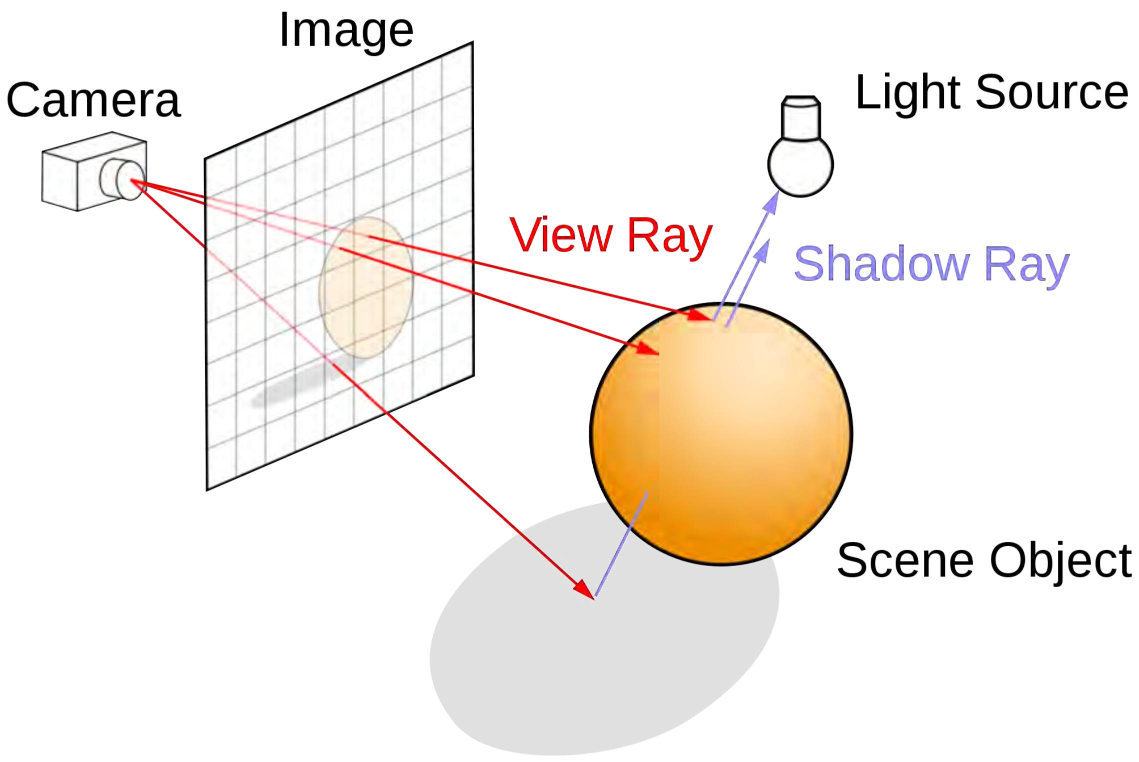


Figure 1.1: Ray tracing used to create shadows.

1.2 Purpose

The purpose of this thesis is to implement and evaluate a software solution for real time reflections in the Frostbite engine which is compatible with the existing SSR system. This thesis aims to replace the IBL implementation with a GPU ray tracer, tracing against low fidelity proxy geometry. As with IBL this is used in order to approximate the reflected color in different types of reflective material in the virtual environment. The thesis will also cover researching supporting data structures that adheres to the given constraints of the project. Any potential integration issue during development will have to be handled, since the solution is intended to be compatible with the currently implemented system. Therefore, development will proceed according to specifications set by the stakeholders Ghost Games [1] and Frostbite. The stakeholders have two important concerns that the method should be able to handle. The first is to load reflection meshes from memory at a sufficient rate, even when moving through the environment at high speeds, as this is problematic with the current implementation. The second is that the solution should be able to handle dynamic lighting for possible future needs. The second concern makes it difficult to utilize pre computed illumination as the current solution does.Ray tracing against a complex environment can imply a performance loss. Furthermore, the amount of memory occupied by the acceleration structure increases with the amount of triangles in the geometry. Therefore, ray tracing will be performed against proxy geometry, a simplified mesh representing the environment. Ray tracing against a proxy geometry also marginally reduces the amount of intersection checks since the ray tracing is performed against less triangles. This will provide decreased calculations in real time at the cost of the quality of the reflections. With this method the aim is to provide reflections that are accurate enough that the end user will not notice the difference, while decreasing the required memory. This method is developed with the racing game Need for Speed in focus, so the scenes will be both large and complex. This leads us to believe that the previously mentioned bottleneck in memory can possibly be decreased by utilizing our method, since the bounding volume will allocate less memory compared to the size of the cube maps. Another advantage the proposed method provides is an improved work flow for artists. With the current solution involving cube maps the artist manually has to place cube maps in the game world. By ray tracing proxy geometry this process can be automated to remove this extra workload from the artists.

1.3 Problem Statement

The core problem is that the current utilized solution for resolving real time reflections with SSR and IBL is not fulfilling the two desired specifications given by the stakeholders. When using IBL, updating the cube maps in sufficient time becomes an issue. The cube maps for IBL are created iteratively as the camera moves through the scene, and at higher speeds the cube map creation can not update the cube maps fast enough. An IBL based solution also has trouble handling dynamic lighting, since it utilizes pre generated light maps. The proposed method of ray tracing proxy geometry coupled with the implemented SSR solution, as an alternative to IBL, can provide solutions to both of the mentioned specifications. However, the computational requirements for constructing a real time reflection solution, in the form of a GPU ray tracer, will entail several bottlenecks. The bottlenecks will, with consideration to the overarching limitations set on the project, be handled on a priority basis. The implementation of the ray tracer will be developed to structurally and architecturally fit the current implementation of Need for Speed. Therefore, part of this project will focus on gaining knowledge and understanding of the rendering software in the Frostbite engine, to efficiently be able work with the existing system. Need for Speed is developed for the current generation of consoles, Xbox One, and Playstation 4 (PS4). Therefore, the hardware [10] [5] on one, or both, of these consoles will act as a benchmark of the available computing power the proxy geometry implementation can utilize.
Existing ray tracers will be evaluated based on the efficiency of the tracing speed. In order to achieve an acceptable frame rate, improvements of the tracing algorithms and acceleration structures for the geometry in the environment will be researched.

The quality of the reflections provided by tracing proxy geometry will then be validated in relation to the quality provided by the IBL method. An issue with ray tracing is how different materials and filtering is to be handled. For example, simulated glossy reflections [42] require additional rays to be cast in order to accurately display a reflection. Computing similar cases by casting multiple rays for each pixel per frame can be detrimental to the execution speed of the algorithm. The goal of this project is to develop an efficient ray tracing solution compatible with the current generation of consoles. This means that the ray tracing will have to be performed in real time and adhere to the memory restrictions developing for consoles implies. Questions that will be answered in this thesis consist of the following; how can ray traced proxy geometry reflections seamlessly be integrated with the current SSR implementation? How can ray tracing algorithms and acceleration structures be used to keep the rendering overhead low while still producing acceptable reflections? And finally, do the reflections produced by the proxy geometry sufficiently replace the IBL implementation in consideration to image quality, while maintaining the same or better performance?

1.4 Limitations

The subject of this thesis will be affected by memory concerns, therefore, the entire proxy geometry cannot be in memory at once. As a solution it will have to be streamed on demand. However, even as it is important that the proxy geometry is optimized to be streamed in an efficient manner, this subject will not be a focus for this thesis since it is outside of the intended scope. The reflections will be limited to compute no illumination, and only use pre computed lighting in textures, in order to keep the amount of computations low. Therefore, ray tracing against dynamic light sources will not be performed in this thesis. However, the implementation could be used as a basis for future research and development in order to enable ray traced reflections which incorporates dynamic light sources. The current SSR solution will not be improved in this thesis. Instead it will solely focus on the the proposed method as an extension to the currently used SSR implementation.

2 Theory

This chapter presents background information and previous work of methods related to this thesis. The purpose of this chapter is to analyze ray tracing and its performance. In order to explore how ray tracing performance can be improved, research on the most common acceleration structures is presented. Mainly, since performance is in focus, the acceleration structures will be compared based on their potential run time speed. To further increase ray tracing performance, heuristics for building acceleration structures will be analyzed on how they can improve ray tracing for this study.

2.1 Ray tracing

Ray tracing has several use cases, for example, collision detection [21]. However, relevant to this thesis is that in computer graphics ray tracing can be used to accurately generate images. The generation performed by tracing the path from the eye through screen space pixels and simulating the behavior of light traversing through virtual objects in the scene. The technique is capable of creating highly realistic reflections compared to other more widely used approximations [14] [43]. Furthermore, ray tracing can be used to produce common visual effects, such as shadows and refractions [12].
Ray tracing in computer graphics was introduced as early as in 1968 by Arthur Appel [13]. Since ray tracing is able to create realistic lighting simulations it remains popular and widely used. However, because it is a very costly process it is not utilized in most real time games. A known exception is the game Enemy Territory: Quake Wars [37] which is a remake of the original game, using ray tracing instead of rasterization. The final result struggled with performance issues when compared to the original game. Nevertheless, even though the game suffered from lower frame rates, it shows that it is possible to use ray tracing in real time applications.
There are two main ways to perform intersection checks when ray tracing, ray marching and ray casting [34]. Ray marching is able to provide results when encountering isosurfaces [16] or representations of volumetric objects [49]. The technique is also used for resolving materials such as marble or skin where the ray scatters inside the material itself. Ray marching is performed by stepping along the direction of the ray and gathering data on how the ray is affected. It is also used to find the first intersection point in some cases, e.g. with SSR. When compared to simple ray casting which performs an intersection check with the ray against potential surfaces, it is a more expensive process.

2.1.1 Acceleration structures

Representing a 3D environment in an efficient data structure is one of the key components of making ray tracing possible with respect to frame rate. In order to make the ray tracer efficient, different acceleration structures are analyzed in order to find which one best suits the specifications for this thesis. Bounding volume hierarchies (BVH) or K-D trees generally achieves the best performance in rendering [40]. The following sections explores some of the common ray tracing acceleration structures and how they diverge.

2.1.1.1 Binary space partitioning trees

A binary space partitioning tree (BSP) is created by recursively splitting hyperplanes along the orientation of the containing primitives. The BSP partitions becomes increasingly smaller further down the tree hierarchy. As the spatial representations becomes smaller, the resolution of given space becomes higher [33]. This results in a binary tree where spatial traversal with ray tracing can be performed.

2.1.1.2 K-D trees

K-D trees are a subset of BSPs, more specifically they are binary trees where nodes are splitting planes and leafs are holders of geometrical primitives such as triangles. The difference between K-D trees and BSPs are that in a K-D tree all dividing lines needs to be parallel to the axes. The nodes are generated by splitting the parent of the node dependent on a pre-defined split heuristic. For a canonical balanced tree the median can be chosen as split heuristic, thus, guaranteeing a $O(\log n)$ search cost. A visualization of a K-D tree is shown in Figure 2.1. K-D trees have a lower number of intersection tests compared to BVHs. Although, the ray tracing performance is still normally better for a BVH [44]. However, Vinkler et al. found that K-D trees tend to outperform BVHs in very large scenes with high occlusion.

