The Open University of Israel
Department of Mathematics and Computer Science

A summary of real time ray tracing techniques in video games and simulations

Final Paper submitted as partial fulfillment of the requirements
Towards an M.Sc. degree in Computer Science
The Open University of Israel
Department of Mathematics and Computer Science

By
Amit Ofer

Prepared under the supervision of Dr. Mireille Avigal

March 2023

Table of Contents

List of Figures 4
1. Abstract 5
2. Introduction 6
3. The graphics pipeline 8
3.1. Rasterization and the standard graphics pipeline 8
3.2. Ray Tracing 12
3.3. Shader Types 13
3.4. Acceleration Structures 15
3.5. Coherency 16
4. Denoising 18
4.1. Spatial denoising 19
4.2. Temporal denoising 20
4.3. DLSS & AI based denoisers 20
5. Shadows 22
5.1. Shadow Maps 22
5.1.1 Disadvantages of shadow mapping 23
5.1.2 Soft Shadow Techniques 26
5.1.3 Fixed sized penumbra filtering techniques 27
5.1.4 Variable sized penumbra filtering techniques 28
5.2. Screen Space Shadows 29
5.3. Ray-Traced Shadows 30
5.3.1 Ray traced contact shadows in “Control” 38
6. Reflections 41
6.1. Planar reflections 41
6.2. Screen space reflections (SSR) 42
6.3. Stochastic Screen Space Reflections 47
6.4. Ray Traced Reflections 47
6.5. Ray traced reflections in “*Control”* 50
7. Ambient Occlusion 53
7.1. Screen Space Ambient Occlusion (SSAO) 54
7.2. Horizon Based Ambient Occlusion (HBAO) 56
7.3. Ground Truth Ambient Occlusion (GTAO) 59

7.4. Ray Traced Ambient Occlusion (RTAO) 61
8. Summary 64
9. Future work 65
10. References 66

List of Figures

Figure 1 – The rendering pipeline 8
Figure 2 – Geometry Processing 10
Figure 3 – Rasterization and Pixel Processing 11
Figure 4 – Ray tracing camera setup 13
Figure 5 – Ray tracing shaders flow 15
Figure 6 – Ray Tracing Acceleration Structures 16
Figure 7 – Example of using a denoiser 18
Figure 8 – Shadow mapping technique diagram 23
Figure 9 – Shadow mapping acne 24
Figure 10 – Shadow mapping Peter Panning artifact. 25
Figure 11 – Shadow Mapping aliasing 26
Figure 12 – Hard shadows vs. soft shadows 27
Figure 13 – PCSS block search diagram 29
Figure 14 – Ray tracing – shadow rays 32
Figure 15 – Hybrid ray traced shadows 34
Figure 16 – Hybrid ray traced shadows results 35
Figure 17 – Sampling mask matrix 37
Figure 18 – Ray traced shadows performance results 38
Figure 19 – Ray traced shadows results in Control 39
Figure 20 – Screenshots of the frames from Control used for performance measurements 40
Figure 21 – Screen space reflections algorithm flow chart 46
Figure 22 – Ray traced reflections algorithm flow chart 49
Figure 23 – Screenshots from Control showing the diference between Screen Space Reflections and Ray Traced Reflections 52
Figure 24 – Screenshot showing the importance of Ambient Occlusion 53
Figure 25 – Diagram explaining the SSAO technique introduced by Crytek 54
Figure 26 – Screenshot showing the results of the SSAO technique introduced by Crytek 55
Figure 27 – Diagram explaining Normal oriented hemisphere for AO 56
Figure 28 – Diagram explaining HBAO and GTAO 58
Figure 29 – Screenshot of a crown model rendered with 4 samples per pixel vs the same model rendered with 2048 samples 61
Figure 30 – Screenshot showing AO results of uniformly distributed samples vs cosine weighted samples 62
Figure 31 – Screenshot showing the results of RTAO before and after denoising 63

1. Abstract

Until recently ray tracing based rendering techniques were only possible in the movie and animation industry as performance is less crucial there. For those types of projects each frame can take a long time to render and multiple frames could be split across multiple computers in a large rendering farm and then composed together to make a whole film.

In video games, performance is critical and a single computer has to render 30-60 frames per second (or even more if we consider VR games) in order to give the player a smooth experience. Therefore, video game developers had to rely on faster techniques that only approximate real world physical behavior in order to achieve high quality visuals while staying within the budget of having only a few milli-seconds to render each frame. Those techniques are based on the rasterization pipeline on the GPU, which will be explained later in this work.

In the last couple of years, there's been a huge development in Graphics hardware and Ray-Tracing dedicated hardware was introduced as part of the new lines of GPUs. Those recent developments brought back the possibility of using Ray tracing along with already used techniques in order to push realism and graphical fidelity even further. Many video game companies, hardware companies (mainly Nvidia and AMD) and researchers in the field of computer graphics began researching on how the new and improved hardware could be used in the next generation of video games.

In this work we discuss a few different rendering techniques that could benefit from using Ray-Tracing and explain how this could possibly be achieved. The following techniques that will be discussed in this work are: Ambient Occlusion, Shadows and Reflections. This work will also discuss denoising techniques that could be used in real time, as these are an integral part of using ray tracing.

2. Introduction

The topic of this work is ray tracing techniques for real-time applications. Ray tracing is a research area in computer graphics which is inspired by the physics of light. As such, its goal is to simulate the way lights interact with materials just as in real life and as a result generate more realistic images.

Ray tracing isn't a new area and some of the research that is done today is based on articles that were published in the 80s such as [1], [2]. In the late 90s animation studios began applying ray tracing techniques in their animation films and the film "A bug's life" by Pixar\Disney that released in 1998 was the first animated film to use ray tracing for some parts of its rendering process. As those techniques kept evolving and hardware became faster, animation films relied more on it for its film production. The film "Monster University" by Pixar that released in 2013 was the first Pixar film to fully rely on Ray tracing for its lighting.

Ray tracing-based rendering techniques were always considered to be slow and to take a lot of computation power and therefore until recently were only used by film production where real time was not important and frames could be rendered over a long period using several computers in a network. Applications that require real-time performance such as video games and simulations rely on techniques that only approximate the physics of light and are therefore less accurate and in many cases less robust and require fine tuning. With recent changes and innovation in computer graphics hardware and APIs ray tracing is slowly becoming viable for those applications as well.

That opened the door to researchers and tech\video game companies to spend a lot of resources on adapting ray tracing techniques to be used in real time and maybe books were written on the subject in the last few years. "Real-Time Rendering" 4th edition by Akenine-Moller Tomas et al. [3] covers a wide variety of computer graphics topics and the latest edition includes an online chapter that covers real time ray tracing. Peter Shirley's ray tracing book series [4] covers a wide area of ray tracing specific topics and discuss both the mathematical and technical aspects of building a ray tracer. The book series "Ray Tracing Gems" [5], [6] incorporate a large number of articles on different ray tracing topics from different researchers and companies across the industry. Ray tracing is also a big focus in Real-time rendering courses in SIGGRAPH conferences in recent years.

As we will discuss later in, ray tracing in real-time can't work without having good real-time image denoising solutions and therefore a big part of ray tracing research in recent years involve research in real-time denoisers and super sampling techniques. This ties in with recent research and innovation in the area of deep learning as a big part of the research in the area of image denoising is AI based image denoisers.

In chapter 3 of this work, we will discuss the standard graphics pipeline that uses rasterization to render images, we will then describe the recent additions to the graphics pipeline which adds hardware ray tracing support and will discuss the main differences between the two ways of using the pipeline. We will also discuss the data structures used for ray tracing and how keeping rays coherent is a big part of having good performance when employing ray tracing.

In chapter 4 we will introduce some of the denoising techniques that will be mentioned in the main part of this work when discussing the main techniques presented in this work.

In chapters 5-7 we will discuss 3 different rendering techniques where ray tracing can be used to enhance the visual quality of the final image and help achieve a more photorealistic look (shadows, reflections and ambient occlusion). Each of these chapters will begin by describing how the technique was implemented using rasterization, it will also explain the problems and drawbacks of each technique. It will then describe how ray tracing can be used to enhance these techniques by either using a fully ray traced algorithm or a raster-ray tracing hybrid technique. This work will also include some practical examples of how those techniques were applied in some video games and will briefly mention performance measurements for some of these examples.

The last part of this work will summarize and discuss some of the areas in real-time ray tracing that were not mentioned in this work and describe the areas of future work and research.

3. The graphics pipeline

3.1. Rasterization and the standard graphics pipeline

Rasterization and Ray Tracing are both techniques that take a 3D world as an input (represented as a collection of points, normals, uv coordinates, textures etc.) and use it to create a 2D image as an output that corresponds to a specific view of the 3D world (depending on where the virtual camera is located). Although ray tracing can be performed using a compute or a pixel shader as part of the traditional rasterization-based pipeline, this work is going to focus on ray tracing using the new hardware accelerated APIs introduced recently such DirectX 12 DXR, Vulkan Ray Tracing etc. Therefore this section will give a brief introduction to how the traditional rendering pipeline works and compare it with the newly introduced ray tracing pipeline. The following sections will describe how some rendering techniques are created using the traditional rendering pipeline and how they can take advantage of the new ray tracing pipeline or a mix of both.

The graphics rendering pipeline [7] can be logically divided into 4 main components: Application, Geometry Processing, Rasterization and Pixel Processing (Figure 1), a brief description of these components will follow. While each stage relies on the previous stages for input, each of these components can run in parallel on different data i.e., multiple pixels computed at the same time. As the GPU is a highly parallel processor each logical component is comprised of multiple Hardware units capable of processing a large amount of data in parallel.

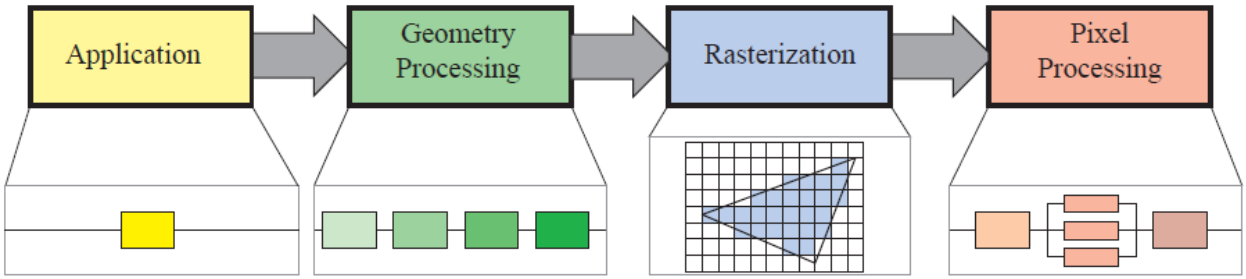

Figure 1 - The rendering pipeline. Image taken from [3]

Application – The application stage is the first stage of the pipeline and is executed on the CPU, therefore giving the developer full control. This stage mainly sets up the commands that are going to be executed on the GPU in later stages. Since this stage is executed on the CPU it is not parallel as the other stages (can only be slightly parallel with utilizing multiple CPU cores). This stage sets up everything to be executed later on the GPU, such as which shaders to run, the type of primitives to render (i.e., points, lines, triangles, etc.), data that is going to be used in the shaders (i.e., textures, constant buffers, samplers, etc.), vertex buffers, render targets and more. In this stage the developer is tasked with utilizing the graphic API in an optimal way therefore it has a

large impact on the final performance of the application (shader code is the other thing that has a high impact on performance). Since both the traditional rendering pipeline and the ray tracing pipelines use the same graphic APIs (DirectX 12, Vulkan, etc.), this stage is the same for both rendering pipelines (applications that use ray tracing usually use raster as well, and ray tracing APIs are just a subset of the graphic APIs).