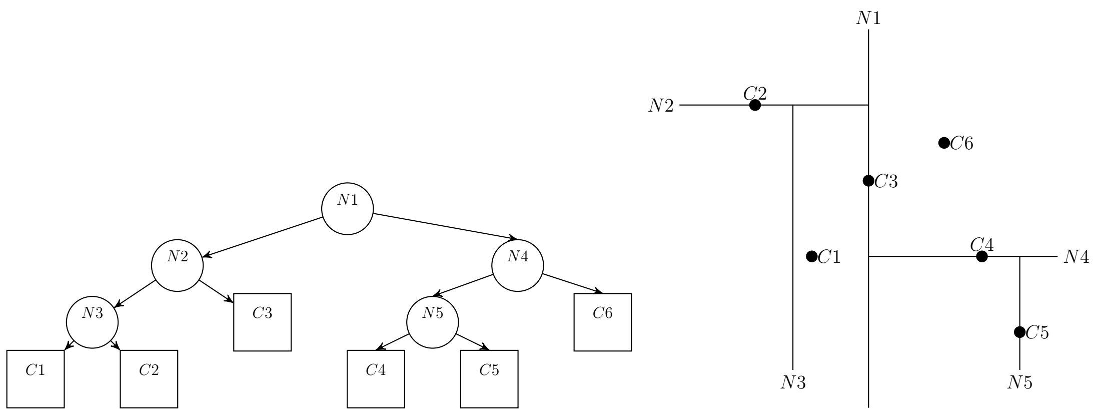


Figure 2.1: The resulting tree the K-D algorithm creates is a space subdivided binary tree with a low/hi split. In the tree on the left side nodes are represented by circles and leaves by squares. In the tree to the right the spatial subdivision of the same nodes are presented.

2.1.1.3 Bounding volume hierarchies

A BVH is a binary tree consisting of recursively built bounding volumes that holds geometry in the same proximity. Constructing a BVH starts with a single root node that encapsulates the entire geometry of the processed working set. This node is then recursively split into two (or more) children, each child a subset of the root node, the splitting continues until the child nodes holds a predefined amount of primitives or less. Figure 2.2 shows child nodes as the inner bounding volumes with the most opaque tint of gray. The construction recursion stops at a predetermined termination expression, such as maximum amount of primitives in the volume or a heuristic. Thus, resulting in workable subsets of the whole geometry.
Utilizing a BVH presents several advantages, such as a simple construction process and a low memory footprint [40]. A low memory footprint is desirable, since, for the purposes of this thesis, memory impact is in focus.
M, Chajdas [17] states that BVH structures on an average has a better traversal performance, coherency, and execution time for ray tracing compared to K-D trees and octrees. Luebke et al. also concludes that BVHs perform better than K-D trees in real time [30]. The mentioned report focuses on a voxel based visualization pipeline, however, the specifics of the data structures and the tracing results remain relevant nonetheless. The results of the report affects the theoretical usage of BSP trees as well, since a K-D tree is a subset of BSP trees.

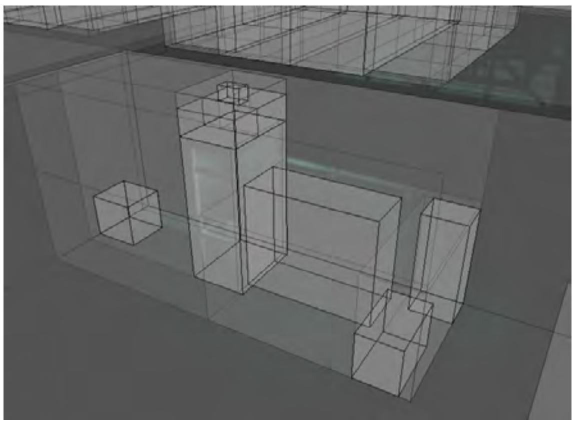


Figure 2.2: Bounding volumes visualized in gray, constructed over several primitives.

A BVH requires a certain spatial dependency, where primitives sharing a leaf node are in the same proximity of each other. Since this dependency hierarchically goes upwards the binary tree, moving a geometrical primitive between subsets can result in costly operations. This leads to BVHs not being well suited for run time construction, especially with heavy split heuristic operations for optimizing tracing performance. Lauterbach et al. [29] proposes two novel approaches to be able to generate BVH trees at run time. The first one, a linear bounding volume hierarchy (LBVH), represents the triangles in the scene with Morton codes which allows a BVH to be reduced to a linear sorting problem. This method splits the BVH at a spatial median distribution, which is, while quick to build, not as optimized as a SAH split for ray tracing. The other method is an approximated GPU SAH approach resulting in better trace performance but slower build times. The paper also proposes a hybrid implementation of the algorithms, letting the LBVH split at shallower depths of the binary tree. Due to the possibility of parallelizing the execution, it allows GPU SAH to take over at finer details of the scene. The hybrid solution shows some promising use since it both offers the construction speed of the LBVH implementation and the tracing optimization benefited from the spatial area heuristic [29]. However, a CPU built, pre-processed, full SAH still offers the highest run time speed and is preferable in scenarios where pre-building is possible.

2.1.1.4 Grids

A grid is created by performing uniform spatial subdivision of an area [23]. The construction speed of a grid is fast and the complexity of the construction algorithm can be made linear [24]. The creation time of the grids can be made quick enough for real time reconstruction, thus, enabling ray tracing dynamic geometry and animated scenes [47]. The underlying reason why the reconstruction speed of grids is fast, is that primitives can quickly be allocated from one grid to another. When reallocating a primitive in e.g. a BVH more than one node can be affected and several volumes may have to be re-sized in order to be optimal. However, using a uniform grid is not suited for ray tracing large scenes and generally a BVH outperforms a grid solution in run time[47]. Grids are most useful in environments with dense primitives [15]. In situations where the primitives are small, a lot of intersection checks will be performed on empty cells. This can be solved by making the cells cover a larger area, which in turn leads to the issue of a lot of primitives inside one cell. This problem is also known as the "teapot in the stadium", which makes it difficult to select an effective resolution for a non adaptive grid. A further improvement is to let grids recursively increase resolution of the cells depending on the amount of triangles each cell contains. This increases the cost of the construction process and also memory consumption, but results in a more efficient traversal time.

2.1.1.5 Octrees

Octree creation starts with a single root node which represents all geometry in the scene. By recursively splitting each node along the axes into eight child nodes, a traversable hierarchy is created. This process results in higher resolution of bounding volumes in complex areas, where the primitives are placed in the child that encloses it [31]. A higher resolution leads to significant increases in memory consumption, especially if the octree is not adaptive to the amount of primitives in an area.
A promising research area in relation to ray tracing and octrees are voxel hierarchies. Specifically sparse voxel octrees such as the implementation proposed by Laine et al. [28]. The technique consists of removing the separation between mesh, normal data, displacement, and color data. This is performed by storing them in the geometrical representation of the analyzed intersection by voxelizing the scene. Each voxel contains the detail of the point in question. The voxels are then stored in an octree, the sparse attribute of the tree is that only sub trees containing geometry is included in the ray traced hierarchy. The sparse voxel octree contains several mipmaps of the

encapsulated environment, thus, letting the data rich voxels optimally limit look ups. This implicitly solves different levels of details of the intersected geometry depending on tracing distance.
The technique exceeds rendering performance compared to a highly optimized traditional triangle tracer [11]. This is mainly because of its efficiency when shading and post-processing.

2.1.2 Spatial splitting

How to properly perform spatial splitting is a widely discussed subject in computer graphics. However, one thing that is agreed on is that good spatial splitting is required for bounding volumes in order to perform real time ray tracing in complex environments. This section will discuss both the heuristics and binning methods that have been analyzed for the ray tracer.

2.1.2.1 Heuristics

Early attempts at creating efficient spatial subdivisions for ray tracing and similar use cases involved picking an arbitrary split candidate for subdividing the node at hand. This split candidate is known as the spatial median, and represents the exact middle point of an axis belonging to the currently analyzed bounding volume. By inherently adhering to the same formula ($\frac{1}{2}axis$). Spatial median split enables a memory efficient split operation, since each node implicitly defines the splitting volumes of its children. Since this method does not take into consideration where primitives are positioned in the resulting bounding volumes, the heuristic leads to memory inefficient fragmentation [31].
Glassners [19] early research involved analyzing the performance of two octree implementations, one with respect to surface volume of the bounding box or object density and one not. At this point the threshold chosen for a split seemed like an important delimiter for the efficiency of said algorithm. Kaplan [25] continues to work in a similar direction to the implementation presented by Glassner, but in the form of a binary tree. In this method one node could hold the same amount of primitives that is divided in up to four nodes with the implementation provided by Glassner. Split heuristics was originally motivated by the fact that ray traversal is the delimiting bottleneck in the operation, not the construction. Therefore, enabling the build procedure to aid the traversal method to perform better, results in an overall better performance [31].
Stone [41] states that rays has a deterministic probability of hitting a convex polygon depending on the surface area of the object, the distribution of the rays, and the distance between the polygon and the ray origin. Since a bounding volume is a primitive representation of a convex object, this statement can be applied for creating a measurable heuristic depending on the surface area of the object. When describing said heuristic, several assumptions [46] are made: The rays are handled as uniformly distributed infinite lines, traversal cost (K_T) and triangle intersection (K_I) are handled as known values and finally, intersecting N triangles are linearly bound to the intersection cost, i.e. ($N * K_I$).

$$C_V(p) = K_T + P_{[V_l|V]}C(V_l) + P_{[V_r|V]}C(V_r) \tag{2.1}$$

Expressing cost ($C_V(p)$) for a plane is the traversal combined with the cost of intersecting the children of the node, this relation can be observed in equation 2.1. $P_{[V_x|V]}$ in the equation is the probability of hitting the child AABB x and $C(V_x)$ is the cost of the sub-tree containing child x.

$$C(T) = \sum_{n \in nodes} \frac{SA(V_n)}{SA(V_S)} K_T + \sum_{l \in leaves} \frac{SA(V_l)}{SA(V_S)} K_I \tag{2.2}$$

In Equation 2.2, $C(T)$ is representing the cost of the entire tree. This is the measuring cost that needs to be as low as possible to ensure that the tracing algorithm maintains a low amount of intersection checks and therefore, more efficient uses of the allocated resources. Because of the recursive nature of the subdividing step, a measurable termination step related to the split cost in comparison to the cost of intersection the primitives contained by the bounding volume is necessary. Thus, enabling the algorithm to only process relevant bounding volumes and not recurse for an eternity.
Stich et al. [40] proposes a combination of both surface area heuristic (SAH) search and traditional spatial median split, thus, resulting in the optimal split being chosen depending on the SAH cost related to each split heuristic. Depending on the needed resolution of the ray tracing pass and the allocated pre-processing resources, techniques similar to this could be beneficial for a cheap, yet efficient, data structure.

2.1.2.2 Binning

Binning is to equidistantly split an arbitrary working set, which in the case of ray tracers are bounding volumes, in pre defined bin sizes. Each bin acts as a pivot element creating subsets of the working set. This allows an algorithm to perform, or analyze a permutation of the set depending on the set binning step. How optimal the final split candidate is, is bounded to the resolution of the binning size.
At each split step, the heuristic is the delimiting factor for evaluating a potential split. How each split candidate is chosen is dependent on the information the split heuristic supplies. Since this information only can be available when compared to each other, binning usually is performed to be able to perform this comparison.
To perform binning is a costly operation but as MacDonald states [31], for static geometry, pre constructing a ray tracing data structure is cheaper than the actual tracing operation. Therefore, any optimization for the tracing step that derives from construction, increases the run time tracing performance.

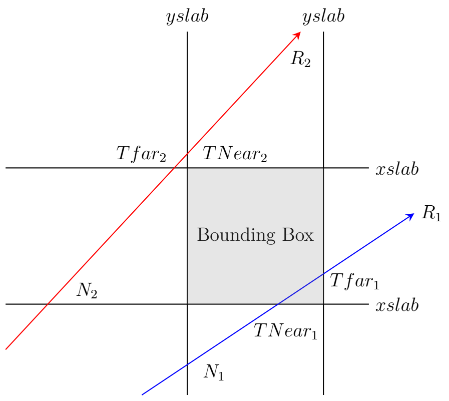


Figure 2.3: Ray-box intersection on an axis aligned bounding box. Ray R_1 is a hit therefore, $TNear_1 < Tfar_1$, whereas R_2 misses and $Tfar_2 < Tnear_2$. N_1 and N_2 is evaluated near intersections which are not chosen as $TNear_n$, because $TNear_1 > N_1$ for R_1 and $TNear_2 > N_2$ for R_2.

2.1.3 Box intersection

The speed of the ray-box intersection during ray tracing is vital when real time speed is desired, since this is normally the most time consuming task in a ray tracer [39]. Thanassis [7] utilizes a method for ray-box intersection which shows fast results. In this method the intersection can be computed as three individual slabs, each representing the X-,Y- and Z-planes. A 2-dimensional version of this is shown in Figure 2.3. The intersection test finds Tfar and Tnear for each pair of slabs with respect to the ray. Comparing the nearest out of the Tfar values with the farthest of the Tnear values you determine if it is a hit of the bounding box or not. The Figure 2.3 shows two near intersections (N_1, N_2) belonging to the rays R_1, R_2, both of them showcases near intersections not selected by the algorithm since the chosen intersections $(Tnear_1, Tnear_2)$ is further away from the ray starting position. The ray has similar intersections (F_1, F_2) not being presented in the figure. They are discarded by a similar logic, when the chosen Tfar for each ray is closer to the ray starting position than F_1 and F_2.

2.1.4 Triangle intersection

After determining if a ray has hit a leaf node or not, the triangles of the leaf has to be evaluated to determine if the ray hits a triangle at all and if so which triangle is closest to the origin of the ray. A common method [36] for triangle intersection consists of a two-pass routine. Firstly, the analyzed ray is checked if it intersects the plane belonging to the triangle. Secondly, the intersection is checked if it is inside the triangle or not.

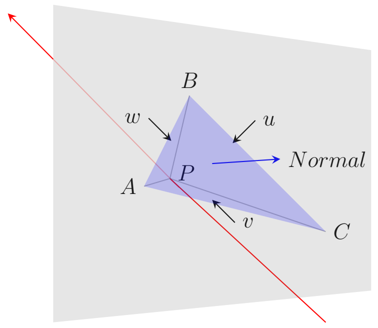


Figure 2.4: Ray-triangle intersection. The picture shows a triangle with vertices A, B and C. The triangle plane and the triangle ABC shares a normal since they are aligned. The red ray intersects the triangle at intersection point P. The areas BAP, CBP and ACP are expressed as w, u, and v respectively.

The plane intersection pass consists of calculating the normal of the plane, as can be seen in Figure 2.4. The normal is used for finding the distance between the origin of the ray and the intersection. This distance is then inserted in the plane intersection equation $P = RayOrigin + distance * RayDirection$, where P is the intersection point. The plane equation, $Ax + By + Cz + D = 0$, can be derived to express D in the form of $D = dot(Normal, V)$, where vertex $V \in [A, B, C]$. By combining the plane equation and the plane intersection equation, distance can be expressed as $distance = -(dot(Normal, RayOrigin) + D) \,/\, dot(Normal, RayDirection)$. Before calculating the distance equation, the equation $dot(Normal, RayDirection)$ has to differ from 0 since this would imply that the ray and the triangle normal is perpendicular to each other. Another special case is when the *distance* is calculated to be negative, this case means that the triangle is behind the origin of the ray, and should therefore, not be considered for further evaluation. For the next step, to determine if a hit is inside a triangle or not, Ericson [18] proposes a solution that originates in the fact that any point P sharing the same plane as the triangle can be expressed as $P = uA + vB + wC$, where $u + v + w = 1$. These coordinates are called areal coordinates or barycentric coordinates and are normalized representations of the contributed areal coverage of the triangle u, v and w (Figure 2.4). Each area u, v and w can be expressed in the form $w = \frac{TriangleBAP_{area}}{TriangleBAC_{area}}$. This equation can be derived to express the barycentric coordinate w as $w = (AB \times AP) * Normal$ [36]. By calculating each area u, v, and w and for each coordinate stop the evaluation for the triangle if u, v or w is < 0, since that the hit is therefore, outside of the triangle. Any triangles that have passed this step is guaranteed to be hit the by the incoming ray.

2.2 Existing method

The existing method utilizes a version of cube map reflections [20] in combination with screen spaced ray tracing (SSR). Cube map reflections are generally a faster technique than ray tracing, but is normally only accurate for a single point in space. Another issue is that seams between two different cube maps can be visible if two neighboring objects use different maps. To increase the accuracy of the cube maps the reflections are parallax corrected [27] before rendered to the screen, as presented in the game Remember Me [6]. The spatial points inside the cube maps are adjusted to get the correct parallax value so the displayed reflections are as correct as possible. However, when cube maps are used with fast moving cameras, as is the case in Need For Speed, the cube maps need to cover larger areas with lower resolution. This is since it puts a strain on the memory bandwidth, which leads to slower streaming times. The lower resolution in combination with larger cube maps leads to poorer quality in the reflections as well as errors for large parallax corrected values. Another problem is that the cube maps are generated in run time, by creating one side of the cube map per frame, and this method is easily overwhelmed when facing large amounts of data that needs to be updated. An issue with the current implementation is that the cube maps has to be manually inserted in the scenes. This in turn increases the workload of the artists. The integration between the SSR and the IBL in the existing solution shows some artifacts in the seams between the SSR and the IBL, as is shown in Figure 2.5. The main reason for this seam is the low resolution of the cube maps used by the IBL. The resolution of the cube maps are 128x128 for each side of the cube, and there are constantly 41 cube maps loaded in memory.

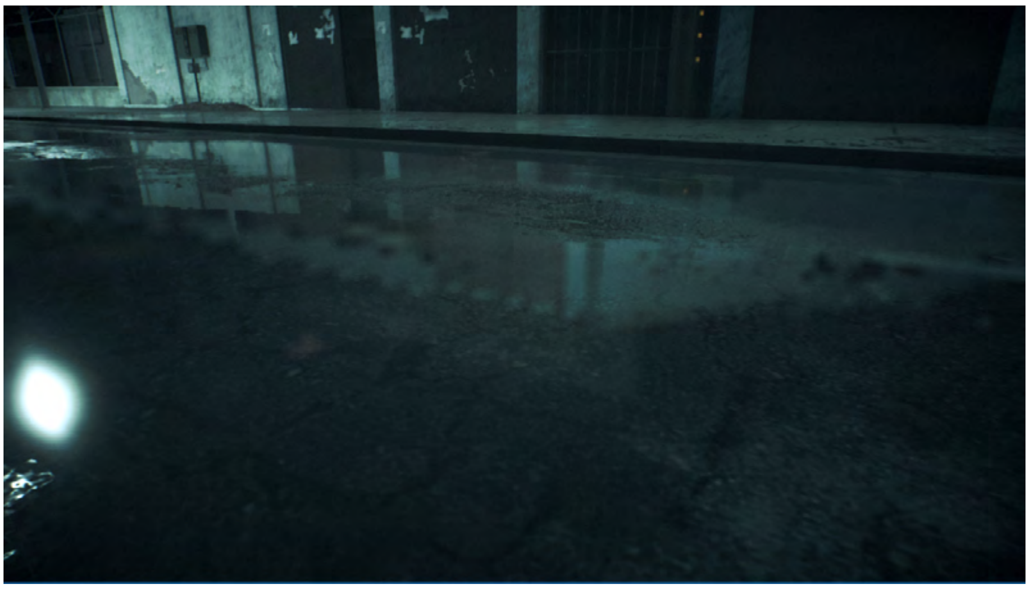


Figure 2.5: Lower resolution on cube maps are visible when compared with SSR.

2.2.1 Screen space reflections

SSR is performed by ray tracing the scene using the depth buffer, as opposed to the actual geometry. SSR normally only requires a ray and the scene, so the implementation can be made isolated if necessary. This technique works well with any reflecting surface, even curved surfaces such as waves on water. Screen space ray tracing can enable fully dynamic reflections, ambient occlusion, and refraction by only using the depth buffer [32]. However, the technique fails to provide accurate results in certain scenarios. The first, if reflections are of objects outside of the view port, as is shown in Figure 2.6, SSR will not be able to draw them. Secondly, back facing information is not accessible through this technique, as is shown in Figure

2.7. This makes it problematic in some cases, e.g. to produce mirror reflections of a character from a third-person view. The third inaccuracy is visible in cases such as the one shown in Figure 2.8. In this case, the reflective information is front facing but obscured by another object. SSR lacks the color information for these cases even if the intersection can be found.

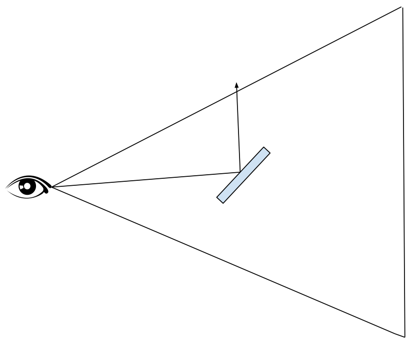


Figure 2.6: Ray reflected outside of the field of view.

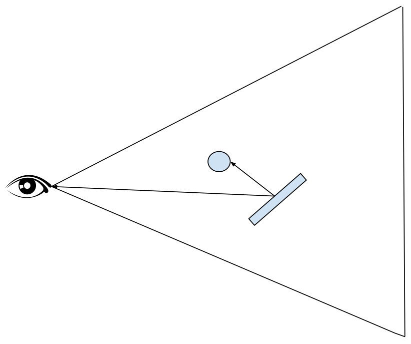


Figure 2.7: Ray intersecting with back face of an object.