Geometry Processing – This stage runs on the GPU and is in charge of most per-vertex and per triangle operations, it can be broken down to the following sub stages: vertex shading, projection, clipping and screen mapping (Figure 2).

Vertex shading or vertex shader is a programmable stage that is traditionally used to compute the position of the vertex and any other types of data that the programmer would like to set for later stages such as texture coordinates and vertex normal. In most common vertex shaders this is where the vertex position is transformed from model space to projection space by multiplying the input position by the MVP Matrix (Model View Projection). The per vertex data that is outputted from this stage is later only interpolated between the vertices of each triangle in order to obtain the correct data per pixel. On modern GPUs all shader stages run on the same HW units called compute units which are highly parallel units inside the GPU. There are two other types of shaders that can be used at this stage: tessellation and geometry shaders, but those are beyond the scope of this work.

After the vertex shader executed the GPU uses the projection matrix provided in order to transform the object into a unit cube's space that represents what the eye sees.

The Clipping stage happens after the vertices have been projected into the camera view space and the unit cube, and it's designed to make sure that only objects that are inside the view frustum are rendered by discarding objects that are outside the view entirely and clipping objects that are partially in view so that the vertex outside the view is replaced by a new one located at the intersection between the view volume and the line between the vertices. At the end of this stage a perspective division is performed that transforms the vertex position to normalized device coordinates (which depends on the graphic API used).

The last stage of geometry processing is Screen mapping, the clipped primitives inside the view frustum are passed to this stage as three-dimensional points. At this stage the x and y coordinates are transformed to screen coordinates whereas the z coordinates are mapped to the range [z1,z2] that depends on the used graphics API (i.e., [-1,+1] for OpenGL, [0,1] for DirectX, etc.). The mapped z coordinates are passed on to the rasterizer stage.

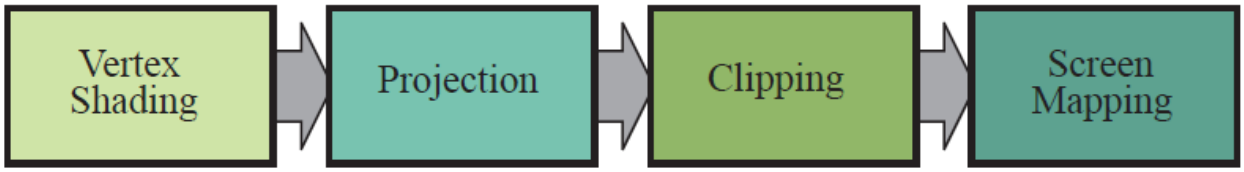

Figure 2 - Geometry Processing. Image taken from [3]

Rasterization – The goal of the rasterization stage (Figure 3) is to take the transformed object from the previous stages and find all the pixels on screen that are inside that object. This process is split into two substages called triangle setup (or primitive assembly) and triangle traversal (shown in Figure 3). Thus rasterization (also called scan conversion) is the process of converting two dimensional vertices in screen space and a depth buffer (and other shading information passed from the vertex shader) into pixels on the screen. The rasterization stage is not programmable as vertex shading and pixel shading, however the way a triangle is checked for overlapping a pixel can be configured by different settings of the GPU pipeline. ([8] describes the different settings that the programmer can configure in the rasterization stage.)

In the Triangle Setup stage differentials, edge equations and other data for the triangle are computed, the computed data is used by the triangle traversal stage or for interpolation of different types of data that is passed from the geometry processing stage (i.e., UV coordinates, normals or any other data that is saved per vertex and is meant for interpolation).

In the Triangle Traversal stage each pixel that has its center covered by a triangle is checked and a fragment is generated for the part of the pixel that overlaps the triangle. That fragment will later be processed in the pixel processing stage. For each fragment interpolated data is generated from the 3 vertices that form that triangle. Finding exactly which pixels are inside a triangle is referred to as triangle traversal and the algorithm is presented more in depth in [McCormack et al.]. All the resulting fragments are sent to the next stage which is pixel processing.

Pixel Processing – In this stage (Figure 3) each pixel (sometimes referred to as fragment) is processed in a pixel shader (a program created by the developer) and the result is sent to the *Merging* stage in order to write the output to a buffer that will be displayed on the screen or used for another rendering stage.

In the *Pixel Shading* stage per pixel computations are performed using the interpolated shading data from previous stages as an input. A shader program that is provided by the developer runs on the compute unit (the same HW unit that runs other shaders) in order to produce one or more output colors that are later on passed to the next stage. This stage is where texturing and lighting calculation is performed.

Each pixel holds its information in the color buffer (or sometimes multiple buffers), the *merging* stage is responsible for combining the color produces in the pixel shader with the color that is currently in the buffer. This is done according to the depth buffer that was produced by the geometry processing stage and is referred to as visibility test or z test. In cases where transparent objects are rendered this stage will also blend colors according to the blend settings that are given to this stage by the programmer. Unlike previous stages that were programmable this stage is fixed but can be controlled by passing some settings in the Application stage (i.e., blend operations, stencil, etc.). As mentioned earlier the results of the merging stage can be used as a framebuffer to be displayed on screen or used as a resource for other rendering passes.

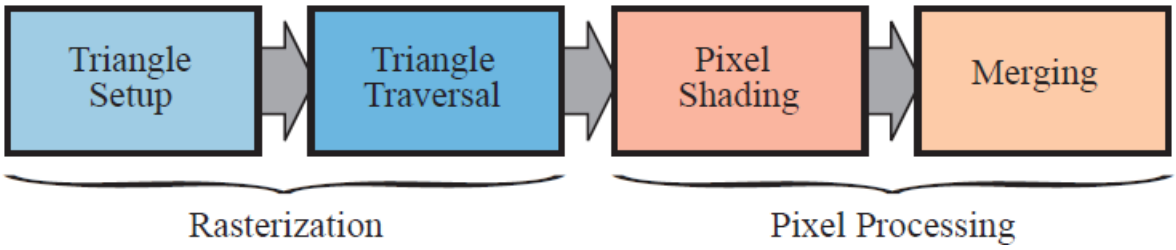

Figure 3 - Rasterization and Pixel Processing. Image taken from [3]

Compute shaders – compute shaders are another way to utilize the parallel nature of the GPU. Compute shaders can be used for both graphics and non-graphics tasks, they can be used to move some tasks that were previously done on the CPU to the GPU in order to improve the performance of the application.

Compute shaders run on the same compute units that the other shader stages use and the main difference between them and the more traditional graphics shaders is that they aren't restricted to working on a 2D buffer that represents the frame buffer, instead the developer specify how to run the compute shader by providing something called a thread group size. The programmer can query the threadGroup ID inside the compute shader to identify which piece of data is currently being computed. One big advantage that compute shaders have is that they operate and access data that is on the GPU so they are more efficient for transferring data between different computations because they do introduce the delay that occurs when sending data between the GPU and CPU. The data can be kept inside the GPU memory for future use. Another big advantage Compute shaders have is that they can utilize an LDS (Local Data Storage) which is a type of GPU cache that can't be used by vertex\pixel shaders. Some of the most common uses of Compute shader are physics calculations, particle systems and operations that prepare data for later rendering stages (i.e., tile classification – which is explained in the reflections chapter of this work).

3.2. Ray Tracing

Ray tracing [9] is a rendering method that is largely based on the physics of light and how it interacts with surfaces. Therefore, it can create results that are much more realistic to real life compared to traditional rendering techniques that use rasterization and the normal graphics pipeline.

Mathematically a ray is defined using the equation:

$$q(t) = o + td$$

Where o is a vector of the ray's origin, d is the normalized ray direction vector and t is the distance along the ray.

Ray tracing can be described using two functions trace() and shade(). trace() is where most of the ray tracing logic lies and it is responsible for finding the closest intersection between the ray and the objects of the scene. When the closest intersection is found trace() will call the shade() function which is in charge of returning the color of the ray.

Often in ray tracing algorithms, in order to find the color of a pixel, we shoot multiple rays and use some form of weighted average to calculate the final color, therefore ray tracing can be thought of as a recursive function and the shade() function could potentially call trace() in order to shoot more rays.

The Rays that are shot from camera position into the scene are called camera rays or eye rays (Figure 4). Other ray types for example are reflection or shadow rays, that are shot from the surface towards some direction.

Given an integer pixel coordinate (x,y), a camera position c and a coordinate frame {r, u , v} (right vector, up vector and a view vector) and screen resolution $w \times h$, a camera ray $q(t) = o + td$ can be computed with the following formula:

$o = c,$

$$s(x,y) = af\left(\frac{2(x+0.5)}{w} - 1\right)r - f\left(\frac{2(y+0.5)}{h}\right)u + v,$$

$$d(x,y) = \frac{s(x,y)}{\|s(x,y)\|}$$

Where the normalized ray direction d is affected by $f = \tan(\emptyset/2)$ with $\emptyset$ being the camera's vertical field of view (the measure of the total area a person or object can see vertically via an optical device, measured in degrees of total visible angle) and $\alpha = w/h$ is the aspect ratio. (The

camera coordinate frame in this setup is left-handed meaning r points to the right, u is the up vector and v points away from the camera towards the image plane.

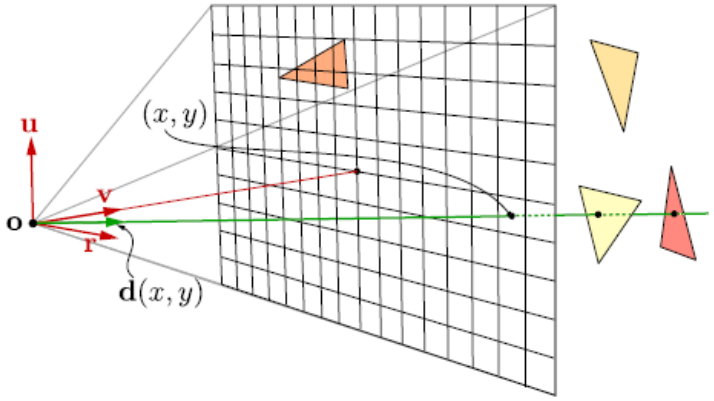

Figure 4 - Ray tracing camera setup. Image taken from [9]

3.3. Shader Types

In recent years Ray Tracing was integrated into real-time rendering APIs such as DirectX12 (DXR) and Vulkan. The addition of Ray Tracing capabilities to those APIs allows developers to create rendering engines that are either solely based on Ray Tracing or take a hybrid approach by mixing Ray Tracing and rasterization.