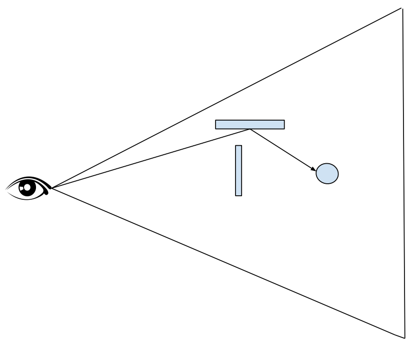


Figure 2.8: Ray intersecting with an obscured object.

2.2.2 Cube maps

Cube maps are a commonly used method for environment mapping which was first introduced by Greene [20]. The name cube mapping is derived from the fact that it is used to map the environment in the form of sides on a cube. A cube map is created by rendering the six sides of the scene from a single viewpoint. The mapping is stored as six textures, one for every side of the cube, or as six regions of a single texture. Therefore, the quality of the reflections are dependent on the quality of the textures. This also leads to increased memory costs in order to get better reflections. The biggest flaw with cube maps are how they are only correct for a single point in space. There are ways of improving this, as with the previously mentioned method of parallax correction. However, due to the fact that it is rendered from a single point in space, reflections that are not in this point will not provide completely accurate results.

2.3 Previous Work

Ray tracing has many applications in computer science. It is applied in different fields for intersection detection and can be applied for tracing the path of light in computer graphics. Thanassis [7] developed a tracing algorithm using BVHs in 2011. The ray tracing algorithm he created is a real time CUDA [3] ray tracer. Due to this the implementation is not compatible with Frostbite in itself, but the algorithm has been studied and used as a base for the implementation of this paper. For more information on fast tracing algorithms, Aila et al. [11] explains how to optimize ray tracing for GPUs. Wald et al. [45] presents a promising implementation of surface area heuristic splitting for improving the traversal speed of BVHs which our implementation relies on to decrease the real time computations.

3
Methods and Implementation

Implementing a new reflection technique in a system as large as Frostbite requires a lot of work with integration. This in turn makes it difficult to understand if bugs in the implementation stems from miscalculations or integration issues. This issue was solved by continuously testing all integrated elements in order to make sure that the output was calculated properly. During the integration with the pipeline software there are many stages were the bounding volume data can be negatively affected. Therefore, we continuously checked to make sure that the data was not corrupted along the way. Another challenge with this thesis is solving for problems that tracing against a geometry which differs from the rendered geometry can create. When tracing reflections against a simplified geometry the results can appear strange in certain situations. For example, in some cases, certain buildings might not be represented at all in the simplified geometry and will therefore, appear to have no reflection. In other cases, the simplified geometry can be larger than the original and it would appear to show a reflection of itself.

3.1 Overview of the implemention

The implementation takes a BVH, a texture containing the triangle colors, and the ray to be traced. If the SSR fails to find a hit, the same ray is used for proxy ray tracing instead. This ray is cast against the BVH and in case of an intersection, returns the color of the closest triangle hit and the hit distance. Currently, the textures does not contain any pre computed illumination, which would be very beneficial for the final results. After the tracing process, the color is filtered with the same filtering system as for the SSR.

3.2 Pre development

The initial phase of the development process consisted of focused testing of specific components of the system. This was done by setting up a testing environment in the form of a CUDA [3] ray tracer. This tracer is based on the implementation of Thanassis [7] and the path-tracer by Kutz and Li [4]. The SAH-based BVH tree is implemented with the SAH construction in I. Wald's implementation [45] as base and specific kernel routines defined in [8]. Early implementations were tested at this stage and evaluated depending on relevancy for the intended implementation.
Efficiency and use case driven functionality such as back-face culling [22] were tested. Back-face culling is necessary for removing artifacts that appears because of the

quality difference between the actual geometry and the approximated mesh. This problem is visualized in 2D in Figure 3.1.

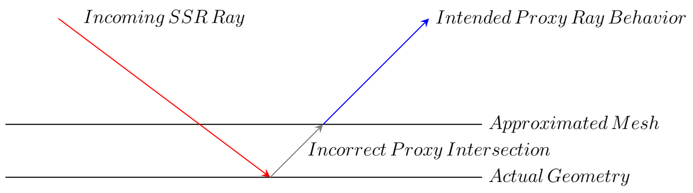


Figure 3.1: The red ray in the figure represents the incoming screen space reflection ray. The gray ray is the incorrectly intersected proxy ray, resulting in self reflection. The blue outgoing ray in combination with the gray ray is the intended behavior of the proxy ray.

3.3 Software development

The software development process was performed on a single computer, since the resources provided to this project was limited to one computer fulfilling the necessary criteria for implementing and testing the software. For this reason all major parts of the code was created using pair programming [48]. This caused our development process to be a bit slower than expected, but at the same time more thorough, so this process likely decreased the time spent debugging, as well as provided both members of the group full insight on all aspects of the implementation.
The development was performed in a test driven manner where all implementations where verified with quick sanity checks. Once the output of the implementations have been validated and the result was satisfying we would move on to the next step in the solution. Testing each part of the implementation directly was an important step in the process, since integration issues could otherwise make it difficult to find out where an error originates from. However, it was also important to keep the tests quick and simple so the process could move on as soon as possible.

3.3.1 Testing and validation

When implementing the new method, integration issues will be faced, especially when considering the overall design. These issues can draw focus from familiarizing oneself with the new method, when focusing on how to properly integrate the method with the game engine. To ease the transition from the idea of the method to its integrated version, test code prototyping the intended method is implemented as a stand alone version. This in order to validate the method on its own. If the results are satisfactory, it will then be implemented in the engine.
During the later course of the implementation phase of this thesis the results are tested through quantitative validation. The results are measured against the current implementation and the maximum frame budget allowed in the context. This provides solid measurement of how computationally effective the solution is.

3.3.2 Pipeline

The existing system available in Need for Speed utilizes a rigid pipeline to supply the large amount of data streams present and to pre calculate as many calculations as possible to maintain a playable real time game. To fulfill the goals of the project (section 1.2), geometrical primitives intended for use has to be pre-calculated at compile time and passed in appropriate form to the pipeline.
The first step toward this was to implement a policy class, `RaytraceProxyPolicy`, acting as an implicit connection between the low level data meshes present in the engine and a run-time class triggering on world traversal. The traversal functionality blends well with what this thesis project aims to achieve. This is because the content of the meshes adheres to a high level split hierarchy, thus, coupling relevant functionality to sub-portions of the game. By only processing and having the currently visited spatial proximity of the car held in memory at any time, contributed rendering overhead is limited without impacting the results.
To properly process the underlying mesh, the contained triangles has to be extracted, saved, and pushed onto the pipeline for run-time availability. Due to the data-driven nature of the system at hand, high level structures and components has to be filtered and interpreted to access the relevant data. This is a costly procedure, but a necessary one. The implied cost is only applied to the pre computations, which does not affect the run-time capabilities of the implementation. Local space vertex position, texture UV coordinates, and local space normals are accessible from the vertex element and used to further define the triangles. Mesh specific transforms is used to correctly project the triangles in world space, thus, creating a unified relation between all triangles. Parts of the computational components used by the ray tracer kernel at run-time can be pre computed at this stage. Components such as dot product `D = dot(planar normal, vertex v0)`), used to estimate the position of a triangle in respect to a ray with the following function `- (dot(N, orig) + D) / dot(N, dir)` and barycentric coordinates used to determine if a cast ray is inside a triangle or not. To prepare the triangles for BVH construction each triangle is represented by a bounding volume, which lets the BVH algorithm have a quantifiable comparison between the volumes in the scene.

3.3.2.1 The BVH implementation

When switching from IBL to ray tracing proxy geometry a different manual setup is required. In order to provide an easier workflow for artist, work was put into allowing the bounding volumes to update themselves when the underlying geometric data for the proxy geometry is changed. Previously, with IBL, the artists had to manually place cube maps in the game world and validate the results. With ray traced proxy geometry, all that the artists have to do is supply the new geometry to the application and the code for constructing the bounding volumes will be triggered. In the build phase of the game, loading a proxy geometry triggers the class `RaytraceProxyPolicy` if changes has been made to the geometry. This class then constructs a bounding volume per proxy geometry zone, which are pre defined, and stores the new addition with the geometry. By doing this it can be accessed after the building phase.

This part of the implementation is divided in two steps, first the BVH is constructed with an object structure where each node holds a reference to their children or leaves. Secondly, when the BVH is completed it is then converted to a GPU and pipeline friendly indexed version. In the indexed version the BVH tree is constructed as a main array with supplementing arrays that holds the triangles and the pre computed triangle data.
The BVH data is then passed through the pipeline of the engine and can be accessed and loaded on demand in run time. In order to load the BVH seamlessly in the game they are triggered to load on a per zone basis. This way there is no need to have more BVHs allocated in memory than what is necessary for the current calculations. When the data is processed through the pipeline the bounding volume structures and their containing primitives can be validated by drawing them in debug mode on top of the game world, as shown in Figure 3.2.

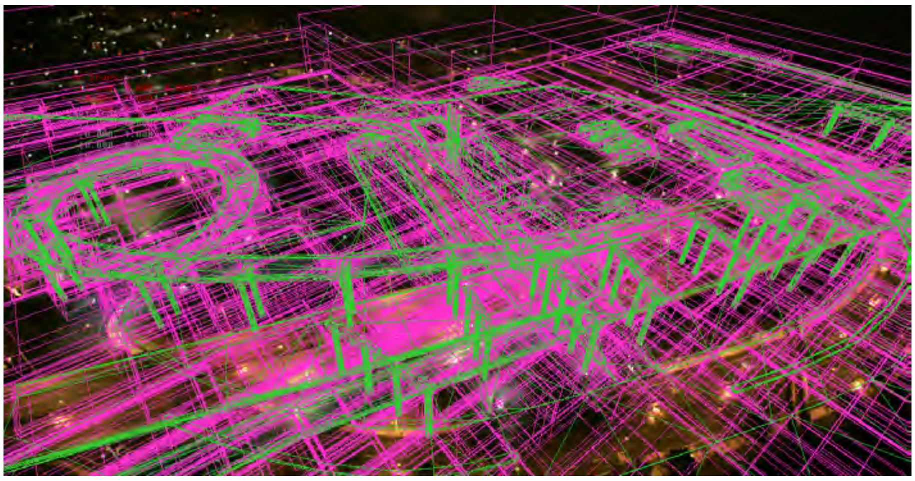


Figure 3.2: Debug view of triangles and bounding volumes. Triangles are green and bounding volumes are pink.

3.3.2.2 SAH based bounding volume construction

An issue when constructing a bounding volume hierarchy in a top-down method is deciding where to split the larger volumes into smaller ones. To solve for this problem, a surface area heuristic algorithm for fast bounding volume hierarchy construction is utilized. Overall, the algorithm slows the construction process. However, the intersection checks are quicker since the bounding volumes are optimally fitted. Since the bounding volume construction is pre computed, the speed of the construction is not prioritized but still desired. Wald [45] proposes an algorithm that uses a top-down greedy SAH construction. The geometry is recursively divided into non-overlapping spaces, as opposed to how a BVH normally partitions primitives without regarding overlapping spaces. The heuristic algorithm Wald proposes works as following: For N triangles in a sub-tree inside the volume V that is to be divided in the two parts L and R with N_L respectively N_R triangles and the volumes V_L and V_R. The estimated traversal cost for the sub-tree can be observed in equation 3.1.

$$Cost(C \rightarrow L, R) = K_T + K_I(\frac{(SA(V_L))}{(SA(V))}N_L + (\frac{(SA(V_R))}{(SA(V))})N_R) \qquad (3.1)$$

The $SA(V)$ is the surface area of the volume V and K_T and K_I represents implementation-specific constants for estimated cost of traversal and intersection. By utilizing this estimation on possible partition cases, the case with the minimum cost can be selected and the process repeated for each volume.

3.3.2.3 Pre compute intersection data

Triangle data, such as edges, world space normals, and uv coordinates are pre computed in compile time. This data is then passed through the pipeline formatted as arrays in order to be accessed at run time. This is important to the run time speed, since less calculations need to be performed by the kernels, thus, decreasing the computation time of the ray tracing. This comes at a small cost to the memory, however, in this case, the run time speed is prioritized.

3.3.3 GPU vs CPU

The implementation of the ray tracer leads to the question of developing it on the GPU or on the CPU. The architecture of each of the processing units has its own strengths and weaknesses and this has to be taken in consideration when deciding which sort of tracer will be developed. Figure 3.3 abstractly showcases the physical architecture of each processor. The size of each core relates to the power of the core itself in terms of, among other features, its operation frequency, ALU complexity, and latency efficiency.

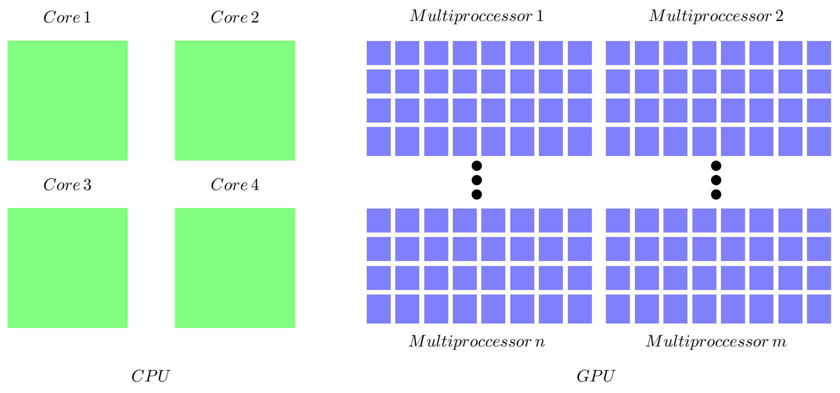


Figure 3.3: CPU (left) and GPU (right) illustration, the CPU has sequential high level ALUs while the GPU has scalable parallelizable ALUs, working against low level instructions.

By sequentially scheduling tasks with its large cache, CPUs are efficient when simultaneously handling different complex tasks on several different threads [38]. The GPUs strength on the other hand is that it is very proficient at processing large amounts of data, performing the same kernel operation on every data point in parallel. This structure is called SIMD (Single Instruction Multiple Data) [35], a common instruction schema used for GPU architectures. A critical difference between the two processing units is that programs developed for CPU use can have a contextual awareness of the state of the program. Therefore, it can use high level information and advanced data structures. GPU's on the other hand, with their low level, free standing, nature needs to have data in low level forms presented when each kernel is processed. This leads to that the necessary data for a kernel instruction has to be presented in low level structures. One way to supply this is by loading indexed arrays into buffers. In relation to the thesis project, the pre computed data, such as barycentric coordinates, needs to be pre processed and loaded in GPU favorable arrays. This in order to enable each thread to calculate off screen reflections. With these specifications in consideration, the project will move forward with implementing a GPU ray tracer.

```
while exists bvh to analyze{
   Pop bvh node from stack;
   if node is inner node {
      if ray intersects node bounding volume {
         Push left child onto stack;
         Push right child onto stack;
      }
   } else if node is leaf node {
      for each triangle in leaf node{
         if triangle is parallel or facing away from ray = break;
            // Backface culling
         if triangle is behind the origin of the ray = break;
         Calculate barycentric coordinates;
         if hit is outside triangle = break;
         Calculate hitdistance from difference between lineStart and hit;
         if hitdistance is shortest distance this far{
            Save hitdistance as shortest hit;
            Save best triangle and best hit;
         }
      }
   }
}
```

Figure 3.4: Pseudo code for calculating which primitive a ray hits.

3.3.4 Implementing ray tracing algorithm

The actual ray tracing is intended to be performed on the GPU. Its the only realistic option, since CPU ray tracing would be too slow for real time applications (Section 3.3.3). The implementation is intended to work as an extension to the existing screen space ray tracing algorithm. Therefore, it is only performed in cases where SSR rays fails to find a target. This method ensures that the traced rays are kept to a minimum amount, thus maintaining a low impact on the overall performance of the ray trace pass. Injecting the functionality at this stage also enables

the off screen reflection pass to benefit from functionality inherited from the screen space pass, such as filtering and sample weight for the currently analyzed pixel.
The ray tracing algorithm , as seen in Figure 3.4, starts off by initializing a stack that keeps track of the children of nodes hit. Initially the stack only contains the root of the BVH tree. While this stack is not empty, the ray will perform intersection tests on these nodes and add their children to the stack in the case of a hit. If the node hit is a leaf node, back face culling and intersection tests are run with the pre computed intersection data for all triangles in this node.
The triangle intersection is performed according to the intersection routine presented by Roman Kuchkuda [26] whom suggests that if the intersection of the current triangle is closer than any previously found, the triangle data and intersection distance is saved. After all the intersection checks are finished and the stack is empty, the closest triangle hit will be used to calculate the reflections.

```
if (lineDirInverse == 0){
  if (lineStart < aabMinValue || lineStart > aabMaxValue) return false;
} else {
  float T1 = (aabMinValue - lineStart) * lineDirInverse;
  float T2 = (aabMaxValue - lineStart) * lineDirInverse;
if (T1 > T2) {
  Swap T1 and T2 value;
}
  Tnear = max(Tnear, T1); // Always keep closest Tnear
  Tfar = min(Tfar, T2); // and farthest Tfar
  if (Tnear > Tfar) return false; // Ray misses bounding volume
}
```

Figure 3.5: Pseudo code for ray-box intersection.

In Figure 3.5 the implementation of the box intersection method is seen. This method is a variant of the method mentioned in Section 2.1.3, where instead of calculating the inverse of the line direction for each trace recursion, it is pre calculated before the algorithm enters the tracing logic. This optimization both makes the algorithm save some computational power and removes the need to check if the calculation divides with zero at any time. Overall, this makes it more streamlined and efficient.

3.3.5 Filtering

The already existing filtering system that is part of the SSR generates pixel color depending on material roughness, incoming radiance, and what values the neighbors has had temporally using neighborhood clamping. The intended implementation is meant to work closely with the SSR and override the logical implication of how the system will handle a missing ray inside the SSR kernel. This leads to a natural adaptation of the filtering used by the SSR to generate coherently filtered off screen reflections.

4
Results

To appropriately evaluate the results of this project each research question needs a comparable measure point. The implementation process consisted of continuous discussion with supervisors and engineers at Ghost and Frostbite. This to ensure that development adheres to similar solutions already present in the engine, and by proxy, assuring that a prototype can be correctly and seamlessly integrated with the current SSR implementation. So by following this mentality, the implied measurement if the research question is answered or not is the current available sub-systems and the use of said subsystems to enable and aid the intended reflection system.
when handling tracing algorithms and acceleration structures the measure point for validation is the available memory, frame budget, and efficiency of said algorithms in comparison with the current IBL implementation. These constraints implied that work has to be done in order to both stream the data structures in the correct fashion, optimize this process, and optimize said tracing algorithms.
To correctly measure if the third and final question, “Do the reflections produced by the proxy geometry sufficiently replace the IBL implementation in consideration to image quality, while maintaining the same or better performance? ”, is correctly answered, the produced reflection is aimed to correctly reflect the approximated geometry. This leads to the approximated geometry to be used as the benchmark if the implementation is fulfilling the proposed research question or not, which can be observed in Figure 4.1.

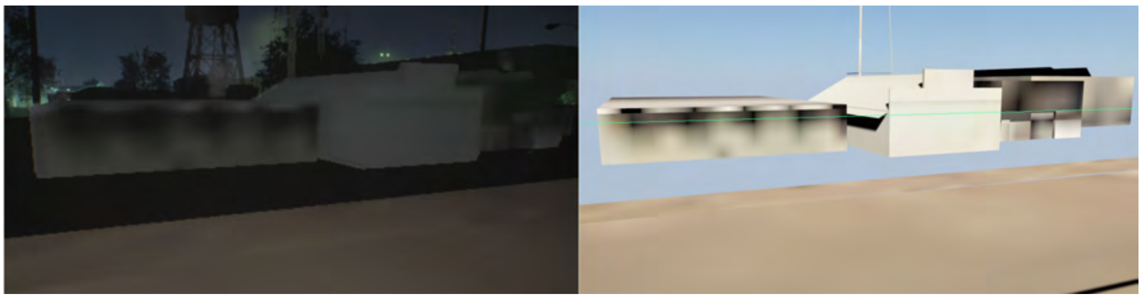


Figure 4.1: Proxy geometry (left) rendered from eye next to the textured source mesh (right) present in the Need for Speed game. The images are taken from a similar but not identical viewpoint.

4.1 Quality of the proxy geometry

The quality of the proxy geometry and the textures that are associated with it is highly influential on the final visual result. As the proxy geometry gets less simplified and more like the original geometry and the textures are a closer match to those used in the original model, the better the results will be. In the cases where the geometry is kept very simple and the textures clearly do not match the original texturing, the reflections will look inaccurate, as is shown in Figure 4.2. Optimally the textures used will have pre-baked lighting, but in the prototype only the diffuse coloring was used. The problem with the lighting of the textures is easily fixed without adding extra memory or computing costs in run time. In order to improve the texturing quality the textures would have to be made larger, which in turn will make the method more heavy on the memory streaming. However, the coloring of the textures can clearly be improved from their current state, which would greatly improve the quality of the proxy reflections. There are some cases where the geometry is overly simplified and gets culled, e.g. small cylindrical shapes can be represented by triangles facing only one direction, so they are not visible in the reflections.