For example, we can generate a G-buffer[1] of the scene using rasterization and then use shoot rays from the rendered scene in order to generate shadows and\or reflection.

Rays are dispatched in a similar fashion to dispatching compute shaders over a grid of pixels, and there are 5 different types of ray tracing shaders [9], [10] (though a typical use case doesn't have to include all of them):

1. Ray generation shader
2. Closest hit shader
3. Miss shader
4. Any hit shader
5. Intersection shader

The **ray generation shader** is the entry point in the ray tracing pipeline. It is usually called for each pixel on the screen, and can call the function TraceRay(), which is similar to the trace() function discussed earlier. A ray is defined using an origin point, a direction vector and an interval

[1] G-buffer is a part of a common rendering technique called "Deferred shading". In Deferred shading the data required for shading computation is first rendered into an array of buffers called the G-buffer and in a later pass the data is used to compute the scene's lighting.

[tmin, tmax] that specify the part of the ray where intersections are accepted (we use a small epsilon for tmin in order to avoid self-intersections*).

*A self-intersection is when the ray hits the surface that is at the origin of the ray due to accuracy issues thus giving us a false intersection.

A data structure called the ray payload can be defined by the programmer in order to send various data between the different ray tracing shaders.

The TraceRay() function includes the functionality of fast traversal of a spatial acceleration structure (which includes the scene's geometry) and a default intersection test which are both implemented in the driver thus making it transparent to the programmer.

The API also allows defining different ray types and associate each with a different set of shaders. E.g. standard (camera) rays can use one set of shaders while a simpler set of shaders can be used for shadow or reflection rays as for those we can stop when finding the first intersecting geometry.

The **closest hit shader** is executed when the closest intersection from the ray origin along its direction is found. This shader is typically where the user implements shade(). If no intersection is found the **miss shader** is executed. This is useful for returning some color value, e.g., sampling an environment map or returning a static background\sky color.

The **any hit shader** is optional, and as its name suggests, is executed for every intersection along the ray. It is especially useful for rendering transparent objects as the ray can continue when the sample is transparent or otherwise stop. The intersections along the way are not guaranteed to execute in order, thus it is up to the programmer to implement some sort of local sorting inside the shader code in order to get the correct results.

The **intersection shader** is executed every time a ray hits an object stored in the acceleration structure, and is used to implement custom intersection tests. Since the driver implements a default intersection test as part of the API, this shader is optional and is not used in most common use cases.

As mentioned earlier, ray tracing can be a recursive process, therefore both miss shaders and closest hit shaders can also call TraceRay() in order to shoot new rays. All ray tracing shaders can output data to a UAV (Unordered Access View), and all of them, except the intersection shader, can modify the ray payload structure.

Figure 5 shows the ray tracing shaders flow.

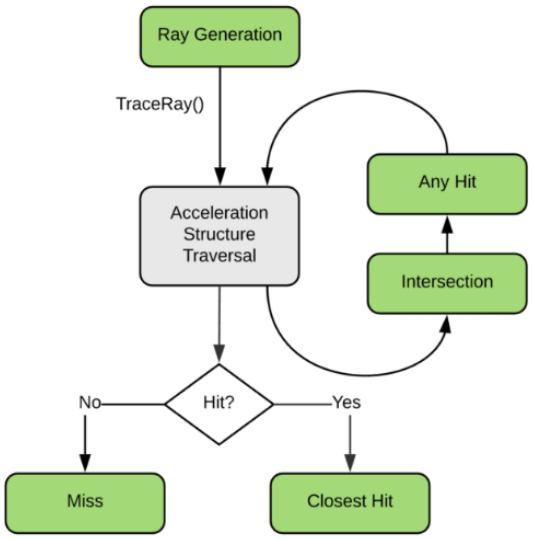

Figure 5 - Ray tracing shaders flow, image taken from [11]

3.4. Acceleration Structures

This work is going to focus mainly on rendering techniques that are implemented in shader programs. However, one of the main reasons that ray tracing rendering techniques have become possible for real time is the added hardware support for acceleration structures.

Ray tracing acceleration structures are data structures that are used for holding the scene objects (geometry information, shading information etc.) and are very effective for performing spatial search in order to compute the intersection of rays with the scene geometry.

At the time of writing, the acceleration structures used by the rendering APIs (Vulkan,DirectX 12, etc.) are opaque to the user and are implemented by the GPU manufacturer in their drivers. The most popular data structure for ray tracing is the *bounding volume hierarchy* (BVH) and is what's currently used by Nvidia, AMD and Intel.
The acceleration structures are implemented in a way that supports efficient construction algorithms in order to make it possible to build and update them in real time. Modern GPUs contain special built-in hardware that performs search queries on those structures in order to find intersections.

The acceleration structures (see figure 6) have two main parts, the *top level acceleration structure* (TLAS) and the *bottom level acceleration structure* (BLAS). The BLAS contains the geometry

used to represent the scene while the TLAS contains a set of instances that each point to a BLAS (Figure 6). Each instance in the TLAS also includes a transformation matrix used to position that instance in the scene and an offset into the shader table which holds material information used to shade a surface when there's an intersection.

It is the programmer's responsibility to use the rendering API to build and update the acceleration structures with the scene data as needed and the way this data is updated is one of the challenges in using ray tracing in real time.

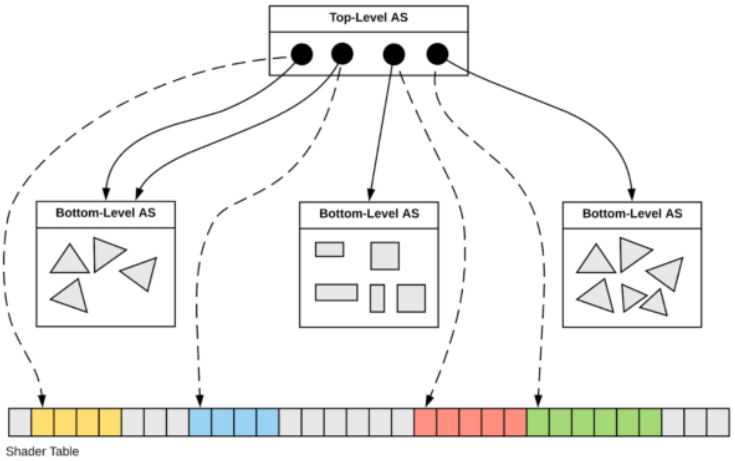

Figure 6 - Ray Tracing Acceleration Structures,On the top, the top acceleration structures which points to the shader table and the bottom acceleration structures, on the bottom the bottom acceleration structures that holds scene geometry, image taken from [11]

3.5. Coherency

Coherency is an important paradigm in software and hardware performance optimization. The idea behind coherency is that we can save effort by reusing results among different parts of a computation. The most expensive operation that is done on hardware are memory operations (both reading from the memory or writing to it) and are a magnitude slower than mathematical operations.

Thus, performance optimizations in general are usually centered around exploiting memory coherency and scheduling computations around memory latency[2].

[2] Memory latency refers to the time it takes the GPU to perform read\write operations to its memory, GPUs are designed to hide memory latency by being able to swap and run different tasks while waiting for the memory operations to finish.

One way of looking at the difference between Rasterization and Ray tracing is that both are algorithms for visibility determination both running a double for-loop but differing by the way it is ordered.

The Rasterization loop can be described as:

```
For (Triangle T : Triangles)
    For (Pixel P : Pixels)
        Determine if P is inside T
```

Whereas the Ray tracing loop can be described as:

```
For (Pixel P : Pixels)
    For (Triangle T : Triangles)
        Determine if ray through P hits T
```

This difference has an impact on how coherency works for each of the algorithms. Since most of the work is done inside the inner loop, and those iterations happen next to each other, this is where the opportunities for data reuse and exploiting memory could be found.

As the inner loop of the rasterization algorithm iterates over pixels of an object's surface, it is very likely that points over a surface will have a highly coherent computation as they might share the same material, textures or even access the same area of these textures.

When computing visibility and shading for a large number of camera pixels, we can make sure that our computation is spatially coherent by, for example, dividing the computations into small tiles on the screen where in each of them there's a high degree of coherency.

In contrast, ray tracing inner loop iterates over the scene primitives performing intersection tests and then performs shading, and there's little coherency that can be exploited when traversing a single list of primitives along a single ray. Therefore, with modern ray tracing algorithms most of the performance optimizations involve with ways of finding coherency in ray visibility queries and subsequent shading computations.

Rays can be cast from any point and in any direction, doing so naively can result in bad coherency and thus an effort must be taken in order to "organize" the way rays are cast in order to achieve good results, this is often referred to as *Ray sorting* [12]. Rays can be reordered and grouped in different ways so that a group of rays will contain rays that are spatially similar and process by the hardware more efficiently. For example, rays can be sorted by their origin or by their direction.

In general, some algorithms will generate a set of rays with a higher degree of coherency than others. If we take shadow algorithms as an example, rays will be highly coherent as for nearby pixels on the screen they will have similar origins and will be directed to the same light. Even with rays that bounce off surface such as reflection rays a high degree of coherency will be retained as two neighboring pixels might hit the same at nearby points with similar normal and then bounce off in the same general direction (especially with mirror reflections). A low degree of coherency will be generated in algorithms where we randomly sample the hemisphere of outgoing directions from a given surface point. For example, when doing ambient occlusion or diffuse global illumination.

When discussing some of the rendering techniques that leverage ray tracing in the next few sections, we will also discuss potential optimizations that can be used to make those algorithms perform better and be viable for real time applications.

4. Denoising

Ray tracing is a computationally expensive operation, therefore for real time application we would normally shoot one or two rays per pixel, sometimes we would even consider only shooting rays from some of the pixels. E.g., when we use ray tracing to calculate soft shadows, we might want to use ray tracing on the edges between shadow and light.

Since we're limiting the number of rays in order to be able to run efficiently and in real time, the amount of data we get is rather limited and will produce a noisy result (Figure 7 – left). In order to overcome this issue and produce a cleaner final image, an image denoising process is necessary. (Figure 7 – right)

Usually, the denoising process will be comprised of both a spatial denoiser and a temporal denoiser.

Figure 7 – On the left: A noisy result of an image with ray traced reflections, on the right: the same image after denoising. Image taken from [13]

4.1. Spatial denoising

A common image processing technique to denoise an image is to use a blur filter. An image is provided as an input and the filter pass computes new values for each pixel using some operation and, in the end, outputs a new image. Since GPUs are highly parallel, they can perform those image processing techniques effectively by utilizing a pixel or a compute shader. As we're discussing real time applications in this work, we will assume that any image processing technique mentioned is done on the GPU.

A new value for each pixel is computed by taking the pixels neighbors into account. A *filter kernel* specifies the exact way that this is done. The number of neighboring pixels that are used to compute the new value for each pixel is called the *kernel diameter*, the kernel diameter can be provided as a parameter that is adjusted to better suit the filtering need of the denoising pass [14].

A common filter to apply to a noisy image is the *Gaussian Filter* which is defined as:

$$G(x) = (\frac{1}{\sigma\sqrt{2\pi}})e^{\frac{-r^2}{2\sigma^2}}$$

where σ is the standard deviation and r is the distance to the pixel's center. A larger σ means a stronger blur.