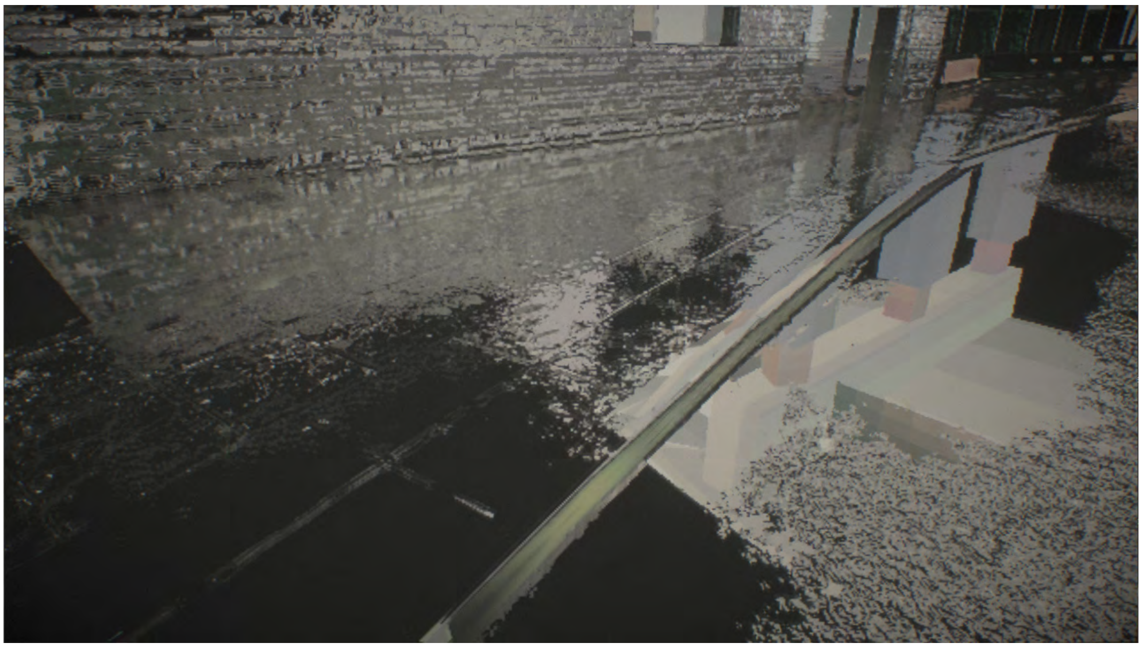


Figure 4.2: Mismatched proxy geometry, where the house does not exist in the generated proxy geometry. The texturing of the bridge pillars is also badly represented.

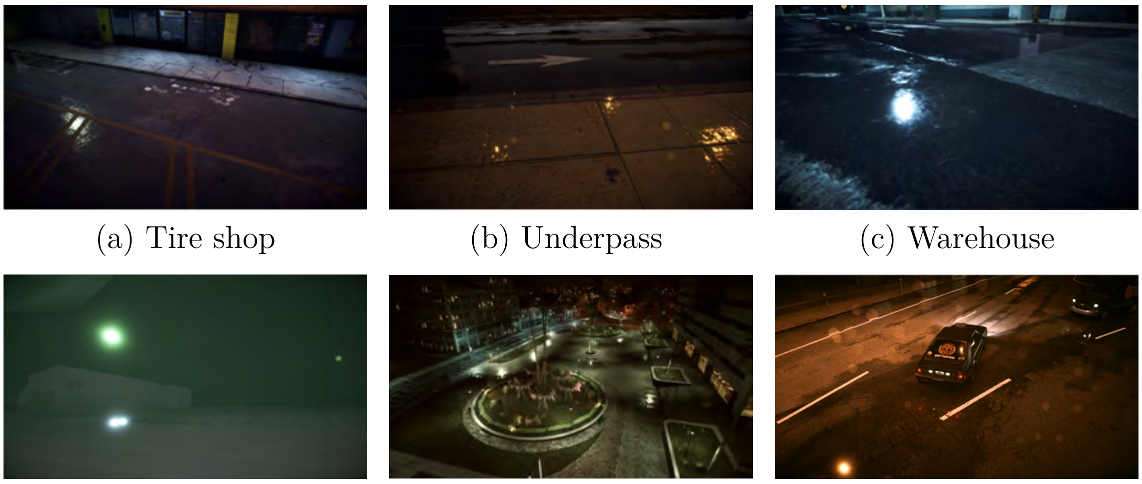
4.2 Memory costs

As mentioned the BVHs and textures are streamed on a per zone basis. The only data in active memory at all times are the BVH for the zone, the triangle data and a 512*512 texture belonging to the approximated mesh present in the engine (the applied texture can be observed in Figure 4.1). The average amount of memory used by the proxy geometry is shown in Table 4.1. Whereas the method utilizing IBL constantly has 41 cube maps with the size 128*128 loaded for each side of the cubes with an additional mipmap level. This is summed up with 41*128*128*6*8*(4/3) bytes to 42991508 bytes.

Data	*Memory Cost*
Texture	786432 *bytes*
BVH	155250 *bytes*
Index Buffer	15028 *bytes*
Delta Buffer	15028 *bytes*
Edge Buffer	45084 *bytes*
Normal Buffer	45084 *bytes*
UV Buffer	30056 *bytes*
Total	1091962 *bytes*

Table 4.1: A table showcasing the average memory cost for each data component that is loaded for the proxy ray tracing implementation.

Except for taking up less memory space, which is approximately 1/20 of the previous space, the proxy geometry zones also cover a larger area compared with the cube maps.


(a) Tire shop (b) Underpass (c) Warehouse

(d) Mirror window (e) Plaza (f) Bridge

Figure 4.3: This collection of figures are sample scenes for performance measurement. (a) and (b) are general scenes where a lot of off screen reflections occur. (c) shows how a warehouse is reflected in a mirror-like wet street, where (d) shows how the entire scene is ray traced when looking into a window that creates mirror-like reflections. (e) shows the efficiency in a large scene and (f) shows how reflections are created when driving under a bridge.

4.3 Performance measurement

The implementation has been tested on NVIDIA GeForce GTX770 in a 1280x720 resolution as well as on a PlayStation 4 console. The bounding volume hierarchies are pre-constructed on a PC with an Intel Xeon 3.50 GHz CPU and 32 GB RAM. There exists 167 zones containing geometry in Need for Speed that the implementation needs to create a BVH structure for. In total the zones contain 627411 triangles and the total BVH construction time for all the zones is 453 seconds, with an average of 2.7 seconds per zone.
Performance has been tested throughout different scenes and six varying scenes has been chosen to represent this. The scenes that are shown in Figure 4.3 show average, best, and worst case scenarios as well as varying reflection types.

	Tire shop	*Underpass*	*Warehouse*	*Mirror window*	*Plaza*	*Bridge*
bv[1]	2983	5599	1997	2495	4003	3415
tri[2]	3322	6085	2214	2716	4331	3688
1	1.19ms	0.97ms	1.70ms	1.05ms	1.07ms	1.83ms
2	2.37ms	3.59ms	2.85ms	1.68ms	1.48ms	4.52ms
3	1.92ms	2.70ms	2.20ms	N/A	2.97ms	2.48ms
4	2.37ms	3.59ms	2.85ms	1.68ms	1.48ms	4.52ms
5	3.60ms	4.83ms	3.94ms	3.83ms	3.87ms	6.97ms
6	7.42ms	17.07ms	7.82ms	7.22ms	8.28ms	19.31ms

Table 4.2: Row 1 represents the existing implementation (SSR & IBL fallback), row 2 is the proposed solution this project has resulted in, row 3 is a pass when the proxy geometry is generated from the eye. Row 4 is the existing SSR solution combined with color data from the proxy geometry (no filtering). Finally row 5 and 6 are the same settings as 1 and 2 running on a PS4 system.

The Table 4.2 shows some interesting benchmark results. The Underpass and the Bridge images represents the worst case scenarios for the algorithm. In these cases the SSR fails to find any hits for most of the pixels, so ray tracing is performed for almost the entire scene, and most of the rays intersects with primitives. The reason that the Bridge shows slightly worse performance results compared to the Underpass is likely that the rays in the Bridge scene intersects with more primitives than the underpass, therefore, more intersection testing needs to be performed. The results in the Tire shop and the Warehouse images shows a performance decrease by roughly 1.2 ms on GTX770 and about 3.9 ms on PS4. These results also represents the average performance decrease that has been detected throughout the environment. The plaza shows a complex scene with open space. In this case the SSR identifies most of the reflections, therefore, only small parts of the scene is actually ray traced. All 6 scenes displayed in Figure 4.3 are performed in 30 frames per second on PS4.

[1]The number of bounding volumes in the BVH
[2]The number of triangles in the BVH

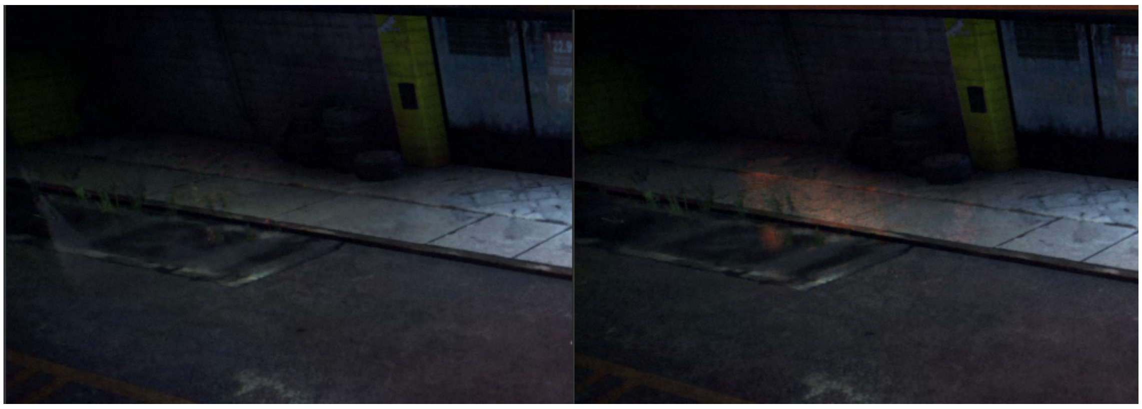


Figure 4.4: Unnatural emissive reflections when using IBL (right) compared to our method using ray tracing (left).

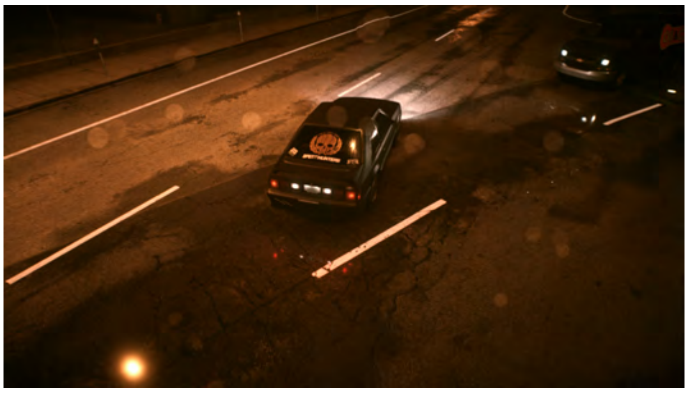


Figure 4.5: Original image using SSR and IBL, without ray tracing the proxy geometry.

4.4 Image quality

The overall results of the final image is of a decent quality that clearly reflects the cruder shapes of the proxy geometry. When compared to the previous method of parallax corrected cube maps [27] the output is more consistent and the reflections do not get unnatural offsets as can be problematic with larger cube maps which is shown in Figure 4.4. The current state of the resulting renderings show a visible difference when transitioning from SSR reflections to the proxy ray traced reflections, as could be expected. This is since the proxy textures varies in coloring from the original textures. A thorough comparison of IBL and our solution is shown in Figure 4.10.