A Gaussian Filter can be separated into two passes, each is applied along a different axis and has the same result as applying both axes in a single pass. Such a filter is referred to as a *Separable Filter* and it is considered better for performance compared to a *non-separable filter* as it reduces the number of samples per pixel from n^2 to 2n where n is the kernel diameter.

The main drawback of using a Gaussian Filter is that it doesn't preserve edge information and can cause some of the edges to become too soft. Another type of filter that can be used instead and does preserve edge information is the *bilateral filter*. A bilateral filter is also a weighted average of pixels, but unlike the Gaussian filter it also takes into account the variation of intensities in order to preserve edges. The idea behind it is that two pixels that are close to each other spatially are also close to each other in terms of their photometric properties, such as their brightness, color or normal. When the filter is applied it will average pixels that are closer together but ignore pixels that are closer together spatially but are different in other properties (mentioned above), thus keeping edge information intact.

The edge preservation property of bilateral filters makes them a very popular choice for denoising and are often used for upscaling as well. In ray traced rendering, it is often common to do the ray

tracing pass in a lower resolution and then use an upscaling technique to adjust the results back to the original resolution.

4.2. Temporal denoising

Another important strategy for denoising is to accumulate and use results from previous frames in a similar fashion to the way Temporal Anti-Aliasing works [15]. Using data from a previous frame is called *reverse projection* and involves keeping a *velocity buffer* that encodes the change in position for each pixel from the previous frame to the current one. By keeping this information, we can reproject the data from the previous frame to the current frame and use it to enhance the overall quality of the frame. A common practice is to have some kind of small random variation in the way we sample or trace rays between frames so that when two frames combine, we benefit from having extra information. It is important to make sure that the pixel we reproject from a previous frame is the same pixel we have in the current frame as sometimes a pixel from a previous frame can be occluded in the next frame and vice versa.

In order to make sure that a pixel information from a previous frame is valid we also compare the pixel color value between frames and avoid using a pixel from a previous frame if the difference is too big. The downside to using temporal techniques is that they can sometimes introduce lag between frames or visual artifacts called *ghosting* special care must be taken in order to minimize such artifacts.

Spatio-Temporal Variance Guided Filter (SVGF) [16] is a reconstruction algorithm that creates a temporally stable sequence of images from a noisy input image. It is designed to be used for 1 path per pixel global illumination techniques and it uses temporal accumulation to increase the effective sample count and spatiotemporal luminance variance estimation to drive a hierarchical image-space wavelet filter which helps distinguishing between noise and detail at multiple scales using local luminance variance.

Adaptive Spatio-Temporal Variance Guided Filter (A-SVGF) [17] is an improved technique that aims to reduce lag and ghosting artifacts introduced by SVGF during fast motion.

4.3. DLSS & AI based denoisers

Deep Learning Super Sampling [18][19] is a feature introduced by Nvidia on their latest GPU series. It uses a trained neural network to perform upscaling to higher resolutions [20]. This feature is especially useful with ray tracing as it can be really beneficial for performance to perform a ray tracing pass in a lower resolution and then upscale it to higher resolutions. DLSS usually provides better results than upscaling techniques that are not based on deep learning. The biggest limitation is that it's only available on Nvidia hardware. AMD and Intel recently introduced their own

upscaling solutions called Fidelity[21] and XeSS[22] Respectively. However, the current version Fidelity isn't based on deep learning and at the time of writing is considered inferior to DLSS but has the advantage of not being restricted to AMD hardware.

Besides AI based super sampling, there's a lot of new research on the topic of deep learning based denoisers, such as [23]. Since denoising techniques and super sampling are important to achieve high quality ray tracing results in real time this area is widely researched and will continue to be so in the future.

5. Shadows

Shadows are an important part in video games and simulations, shadows contribute to realism and help us comprehend spatial relationships between objects and where they are located inside the scene. Furthermore, shadows can provide the obsever with other information about the scene such as the complexity of shadow occluders and receivers and light characteristics such as the shape and size of the light source and its intensity. Shadows can also be used as artistic means and many movies (and sometimes video games) exploit shadows to illustrate the presence of a person or an object without revealing its actual appearance.

A shadow is defined as a dark area where light from a light source is blocked by an opaque object.

Shadows are created when a surface blocks the light's path. The shadows caused by an ideal point light source would be sharp but in the real-world shadows have blurry edges, this is called the penumbra. A penumbra arises because real world light sources cover some area and so produce light rays that gaze the edges of an object at different angles.

5.1. Shadow Maps

Shadow mapping is the most commonly used shadow technique and it was first introduced in [24]. Its main principle is that the light sees all the lit surfaces of the scene, therefore every unseen surface must lay in shadow. In order to determine which surfaces are visible to the light the algorithm first creates an image of the scene from the lights position, where each pixel in this image holds the depth of the visible surface (its distance from the light), this image is known as the shadow map. The creation of such image uses the same mechanism used for rendering depth maps in order to determine a surface's visibility during standard rendering; therefore, it's supported by graphics hardware.

After we have obtained the so called shadow map the second step of the algorithm would be to render the scene from the actual viewpoint (Figure 8). In the fragment shader stage each fragment position p is transformed into light clip space so that $p^{LC} = (p_x^{LC}, p_y^{LC}, p_z^{LC})$, where (p_x^{LC}, p_y^{LC}) are the coordinates in the shadow map to where p would project when viewed from the light's position, and p_z^{LC} is the distance between the fragment and the light.

In order to determine if the fragment is seen from the light (and therefore is lit), all we need to do is test if p_z^{LC} is smaller the value stored in the shadow map at (p_x^{LC}, p_y^{LC}), otherwise the fragment must be hidden from the light by some other surface that's closer to the light source and therefore lie in shadow.

We should note the fact that the depth map is not accessed directly with light clip-space position p^{LC} but with its according shadow texture coordinates p^{s} which can be derived according to the same matrix Mt we used earlier to calculate the texture coordinates in the range of $[0,1]^2$.

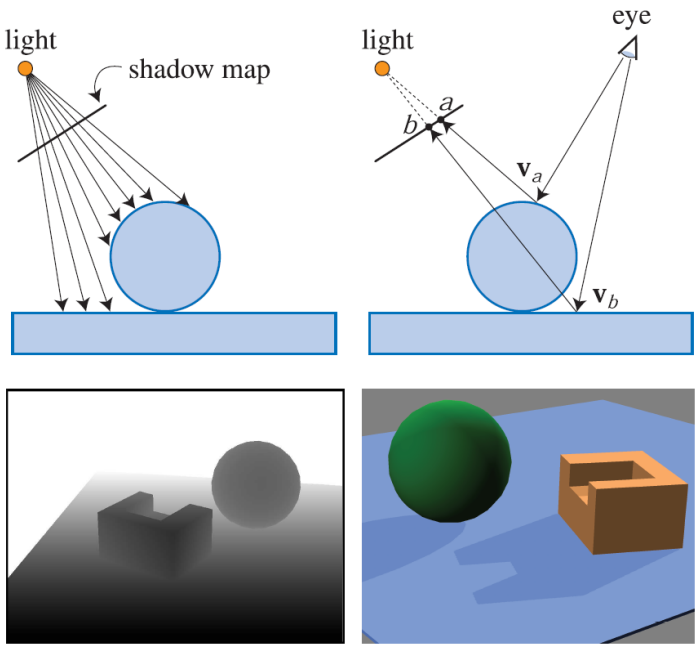

Figure 8 - Shadow mapping technique. Image taken from [3]. **On the top left, a shadow map is formed by storing the depths to the surface in view. On the top right, the eye is shown looking at the two locations, the sphere is seen at point Va, and this point is found to be located at texel a on the shadow map. The depth stored there is not (much) less than point Va is from the light, so the point is illuminated. The rectangle hit at point Vb is (much) farther away from the light than the depth stored at texel b, and so is in shadow. On the bottom left is the view of a scene from the light's perspective, with white being farther away. On the bottom right is the scene rendered with the shadow map.**

The biggest advantage in using the shadow mapping technique is the fact that it works great with almost any input as long as depth values can be produced and since both steps of the algorithm involves standard rasterization gives the technique a huge potential for graphics hardware acceleration.

Although this technique is very powerful and very common in modern rendering and game engine, it does have a few problems that should be addressed when used including imprecisions due to the depth test step, aliasing artifacts caused from the pixel representation of the shadow maps, and last the need for special treatment for omnidirectional light sources.

5.1.1 Disadvantages of shadow mapping

When implementing the shadow mapping technique there are two main issues that must be addressed.

Depth bias

In order to test if a point is farther away than the reference in the shadow map, some tolerance threshold is required for the comparison. Otherwise, due to the limited shadow map resolution and other numerical issues, the shadow will include unwanted artifacts that are commonly referred to as z-fighting or shadow acne (Figure 9).

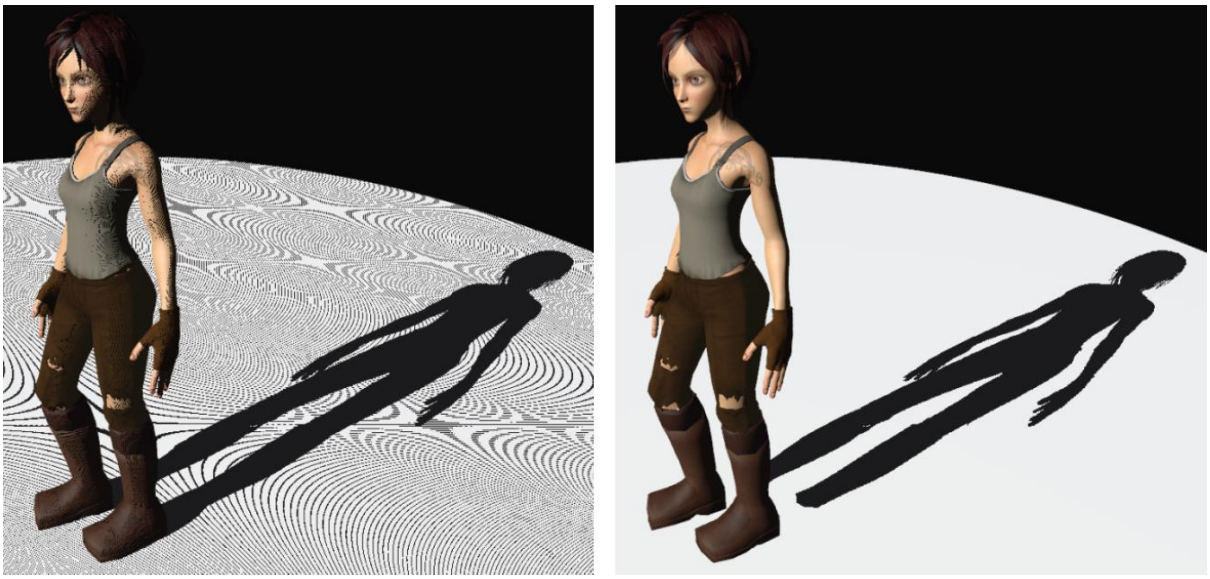

Figure 9 – On the left: shadow acne artifacts caused by the lack of shadow bias, on the right: the shadow bias is too high which causes the shoe to not cast contact shadows. Image taken from [3]

The explanation to this issue is that if the shadow map had infinite resolution, the shadow testing would be a matter of checking if the point is represented in the shadow map, however with a discrete shadow map resolution, sampled points from the eye position are compared to an image consisting of pixels, each of these pixel's value is defined by the world sample corresponding to that pixel's center. When querying a view sample, it usually will not project exactly to the same location as was sampled in the shadow map and therefore will be compared to the average between nearest neighbors. This leads to problems when the view sample is farther from the source than the corresponding value in the shadow map because unwanted shadows can occur.