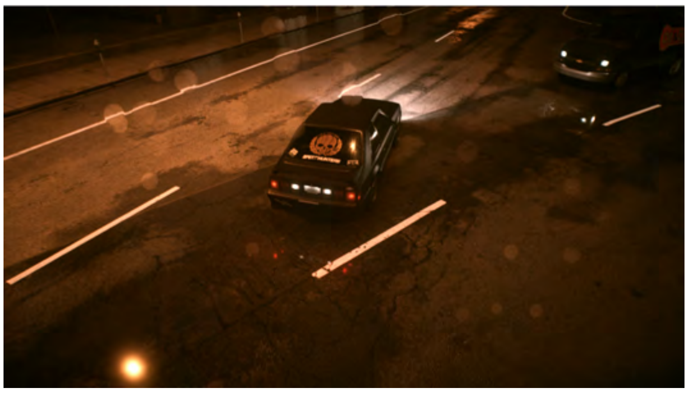


Figure 4.6: Image using SSR and proxy ray tracing with filtering.

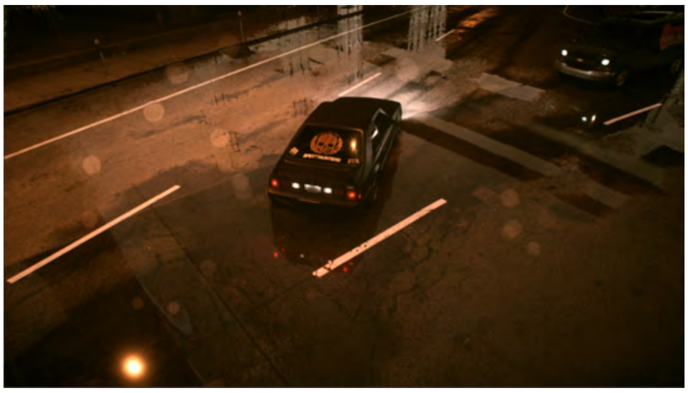


Figure 4.7: Image using SSR and proxy ray tracing

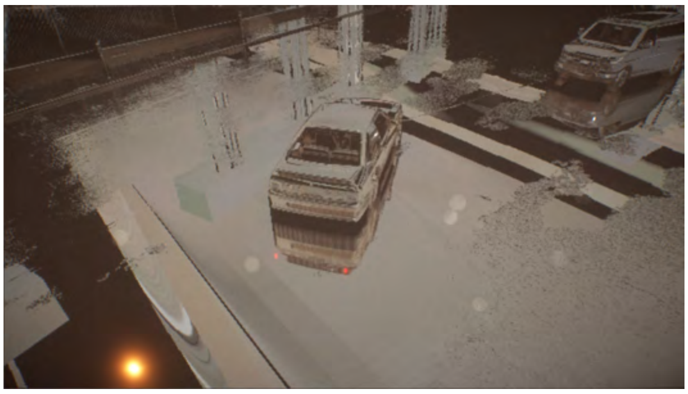


Figure 4.8: Image using SSR and proxy ray tracing, with mirror like reflections.

Using ray tracing against proxy geometry gives more control of which reflections will be visible when compared IBL. Comparing Figure 4.5 with 4.6 the most notable difference is how the IBL fails to produce reflections of the bridge pillars that are seen in the top middle when using ray tracing. The reflections shown in Figure 4.7 shows a clearer image of all objects that are taken into account before filtering is applied, where these are all objects that the IBL solution does not take into account when creating reflections. In Figure 4.8 the poorer quality of the texture and modeling

in the proxy geometry is visible. The texturing of the proxy geometry is monotone and does not match the standards of the actual models in the game world.
The proposed implementation correctly renders the seam between screen space reflections and off screen reflections. In situations with dynamically lit environments the disparity between the actual geometry of the world and the off screen approximation would become more explicit. The seam comparison can be observed in Figure 4.9 where the difference in accuracy is apparent.

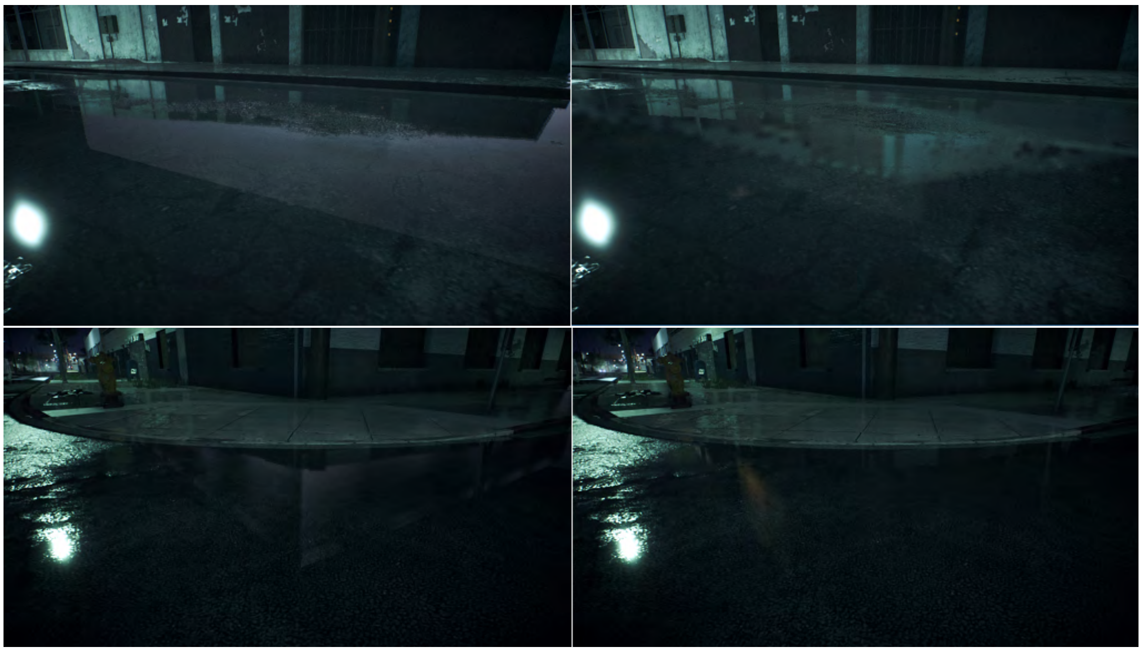


Figure 4.9: Images using proxy ray tracing (left column) and IBL cube maps (right column) to showcase how each implementation handles seaming screen space reflections together with off screen space reflections.

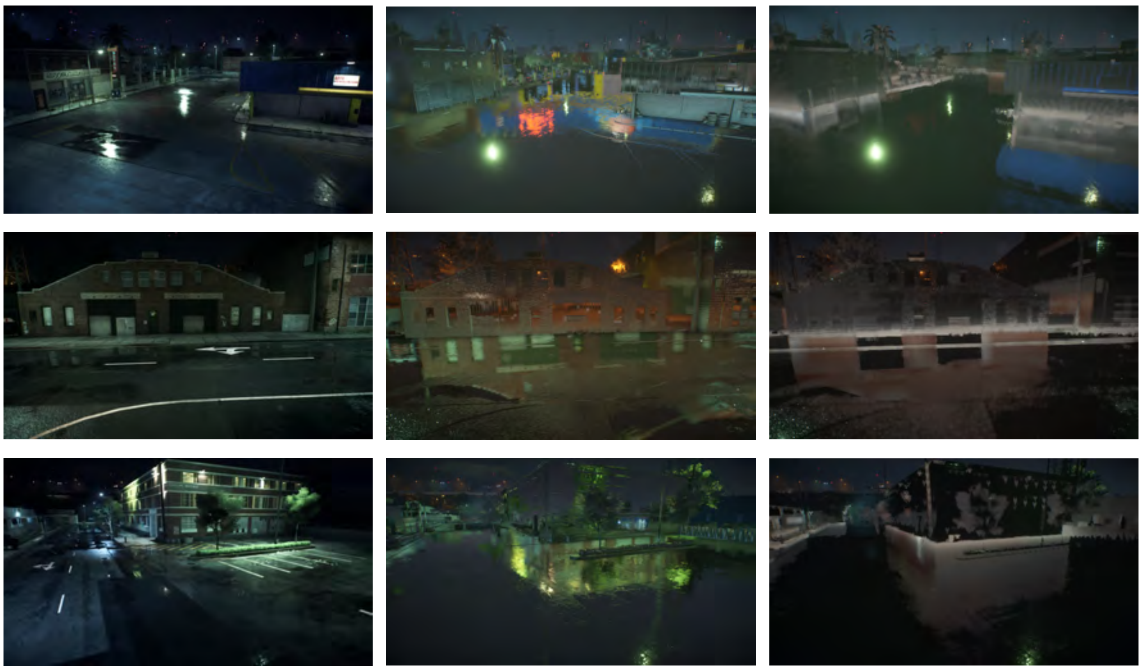


Figure 4.10: The presented figures are three scenes (one per row) taken from the same angle with three different reflection settings (one per column). The left column is SSR with off screen proxy reflections. The middle column is IBL mirror reflections and the right column is mirror proxy reflections. The first row showcases the large difference in reflection placement the IBL can result in compared to the proxy solution. While producing correct emissive reflections, worse case scenarios like this results in a detrimental accuracy to the reflected color. The second row showcases a decent approximation of the IBL solution, but as seen on the right side of the picture stretches out in an unnatural way. The third row is an optimal angle for the IBL solution, taken close to the probe position for the cube map. Thus, this angle produces accurate reflections of the building. The proxy reflection however also produces similar results and with correct texturing and lighting would look very similar to the results of the IBL solution.

5
Conclusion and Discussion

The study has in correlation with the research questions shown a novel solution on how ray traced proxy geometry can be integrated with the SSR implementation present in Need for Speed. The solution is a memory conservative approach, reusing already loaded geometry and streaming logic. This approach follows the guidelines specified by the stakeholders (Section 1.2), where automatically generated off screen reflections as well as the possibility for dynamic lighting in the future is prioritized. This while still keeping a low memory overhead (Section 4.2). The image result of the off screen reflections is not coherent with the SSR implementation. However, compared to the proxy geometry mesh that the reflection is based of, the solution correctly reflects the source content, as is shown in Figure 4.1. This implies that it is beneficial to further explore this subject. We believe that proxy tracing with high texture quality and lighting would with high probability result in high quality off screen reflections.

The ray tracing implementation also manages to render the transition between proxy reflection and SSR reflection without the seam visible when using IBL, as is shown in Figure 4.9. Therefore, it is an accurate and dynamic replacement for the IBL implementation. We have also shown through the methods we use how to produce an acceptable frame rate with our image quality. As mentioned, the image quality can be increased with the quality of the texture without affecting the computational cost. To answer the last question, “do the reflections produced by the proxy geometry sufficiently replace the IBL implementation in consideration to image quality, while maintaining the same or better performance”, we believe it does in some regards. As of now, with the texture coloring not being fully utilized, the changes between the SSR and the traced proxy geometry is too clear. The performance of the ray tracing algorithm is costly compared with IBL, but it is more memory efficient. However, with further work on this implementation it can prove to be an excellent replacement for IBL that not only occupies less memory, but also provides less work for artists.