In order to deal with the shadow acne issue a threshold needs to be added in the form of a depth bias that will offset the light samples slightly from the light source.

Although this might sound simple, it is in fact more problematic than it seems, as the greater the slope of the surface is the bigger the change in depth value between adjacent samples, which means that a larger value of depth bias is required in order to avoid the surface incorrectly shadowing itself.

Furthermore, if the depth bias is too large it might lead to light leaking at contact shadows, causing the shadow to look disconnected from the caster, a problem usually referred to as Peter Panning (Figure 10).

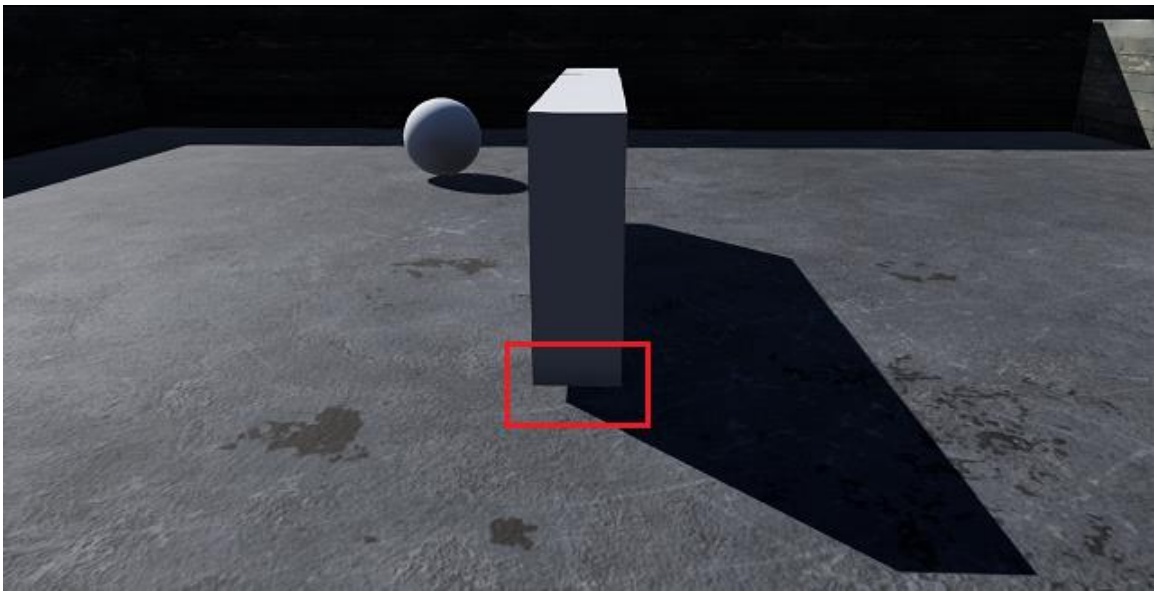

Figure 10 - Peter panning artifact. Image taken from [25]

The standard approach is to use two parameters for the bias, one constant and the other one depends on the slope of the triangle as seen from the light source.

Shadow map aliasing

A major drawback of using an image-based scene representation for calculating the shadows is that the shadow map texture is limited in resolution and therefore causes aliasing artifacts (Figure 11). Aliasing in the shadow map means that the shadow created using the shadow map will reflect the discretization of the texture in the form of pixels. Consequently, the edges of the shadow will contain visible stair stepping artifacts.

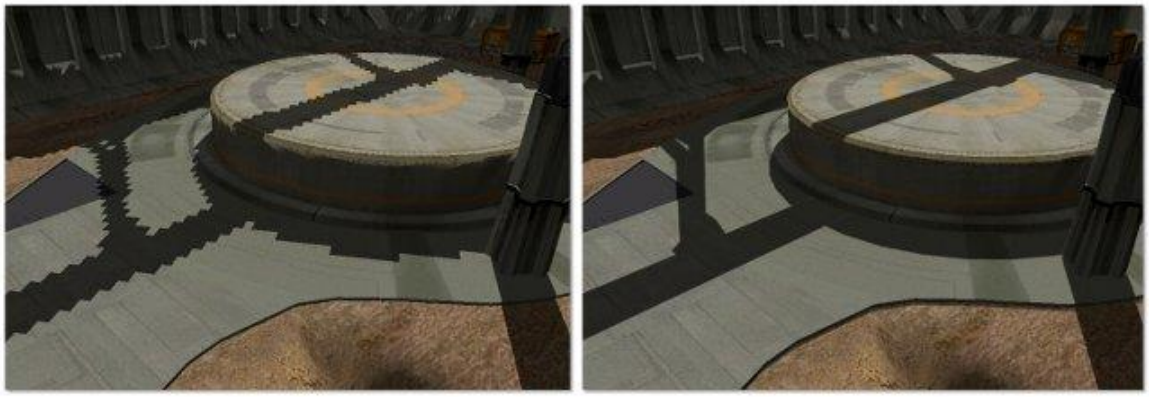

Figure 11 - Aliasing artifacts; Left – shadows without anti-aliasing; right – shadow with anti-aliasing. Image taken from [26]

The reason for this is that several view samples on a receiver object might be projected onto the same texel (short for texture element) in the shadow map, hence will receive the same shadow response. As soon as the object projects onto the next texel the shadow response might change drastically causing the stair stepping artifacts mentioned above.

The best approach to dealing with these artifacts is to increase the shadow map resolution and usually in modern games shadow map resolution exceeds the window size; 4096 and 8192 are common choices. By using such large textures, the chance of two view samples to fall into the same texel is reduced.

Another solution is to create a shadow map per object and to adapt the shadow map resolution. Although this solution is useful for creating high quality shadows, it is considered more complicated, as it requires specialized scene structures and separated rendering passes.

Another common technique for dealing with aliasing in shadow maps is called "Percentage Closer Filtering" or in short PCF. The basic principle of PCF is that instead of sampling one texel in the shadow map in order to determine if a pixel is in shadow, we also sample the texel's neighbors and return a float in the range [0, 1] that reflects the weighted average of these samples. We can use this result as the shadow factor for the pixel and by that makes the shadow borders much smoother.

This technique is so common that a hardware implementation already exists in modern GPUs and is also a base for a soft shadows technique I will describe in 5.1.4.

5.1.2 Soft Shadow Techniques

As mentioned earlier, soft shadows are important to achieve high realism in scenes (Figure 12) and are the result of using an area light (which has a clear size and shape as opposed to a point light that has no size).

Figure 12 – On the left: hard shadows, on the right: soft shadows. Image taken from [27]

In this section I will give a brief introduction to how soft shadows can be created and describe briefly some of the most common techniques used today.

Shadow mapping is the most used technique today to create shadows. Unfortunately, the standard shadow mapping algorithm is unable to generate shadows with penumbra as it cannot handle area lights, therefore shadow mapping needs to be extended in order to support the creation of soft shadows.

5.1.3 Fixed sized penumbra filtering techniques

The following techniques can be used to blur the shadow in order to create a soft-shadow-like appearance. The results these techniques can achieve are pretty good but the drawback is that the resulting penumbra is fixed sized thus does not reflect the occluding object's distance from the shadow receiving object.

- PCF (Percentage Closer Filtering) – We mentioned PCF earlier as a technique to handle shadow map aliasing. This technique is also commonly used to create a fixed sized penumbra soft shadows.
- VSM (Variance Shadow Maps) – Introduced by Donnelly and Lauritzen [28], this technique uses statistics to facilitate precomputation of shadow-map filtering. The basic idea of this technique is that instead of the standard depth shadow map we write both the depth and depth squared to a two component variance shadow map. Then by filtering over a region of the shadow map we can compute the mean and variance of the distribution for that region. We can then use the mean and variance to apply Chebyshev's inequality in order to compute an upper bound on the probability that the currently shaded surface is shadowed. The benefit of this method is that it allows a linear filter to be used to filter the shadow map therefore improving computation time when using a large filter kernel (which gives softer shadows).

- CSM (Convolution Shadow Maps) – Introduced in [29], is the first approach that allows filter precomputation based on a linear signal theory framework by using Fourier expansion to derive suitable basis functions for the approximations of the shadow comparison function.
- ESM (Exponential Shadow Maps) – Introduced in [30] and [31], this technique is based on the CSM technique but instead of using Fourier expansion it uses simple exponential which voids much of the storage requirements that the CSM method has.

5.1.4 Variable sized penumbra filtering techniques

The following techniques are different than the ones mentioned before, as they take into account the distance between the object and therefore the penumbra width is of variable size.

- PCSS (Percentage Closer Soft Shadows) – one of the most common techniques for image based soft shadows, this technique is based on the PCF technique and will be discussed in details later.
- PCSS + VSM\CSM – a combination of the PCSS technique with either VSM or CSM techniques mentioned in the previous section.
- Backprojection – refers to a group of algorithms initially introduced in [32], the basic idea in these techniques is to obtain a shadow map from the light source's center and reconstruct potential occluders in world space from it. These occluders are then back-projected from the currently considered receiver point **p** onto the light's plane to estimate the visible fraction of the light area. These techniques mainly differ in the type of occluder approximation that is employed, the way the computation is organized and how the occluded light area is derived. These choices influence the performance of the technique, its generality, robustness and visual quality.