The memory savings shows great results, as it on average occupies up about 1/20 of the space that the IBL implementation utilizes. With this in mind, if the memory streaming where not to be a concern, the quality of the proxy geometry could in average be approximately doubled and still be as memory efficient as the IBL. If the zones that are currently being utilized were to be split in smaller parts, only the relevant ones would need to be loaded. This can lead to less unnecessary geometry being loaded and further increase the streaming potential. We have noticed that even though all proxy geometries are coupled with a 512*512 texture, some of the geometries only occupies 256*512 of the texture. This shows that in some cases the

texture resolution could be increased without altering the memory costs.
In the worst cases that has been found show a performance drop from around 8 ms to 20 ms on a PS4 console. We believe that these cases can be optimized further by performing further integration work between the SSR and proxy ray tracing shader code. However, even in the worst cases, acceptable frame rates are still achieved.
As of the build speed of the bounding volumes, we believe it can be highly optimized. The build times could be lowered by optimizing the calculations and parallelizing the build tasks. However, this was not prioritized during the work of this thesis since only the run time performance was in focus.
The image quality is varying with how the proxy geometry is approximated. In some cases the texturing is better matched compared to other cases, where, e.g. white texture is used as an approximation of gray buildings. We do not dispute that the results would be improved with better texture quality and increased fidelity in the proxy geometry. However, the quality of the texturing could be improved while still utilizing the same amount of memory. Another way to improve the textures would be to do a single light pass over the proxy geometry and add some lighting to the textures, as they currently are very dark. Adding light would also help making the proxy geometry a closer match to the real geometry. Ray tracing proxy geometry does produce results that are more correct and not as positional based as parallax corrected cube mapping.
There are cases where clear artifacts are visible that would not be there if the actual geometry is ray traced. When tracing the proxy reflections that differ in size from the actual geometry where e.g. a roof is set higher on a building, the reflection that normally could be traced entirely with SSR is filled in with a mismatched proxy reflection on top of it. Another issue is the opposite and more frequently occurring case, where the proxy geometry is smaller than the actual geometry. This leads to cases where the proxy trace fails to find hits where there actually should be geometry.

6 Future work

The implemented prototype is dependent on the zone switching triggers that handles skyline visibility (Section 3.3.2.1). While this is beneficial for prototyping and evaluating the functionality of the off screen reflection solution, this area could be researched for memory and build efficiency benefits. If the zones were to be split up in smaller areas, they could then be incrementally replaced when the camera moves towards other areas. This would decrease the stress of loading an entire zone at once. Having smaller zones replaced depending on the camera position will also increase the smoothness of transitions when calculating reflections close to the edge of a zone. The resolution and efficiency of the BVH implementation is sufficient (Section 4.2), but this subject could be further explored and optimized. Hybrid approaches such as the LBVH proposed in Section 3.3.2.1 could be implemented and tested for further optimizations of the tracing kernel. For future implementations where light maps and direct illumination, e.g. from the sun, could be implemented. Worst case scenarios, such as the Underpass in Figure 4.3, would benefit of optimization operations like a termination expression relevant to the importance of the analyzed scene.

Bibliography

[1] Ghost games is an ea games studio. http://ghostgames.com/. (Accessed on 05/10/2016).

[2] Need for speed - official site - us. http://www.needforspeed.com/. (Accessed on 05/10/2016).

[3] Parallel programming and computing platform | cuda | nvidia | nvidia. http://www.nvidia.com/object/cuda_home_new.html. (Accessed on 05/30/2016).

[4] Peter and karl's gpu path tracer blog. http://gpupathtracer.blogspot.se/. (Visited on 01/27/2016).

[5] Playstation 4 hardware specs - playstation 4 wiki guide - ign. http://www.ign.com/wikis/playstation-4/PlayStation_4_Hardware_Specs. (Visited on 01/26/2016).

[6] Remember me (official). http://www.remembermegame.com/. (Accessed on 05/09/2016).

[7] Renderer 2.x - porting to cuda (one month later). https://www.thanassis.space/cudarenderer-BVH.html#phase2. (Accessed on 05/15/2016).

[8] Robbin marcus: Real-time raytracing part 2. http://robbinmarcus.blogspot.se/2015/10/real-time-raytracing-part-2.html. (Accessed on 05/16/2016).

[9] A smooth and collaborative game development experience - frostbite. http://www.frostbite.com/about/mission/. (Accessed on 05/10/2016).

[10] Xbox one hardware specs - xbox one wiki guide - ign. http://www.ign.com/wikis/xbox-one/Xbox_One_Hardware_Specs. (Visited on 01/26/2016).

[11] T. Aila and S. Laine. Understanding the efficiency of ray traversal on gpus. In *Proceedings of the conference on high performance graphics 2009*, pages 145–149. ACM, 2009.

[12] J. Amanatides, A. Woo, et al. A fast voxel traversal algorithm for ray tracing. In *Eurographics*, volume 87, page 10, 1987.

[13] A. Appel. Some techniques for shading machine renderings of solids. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 37–45. ACM, 1968.

[14] K. Bjorke. Image-based lighting, 2004.

[15] S. Boulos. Notes on efficient ray tracing. In *ACM SIGGRAPH 2005 Courses*, page 10. ACM, 2005.

[16] H. Carr, J. Snoeyink, and M. van de Panne. Simplifying flexible isosurfaces using local geometric measures. In *Proceedings of the conference on Visualization'04*, pages 497–504. IEEE Computer Society, 2004.

[17] M. G. Chajdas. *A voxel-based visualization pipeline for high-resolution geometry.* PhD thesis, München, Technische Universität München, Diss., 2015, 2015.

[18] C. Ericson. *Real-time collision detection.* CRC Press, 2004.

[19] A. S. Glassner. Space subdivision for fast ray tracing. *Computer Graphics and Applications, IEEE*, 4(10):15–24, 1984.

[20] N. Greene. Environment mapping and other applications of world projections. *Computer Graphics and Applications, IEEE*, 6(11):21–29, 1986.

[21] E. Hermann, F. Faure, and B. Raffin. Ray-traced collision detection for deformable bodies. In *GRAPP 2008-3rd International Conference on Computer Graphics Theory and Applications*, pages 293–299. INSTICC, 2008.

[22] M. J. Hopcroft and A. Spyridi. Backface primitives culling, Mar. 19 2002. US Patent 6,359,629.

[23] T. Ize, P. Shirley, and S. Parker. Grid creation strategies for efficient ray tracing. In *Interactive Ray Tracing, 2007. RT'07. IEEE Symposium on*, pages 27–32. IEEE, 2007.

[24] J. Kalojanov, M. Billeter, and P. Slusallek. Two-level grids for ray tracing on gpus. In *Computer Graphics Forum*, volume 30, pages 307–314. Wiley Online Library, 2011.

[25] M. R. Kaplan. The use of spatial coherence in ray tracing. *Techniques for Computer Graphics*, pages 173–193, 1987.

[26] R. Kuchkuda. An introduction to ray tracing. *Theoretical Foundations of Computer Graphics and CAD, Italy*, 1987.

[27] Z. Lagarde. Local image-based lighting with parallax corrected cubemap. GameConnection, 2012.

[28] S. Laine and T. Karras. Efficient sparse voxel octrees. *Visualization and Computer Graphics, IEEE Transactions on*, 17(8):1048–1059, 2011.

[29] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. Fast bvh construction on gpus. In *Computer Graphics Forum*, volume 28, pages 375–384. Wiley Online Library, 2009.

[30] D. Luebke and S. Parker. Interactive ray tracing with cuda. *Technical presentation, http://www. nvidia. com/content/nvision2008/tech_ presentations. html. PDF URL: http://www. nvidia. com/content/nvision2008/tech_presentations/Game_Developer_Track/NVISION08-Interactive_Ray_ Tracing. pdf*, 2008.

[31] J. D. MacDonald and K. S. Booth. Heuristics for ray tracing using space subdivision. *The Visual Computer*, 6(3):153–166, 1990.

[32] M. McGuire and M. Mara. Efficient gpu screen-space ray tracing. *Journal of Computer Graphics Techniques*, 2014.

[33] B. F. Naylor. Binary space partitioning trees. *Handbook of Data Structures and Applications*, pages 20–1, 2005.

[34] A. Neubauer, L. Mroz, H. Hauser, and R. Wegenkittl. Cell-based first-hit ray casting. In *Proceedings of the symposium on Data Visualisation 2002*, pages 77–86, 2002.

[35] D. A. Patterson and J. L. Hennessy. *Computer organization and design: the hardware/software interface.* Newnes, 2013.

[36] S. Pixel. Ray tracing: Rendering a triangle. `http://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle`, 2012.

[37] D. Pohl. Light it up! quake wars gets ray traced. *Intel Visual Adrenaline*, (2), 2009.

[38] A. Rege. An introduction to modern gpu architecture. *En ligne]*, 2008.

[39] B. Smits. Efficiency issues for ray tracing. In *ACM SIGGRAPH 2005 Courses*, page 6. ACM, 2005.

[40] M. Stich, H. Friedrich, and A. Dietrich. Spatial splits in bounding volume hierarchies. In *Proceedings of the Conference on High Performance Graphics 2009*, pages 7–13. ACM, 2009.

[41] L. D. Stone. Theory of optimal search. 1975.

[42] K. G. Suffern and K. Suffern. Glossy reflection. In *Ray Tracing from the Ground up*, chapter 25, pages 529–542. AK Peters, 2007.

[43] L. H. Sébastien Lagarde. The art and rendering of remember me. Game Developers Conference Europe, 2013.

[44] B. Vinkler, Havran. Bounding volume hierarchies versus kd-trees on contemporary many-core architectures. 2014.

[45] I. Wald. On fast construction of sah-based bounding volume hierarchies. In *Interactive Ray Tracing, 2007. RT'07. IEEE Symposium on*, pages 33–40. IEEE, 2007.

[46] I. Wald and V. Havran. On building fast kd-trees for ray tracing, and on doing that in o (n log n). In *Interactive Ray Tracing 2006, IEEE Symposium on*, pages 61–69. IEEE, 2006.

[47] I. Wald, T. Ize, A. Kensler, A. Knoll, and S. G. Parker. Ray tracing animated scenes using coherent grid traversal. In *ACM Transactions on Graphics (TOG)*, volume 25, pages 485–493. ACM, 2006.

[48] L. Williams and R. Kessler. *Pair programming illuminated.* Addison-Wesley Longman Publishing Co., Inc., 2002.

[49] K. Zhou, Z. Ren, S. Lin, H. Bao, B. Guo, and H.-Y. Shum. Real-time smoke rendering using compensated ray marching. In *ACM Transactions on Graphics (TOG)*, volume 27, page 36. ACM, 2008.