PCSS

The basic idea of the PCSS technique [33] is to calculate the penumbra width and use it when filtering the shadow map to achieve soft shadows. There are 3 parts in this technique:

- Blocker search – first we need to calculate the shadow region R_S that contains samples of the relevant occluders. The shadow region is defined by the intersection of the light-receiver-point pyramid (where the light source is the base and the point **p** is the apex) with the shadow-map's near plane, the region is then searched for blockers which are simply identified by performing a standard shadow map test, meaning all shadow map entries that are closer to the light source than **p** are considered as blockers. (Figure 13 shows an illustration of the blocker search, the

light source in yellow is the base of the pyramid while the red point on the surface represents the apex, a shadow blocker is represented in green)

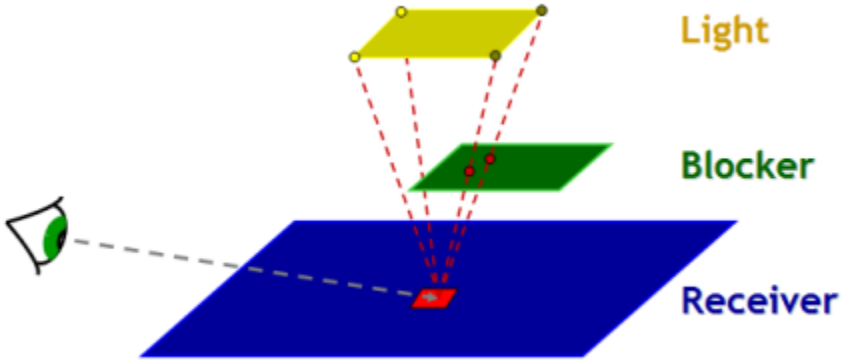

Figure 13 - Illustration of the blocker search step. Image taken from [33]

- Penumbra-width estimation – in the second part of the algorithm we need to calculate the penumbra width, we make the simplifying assumption that there is only one occluder that is furthermore planar and parallel to the area light's plane. For this plane, we use the average depth of all the blockers that we found in the blocker search phase, we mark it with Z_{avg}. We also make the assumption that point **p** belongs to a planar receiver that is also parallel to the planar occluder, the penumbra width is then calculated using the formula:

  $$W_{penumbra} = \frac{p_z - z_{avg}}{z_{avg}} w_{light}$$

  Where p_z is the z coordinate of point p, z_{avg} is the average depth of all the blockers found in the blocker search phase and w_{light} is width of the light.

- Filtering – now that we have the penumbra width, we can use it to derive a suitable PCF window by projecting the penumbra width onto the shadow-map's near plane using the equation:

  $$w_f = \frac{z_{near}}{p_z} w_{penumbra}$$

  Where z_{near} is the near plane and p_z is the z coordinate of point p.

  After that the shadow map can be queried and filtered according to the window size.

5.2. Screen Space Shadows

Screen space shadows is a technique that aims to augment shadows map in areas of small-scale details. Since shadow maps usually aims to capture a large portion of the scene inside the shadow map, areas with small scale details will suffer from bad quality shadows and aliasing. One solution would be to increase the overall shadow map resolution, but this approach will have high

performance and memory impact. Another solution would be to increase the shadow map resolution during key moments such as character camera close ups, but this approach involves a hard work of manually tweaking the lights for each scene and isn't very practical.

Screen space shadows aim to improve the quality of the already existing shadow maps with a different approach that doesn't have a big impact on the overall performance and doesn't require a lot of manual tweaking.

Screen space shadows [34], also referred to as *Contact Shadows*, uses depth buffer ray marching in a similar way that this technique is used for Screen Space Reflections. The idea is that we start at the pixel position and then we ray march the depth buffer towards the light source's position. At each step we compare the depth of our ray against the depth stored in the depth buffer. If the ray depth is larger (further away) from the value stored in the depth buffer we assume that the pixel is occluded from the light source and therefore is in shadow.
Since Screen Space Shadows is a screen space technique, it has the same limitations as other screen space techniques and therefore, can't reliably tell if a pixel is in shadow or not. However, this technique is meant to supplement shadow mapping instead of replacing it, and as a technique that aims to enhance small scale areas with higher detailed shadows, its results are quite plausible.

Stefan Seibert et al. [35] present two algorithms that enhance screen space shadows and provide more appealing visual results. *Spatial Depth Buffer Multisampling* is presented as a solution to improve errors caused by using a bad step size when ray marching the depth buffer. For example, when stepping over a really small occluding object. Multisampling the depth buffer the calculated step values have a higher chance to match the correct depth value and therefore, deliver better results. Multipass Occlusion Testing doesn't require increasing the resolution of the depth buffer like in SDBM, it builds on running multiple passes of ray marching with different step sizes. The number of passes can be configured according to available hardware and performance requirements. Every time a different step size is used and a hit occurs a counter is raised. The final hit decision is made by comparing the counter with a predefined threshold.

5.3. Ray-Traced Shadows

As we've seen in the previous section, shadow mapping algorithms and screen space shadows can provide a good approximation of shadows in real time scenes, but comes with several issues such as:

- Perspective aliasing which makes the shadow edges seem jaggy due to low shadow-map resolution or a poor usage of it.
- Artifacts like shadow acne and Peter Panning.

- Soft shadows are hard to accomplish and are not accurate with traditional shadow mapping techniques.
- Don't offer really good support for semitransparent objects.

Although there are ways to mitigate those problems with enhancing the existing shadow map algorithm, those usually come with the additional cost of adding complexity to the way shadows are set up for each scene and require a lot of manual work in order to achieve the desired result for each scene.

Ray tracing can provide a more flexible and accurate paradigm for rendering shadows and it can handle some of the complex situations where shadow maps fail and is a much more scalable approach to rendering shadows. In fact, ray tracing is the de-facto way of rendering shadows in offline renderers such that are used for movies. With the recent advances in graphics hardware using ray tracing for real time shadows have become a viable solution.

However, even with the new hardware support care must be taken when designing a ray tracing algorithm for real time and resources must be used wisely in order to ensure good performance cost. A naïve approach might trace several rays per pixel in order to get samples for multiple light sources which can lead to low frame rates.

In this section we will discuss the basic algorithm of ray traced shadows and will discuss how we can make ray tracing work in real time while supporting multiple light sources, both point lights that produce hard shadows and area lights that produce soft shadows [36]. We will also see some examples to how developers used ray traced shadows in their games\game engines.

Due to recent advances in graphics hardware and ray tracing APIs researchers started investigating using ray traced shadow techniques as an alternative to shadow mapping [37]. But instead of performing a full ray tracing pass, rasterization is used for primary rays (computing surface color) and ray tracing for shadow rays, sometimes referred to as a hybrid shadowing technique.[38], [39]

The basic idea of ray traced shadows is pretty straightforward. As with shadow maps we determine if a pixel is in shadow if it's path to the light source is blocked by another object. Unlike shadow mapping we don't need to render and use a discrete texture to do so. First of all, we use the rasterized g-buffer and camera parameters to find out each pixel's world space position. In order to find out the x and y we take the position of the pixel on screen and use the inverse view-projection matrix. To obtain the z position we sample the depth buffer. It's also possible to shoot a primary ray from the camera position in order to find the shadow ray origin point, but that will waste GPU resources and will impact performance. Once we have the ray

origin we can shoot a ray towards the light source, if the light intersects another surface along its way, the pixel is in shadow, otherwise it's in light. (Figure 14)

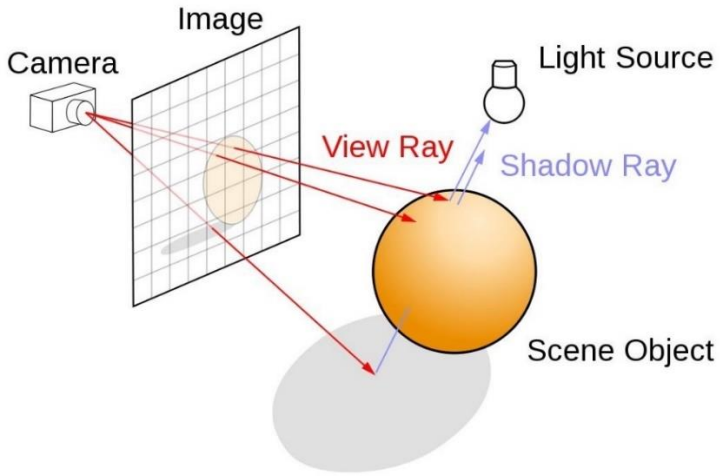

Figure 14 - Ray tracings shadow ray, image taken from [40]

The outgoing radiance $L(P, w_0)$ at a point P in direction w_0 is defined by the rendering equation [41]:

$$L(P, w_0) = L_e(w_0) + \int_{\Omega} f(P, w_i, w_0) L_i(P, w_i)(w_i \cdot \vec{n}_p) dw_i$$

Where $L_e(w_0)$ is the self-emitted radiance, $f(P, w_i, w_0)$ is the BRDF[3], $L_i(P, w_i)$ is the incoming radiance coming from direction w_i and $\vec{n}_p$ is the normalized normal to the surface at P.

Since we're only interested now in direct illumination, we can define the component of L for point light sources as the sum of contributions from individual light sources:

$$L_d(P, w_0) = \sum_l f(P, w_i, w_0)\, L_l(P, w_l) v(P, P_l) \frac{w_l \cdot \vec{n}_p}{\|P - P_l\|^2}$$

Where P_l is the position of light l, $L_l(P, w_l)$ is the radiance emitted from light l in the direction of w_l, where $w_l = \frac{P_l - P}{\|P_l - P\|}$ is the light direction vector, and $v(P, P_l)$ is a visibility function that equals to 1 if the point P_l is visible from P and 0 otherwise. The visibility function can easily be evaluated by shooting a ray from the surface point P towards the light position P_l and checking for

[3] The Bidirectional Reflectance Distribution Function (BRDF)[70] is a function that defines how light is reflected at an opaque surface. BRDF are commonly used in computer graphics to describe a lighting model.

any intersections along the way. Note that care must be taken near the origin of the ray to avoid self-intersections, which can be done by adding a small ε to the ray origin.

In a similar manner the radiance for an area light source a can be defined by the equation:

$$L_d(P, w_0) = \int_{X \in A} f(P, w_x, w_0) L_a(X, w_X) v(P, X) \frac{(w_X \cdot \hat{n}_X)(-w_X \cdot \hat{n}_X)}{\|P - X\|^2} dA$$

Where A is the surface of the light a, $\hat{n}_X$ is the normal of the light surface at point X,

$w_X = (X - P)/\|X - P\|$ is the direction vector from point P to the point X on the light surface, $L_a(X, w_X)$ is the radiance emitted from point X in the direction w_X and $V(P,X)$ is the visibility term that equals to 1 if point X is visible from point P and 0 otherwise.

Using Monte Carlo integration with a set of well distributed samples on the light source surface, the integral can be evaluated to:

$$L_d(P, w_0) \approx \frac{1}{|S|} \sum_{X \in S} f(P, w_x, w_0) L_a(X, w_X) v(P, X) \frac{(w_X \cdot \hat{n}_X)(-w_X \cdot \hat{n}_X)}{\|P - X\|^2}$$

Where S is a group of samples taken on the light surface and $|S|$ is its size. [36]suggests separating the shading and visibility terms and approximating area light source with a centroid C of the light source which yields the following equation:

$$L_d(P, w_0) \approx f(P, w_C, w_0) L_a(X, w_C) \frac{(w_C \cdot \hat{n}_C)(-w_C \cdot \hat{n}_C)}{\|P - C\|^2} \frac{1}{|S|} \sum_{X \in S} v(P, X)$$

These equations allow us to shoot rays from a surface point towards each light source in the scene and sum the results in a *visibility buffer* for each light. The visibility buffer holds the visibility term for each light and can later be used to compute the light contribution for each pixel.

Since the number of lights in a scene can be large, the number of rays that will be required in a naïve implementation can be quite large, especially if the scene contains area lights that will require multiple rays per light in order to sample different points on the light surface and obtain good quality soft shadows. This approach isn't feasible for real time applications which means that care must be taken in order to obtain valuable visibility information with only a small number of rays per pixel (sometimes even as small as 1).

Usually when we discuss hard shadows or contact shadows, we refer to shadows created by a punctual light (a light that is infinitely small such as a point light, directional light or a spotlight)

and when discussing soft shadows, we refer to shadows created by an area light. However [38] presents a technique that allows hard shadows that transition into soft shadows as the distance between the receiving surface and the occluding surface increases. Their approach combines shadow mapping with a PCSS filter and ray traced contact shadows by interpolating (blending) between the two using the distance between from the receiving surface to the occluding object and the desired percentage of soft shadows as the ratio for interpolation. (Figure 15)

*The desired percentage can be configured by an artist\level designer globally or per level or part of a level.

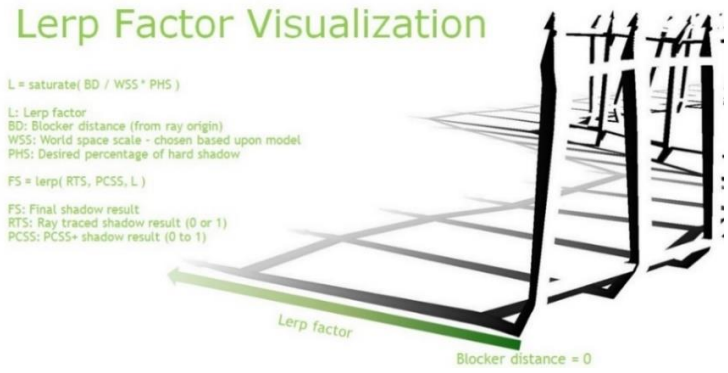

Figure 15 - Hybrid ray traced shadows. The shadow on the left side of the image is softer, meaning a higher percentage of PCSS shadow is used, as we get closer to the occluding object, the shadow becomes a hard shadow, meaning a higher percentage of ray traced shadow is used. Image taken from [38]

The results are a combination of soft shadows and accurate contact shadows without aliasing artifacts or peter panning. (Figure 16)

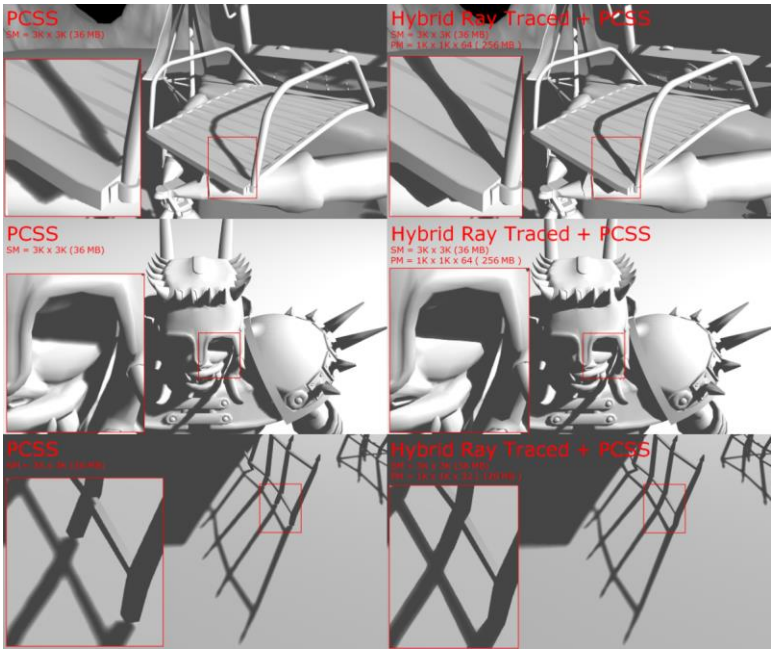

Figure 16 - Hybrid ray raced shadows results. On the right: results of standard shadow maps with PCSS filter, on the left: Hybrid Ray Traced shadows + PCSS shadow maps. Image taken from [38]

[36] describes an adaptive sampling algorithm for hard and soft shadows which helps achieving real time performance and keeping the visual benefits of using ray traced shadows over standard shadow mapping. The algorithm that is described includes the following parts:

- Temporal Reprojection: Temporal Reprojection is used in order to effectively increase the sample count per pixel.
  As we mention throughout this work, temporal reprojection, a standard tool that is used in many real time rendering techniques, so it's often already implemented as part of the rendering engine and is used as part of application rasterization pipeline. By using this method, we can accumulate visibility values for visible surfaces over time that can be used for estimating visibility variation to derive the required sample count but also for determining the kernel size that is used for filtering. The author suggests keeping visibility results from three previous frames which will be used along with the current frame.
  Reverse reprojection [36] is used in order to handle camera movement and bring the 3 previous frames to the same camera space as the current frame. As with other techniques that use temporal reprojection, care must be taken to discard pixels that provide false information (such as disoccluded pixels or newly revealed surfaces, for example when a new object enters the scene), this can be done by tracking the pixels using velocity vectors as is commonly done with temporal anti-aliasing.

- Penumbra regions identification: The number of samples at each point required to compute the shadow factor depends heavily on the light size, its distance to the shaded point on the surface and the complexity of any occluding geometry. Soft shadows coming from area lights require a high number of samples per pixel in the penumbra (soft) regions of the shadow. In order to identify those regions a temporal visibility variance measure is computed from the 4 cached frames (the 3 previous frames from the temporal reprojection and the current frame)

  $\Delta V_t(x) = \max\big(v_{t-1}(x), \dots, v_{t-4}(x)\big) - \min\,(v_{t-1}(x), \dots, v_{t-4}(x))$

  Where $v_{t-1}(x), \dots, v_{t-4}(x)$ are the cached visibility values from previous frames that are cached in a single texture per light.

  The variation will be zero for a fully lit or fully shadows area and will usually be great than zero in penumbra areas (the fact that each frame uses a different sample set that only repeats after 4 frames helps to achieve good variation results).

  A spatial filter is also used on the results in order to make them more stable by eliminating abrupt changes in variation values and thus helps avoid flickering.

  $\widetilde{\Delta V_t(x)} = M_{5x5}(\Delta V_t) * T_{13x13}$

  Where M_{5x5} is a maximum filter with a 5×5 kernel and $T_{13\times13}$ is a low pass filter with a 13×13 kernel.

  In the final step the variation value $\widetilde{\Delta V_t(x)}$ is combined with a temporally filtered value from the four previous frames using a box filter:

  $\overline{\Delta v_t} = \frac{1}{2}(\widetilde{\Delta V_t} + \frac{1}{4}(\Delta V_{t-1} + \Delta V_{t-2} + \Delta V_{t-3} + \Delta V_{t-4}))$

  According to the author this combination provided good results and allowed the variation value to propagate over large regions while also not missing small regions with large variations.

- Choosing the number of samples: the number of samples (rays shot) that are going to be used for each pixel is based on the number of samples used in the previous frame and the current variance computed in the previous step we described. A threshold δ is used on the variation measure $\overline{\Delta v_t}$ to determine whether to increase or decrease the sampling density for the current pixel. The number of samples per pixel also has a hard limit for keeping good performance and was set to 5 for standard settings and 8 for high settings (settings that provide optimal results depending on the used hardware).

- Sampling mask: The algorithm described so far computes a zero sample count in areas with no temporal or spatial variation, which is mostly in areas that are fully lit or fully

shadowed. For these areas computation could be skipped and values from a previous frame could be used. However, this could lead to errors being accumulated over time. In order to avoid those errors a sample mask is used to ensure that those areas are sampled for at least one fourth of the pixels. This is done by enforcing sampling for a block of $n_b \times n_b$ pixels on the screen ($n_b = 8$ was chosen as it gave the best results). A mask of size 4×4 is repeated over the screen which corresponds to the location of the block. If the entry is equal to the current frame's number modulo four, the pixels in that block with zero sample count are sampled with one ray per pixel per light source. This technique ensures that every 4 frames each pixel is evaluated. (See Figure 17)

- Visibility values computation: In the final step of the algorithm two filters are used on the visibility values (as opposed to filter the visibility variation measure as mentioned earlier). A temporal filter which makes use of the results of previous frames and a spatial filter which applies a low pass filter over the visibility values to remove any remaining noise.

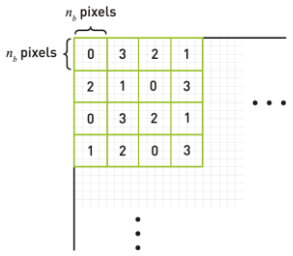

Figure 17 – an example of the sampling mask matrix. In each sequence of four consecutive frames, the shadow rays are enforced even for pixels with low visibility variation. Image taken from [36] **under the license** [42]

The author also mentioned as part of the implementation details that a distance-based light culling system was used in order to cull distant and low intensity lights. For each light a range is calculated – that is the distance at which the intensity of a light becomes negligible is due to its attenuation function. Before evaluation visibility for a light a comparison is made between the distance to the light and its range and a zero value is stored for non-contributing lights. Since typical attenuation functions (such as inverse of squared distance) never actually reach zero, those functions needed to be modified by implementing a linear drop-off below a certain threshold. That made sure that the attenuation can get to zero and lights are efficiently culled and also helped with preventing popping when a light starts contributing again.

* A popping is an artifact that is caused when a light suddenly starts affecting and cause a big change in values between two consecutive frames.

Since the adaptive sampling algorithm results in more samples in penumbras, in some cases where the penumbra area covers a large portion of the screen a big performance decrease can happen. In order to mitigate those cases a limit was put on the total sample count. The sum of the

variation measured $\overline{\Delta v}$ over the whole image is computed, if at certain frame the sum rises above a threshold, the number of samples that can be used for each pixel are progressively limited.

The author evaluated the technique by comparing the results to a reference shadow-mapping implementation. They used 3 different test scenes, each with a 20 seconds camera animation. Two of the scenes had similar complexity and the third one with much larger area lights. The shadow map resolution in the reference implementation was 2048×2048 and the screen resolution for all tests was 1920×1080. They provided the following table that shows the average computation time (in milliseconds) for each of the scenes with different combination of lights, algorithm (Hard shadow maps, hard ray traced shadows and soft ray traced shadows with two quality settings). The measurements were taken on an Nvidia RTX 2080TI GPU (Figure 18).

The results are interesting because they show that when comparing hard shadows between shadow maps and ray tracing in some cases RT hard shadows were actually faster. The author explains that this is possibly due to those scenes having a large number of triangles which slows down the rasterization pipeline used for shadow mapping more than with the RT Cores (the hardware used for ray tracing intersections). There is no direct comparison between soft shadow mapping techniques with soft RT techniques but the results show that computing soft shadows costs about 2-3 times more than hard shadows, which means area lights producing soft shadows should be used with care in any real time application.

	281k triangles		376k triangles		1.4M triangles	
	1 light	4 lights	1 light	4 lights	1 light	4 lights
SM hard	0.7	2.6	2.3	9.1	0.9	5.9
RT hard	1.4	4.8	1.3	3.4	1.6	3.7
RT soft SQ	3.2	13.5	2.7	8.3	4.7	11.0
RT soft HQ	3.5	19.9	2.9	12.0	6.5	16.2

Figure 18 - Results of the shadow algorithm presented above. Image taken from [36] **under the license** [42]

5.3.1 Ray traced contact shadows in "Control"

"Remedy Entertainment" the developers behind the video game "Control" used ray tracing to augment their existing shadows pipeline with ray traced contact shadows. Since shadow maps are

a good and fast solution for large scale scenes but suffer from artifacts caused by resolution issues and shadow map bias whereas ray traced shadows excel with shorter rays and finer details, they would combine the two. Therefore, they decided on the following approach: Shadow maps are rendered for all the shadow casting lights in the scene, then only a few of the most important lights are selected for ray traced contact shadows. The visibility buffer is then denoised and both shadow maps and the denoised ray traced shadows are used for lighting.

Control's rendering engine uses a frustum volume to cull lights and lighting computation is deferred (Using a deferred rendering system). In the main lighting pass the point lights and spotlights with maximum intensity are recorded per pixel in a texture (point lights are stored in one channel and spot lights in another). Those lights are ignored in the main lighting pass and then used for doing the ray traced contact shadows. In a later pass those lights are added to the lighting calculation along with visibility information from the shadow maps and ray traced contact shadows.

In order to have soft contact shadows, during the ray tracing pass the ray direction is jittered with an offset that is within the light's radius. Both point lights and spotlights are treated as spherical lights which will result in a soft penumbra at a cost of some noise that will be denoised before the contact shadows results could be used for lighting (Figure 19).

Figure 19 – On the left: A screenshot from the game with shadow maps, on the right: The same screenshot but with ray traced contact shadows, see how the leg of the chair seems floating in the air in the left bottom image. Image taken from [43] **under the license** [42]

Contact shadows in Control are only computed for opaque surfaces within the camera frustum and in order to keep performance reasonable only the two most significant lights are used with a limit on the ray length and a single ray per pixel. The author shows timing measurements for two different frames (Figure 20) with performance measurements in Table1

Figure 20 - The two frames that were used for performance measurements. Image taken from [43] **under the license** [42]

	Frame 1	Frame 2
Pass	Time (ms)	Time (ms)
Acceleration structure building*	0.6	1.0
Reflection ray tracing	1.0	1.4
Reflection shading	1.4	1.1
Reflection denoising	0.8	0.8
Transparent reflection ray tracing	0.8	0.3
Transparent reflection shading	0.7	0.1
Indirect diffuse ray tracing	0.8	0.7
Indirect diffuse shading	0.8	0.6
Indirect diffuse denoising	1.1	1.0
Contact shadow ray tracing	0.8	0.4
Contact shadow denoising	0.7	0.7
Total cost for reflections	8.9	7.1

Table1 - Ray tracing effect performance measurements in "Control". Taken from [43] **under the license** [42]

Measurements were taken at a 2560 X 1440 resolution with an Nvidia RTX 3090 GPU.

*Note that the acceleration structures are used for all of the ray tracing effects used in the game (including reflections that we will discuss later) therefore, the cost of maintaining these structures is for all ray tracing effects used in the game and as we'll see later relatively small compared to the total cost of all ray tracing effects.

6. Reflections

When talking about Reflections in computer graphics, we refer to a group of techniques that allows us to simulate reflective surfaces, like mirrors, puddles or other shiny objects that reflect light. In order to achieve accurate reflections a ray needs to be shot from the camera toward the reflective surface and then bounced off of it and followed until no surface or a non-reflective surface is found. Since shooting such a ray per pixel and calculating multiple bounces off of surfaces in our scene (i.e., surfaces that shows reflections of other reflective surfaces) is expensive and not a feasible solution for real time rendering graphics, researchers and engineers have come up with cheaper\faster ways to approximate those calculations and achieve reasonable results.

The most common techniques are: Planar Reflections, Screen Space Reflections (SSR) and Environment mapping. Each technique has its pros and cons and usually multiple techniques are used together to leverage each technique's strengths.

The following section will present those techniques and discuss their weaknesses, then I will discuss how the recent improvements in GPUs and the addition of Ray Tracing specific hardware can be used to enhance those techniques with ray tracing in order to improve the quality of the reflections while still rendering the scene in real time.

Reflections are usually divided into the following categories:

- Polished: a very accurate, undisturbed reflection like in a mirror
- Blurry: the surface roughness causes the reflection to look blurry
- Metallic: a reflection on a metallic surface retains the color of the metal
- Glossy: a reflection on a glossy surface will look blurry but will also show highlights from the light source being reflected

However, for simplicity we will usually talk about either mirror or glossy reflections.

6.1. Planar reflections

Planar reflection is one of the simplest techniques that can be used to render reflections. As the name suggests, this technique is used to render reflection on a planar surface (such as a water puddle, mirror, etc.) which one of the drawbacks of this technique as opposed to techniques that can also work on curved surfaces. The technique is mentioned in [44] and discussed and enhanced further in [45].

The general idea is to render the scene again for each planar reflection or to use an environment map (such as a cube map or a probe), the scene is rendered from the point of view of the reflective surface using a transformation matrix and is later used as a texture when rendering the reflective

surface from the main camera's point of view.
Besides the fact that this technique is limited to planar surfaces, the other important drawback is that it is highly expensive. This is due to the fact that each reflective surface mean we have to render the scene again from a different point of view. Thus, our scene is limited to having a very small number of reflective surfaces rendered at the same time in order to maintain a reasonable framerate.

6.2. Screen space reflections (SSR)

Screen Space Reflections was first described by Crytek in their presentation about the video game Crysis 2 [44] and was later discussed in The Devcon and SIGGRAPH talks about the video game "Killzone Shadow Fall" [46], [47] and in the GDC talk by Bart Wronski about the video game "Assassin's Creed Black Flag [48][49].

The main principal of this technique is to use the scene that was already rendered in a previous pass without reflections in order to get color information from reflected objects into reflective surfaces that are in the scene.
For each pixel rendered to the back buffer that has a reflective surface we shoot a ray from the camera to that pixel and then reflect it along the surface normal to get the reflection vector, then we ray march along that vector and sample the depth buffer in order to find the reflected surface. Once we find that surface, we can sample its color to find out the reflected color on the original reflective surface.
In the final pass we blur the result according to the reflective surface roughness in order to make rough surfaces show blurry reflections while keeping sharp reflections on mirror like surfaces.

Figure 21 shows a flow chart of the main components of the algorithm.

The basic version of this technique is composed of 3 general passes:

Reflection Mask Generation\Tile Classification – The heart of this algorithm is the ray marching pass, since it consists of shooting a ray towards the scene and then ray marching the depth buffer to find intersecting objects this pass can be quite expensive in terms of performance.

Therefore, the goal of the initial pass is to determine which pixels in our screen space buffer should be used during the expensive ray marching pass.

Reflection Mask Generation

Reflection Mask Generation is an older technique that was presented in [48], where its idea is to have a fast pass that will generate a mask of all the pixels in the scene that have the potential of having Screen Space Reflection information. The mask was generated by taking a down sampled

half resolution color and depth buffer of the scene and using a sampling pattern in order to sample the buffers and do a conservative ray marching, where a pixel is marked after the first collider the ray hits.

When a ray would hit the first potential collider it will also mark some area around that pixel in order to deal with some flickering artifacts that would occur. Using this mask for the main ray tracing pass saved a lot of calculation when performing the more precise ray tracing.

Tile Classification

Tile Classification is used the same way as Reflection mask. The goal of this pass is to save on computation time when running the ray tracing pass by going over the pixels in the scene color buffer and marking the pixels that could potentially have screen space reflection information and thus are needed to be processed during the ray marching pass.

The color buffer is divided into tiles of predetermined size (for example 32×32 pixels). The tile classifier runs in a compute shader which processes each tile to determine if there are some pixels inside that tile that can potentially hold SSR information.

The compute shader is configured so that each tile will run in a "Threadgroup" so that different pixels on the same tile could run in parallel.
In order to determine if a pixel needs to be marked for the ray marching step, we will look at its roughness value, where a smoother surface (with a low roughness value) has the potential of having SSR information and thus can be marked for the ray marching pass.

In [50] another suggested optimization was to run a conservative ray marching step when doing the classification step to try to quickly find an intersecting object. This cheaper ray marching step would stop at the first intersection and therefore will provide a rough but conservative estimation whether or not the pixel contains valuable SSR information (rays that intersect surfaces that were previously rendered to the scene buffer). Another advantage of using Tile Classification as a pre-pass for the ray marching step is that we can use the result of the tile classification in order to fill an "Argument buffer" which will allow running the ray marching step with "ExecuteIndirect"[51].

By using ExecuteIndirect our rendering engine doesn't need to read any of the tile data that was calculated during the tile classification back to the CPU in order to dispatch new commands to the GPU for the ray marching step, which can be a nice boost in performance.

Screen Space Ray Tracing

The main pass in SSR is sometimes referred to as Ray Marching\Tracing the depth buffer or Screen Space Ray Tracing, not to be confused with techniques that Ray Trace the scene which will be discussed later.

The main principle of this technique is that for every pixel we want to calculate the reflected color, we can shoot a ray along the reflection vector against the scene's depth buffer. We sample the depth buffer along the ray until we find an intersection point, at that point we can sample the framebuffer in order to find the reflection color. When sampling the framebuffer there are two options: we could sample the current frame's framebuffer or we could use the frame buffer from a previous frame with reprojection (slightly shifting the frame to make up for camera movement). Although using the current frame will give the most accurate results, it is usually more common to use the previous frame as this means that some effects such as particles that are usually rendered later in the frame will also be reflected on the surface.

Linear 3D iteration is a simple approach to screen space ray tracing and was presented at [44] for mirror reflections in older games. This method linearly ray marches the sample point along a 3D ray for a limited distance meaning every sample point is at a constant distance from the previous one.

Each 3D sample point is then projected into screen space where it is then checked against the depth buffer and marked as a hit if it is behind the depth buffer value. In order to further improve the precision of the hit, binary search could be used and some heuristics could be used to detect depth discontinuities. The main drawback of this technique is that since we are choosing our sample points in 3D space, in some cases the sample points can be too far apart, thus allowing the algorithm to miss important information. In other cases, multiple 3D sample points might be projected to the same pixel, therefore causing oversampling (which is bad for efficiency).

Morgan McGuire in his paper [52] suggests using a linear 2D iteration technique. This technique addresses the drawbacks of the simpler linear 3D iteration technique by making sure that each sampled pixel is adjacent to the previous one, pixels are sampled at most once and that the ray is clipped against the view frustum. This technique when implemented properly, use GPU resource more efficiently.

This technique is further explained in [48], [49] and was used in the game Assassin's Creed Black Flag.

The biggest drawback of SSR comes from the fact that we are operating in screen space. When looking for a reflection color we ray march the depth buffer to find an intersection. However in

many cases no intersection will be found, either because there is no reflected object and the sky will be reflected instead or there's an object outside the camera view. In order to mitigate the issue most SSR implementation fall back to using *Environment Maps* (sometimes referred to as Image-based Lighting or IBL). [53] explains this technique and how it can be used for computing reflections. The general idea is to prepare a cube map that captures the environment and sample from it according to the reflection direction.

Since environment maps involve rendering the scene from multiple views (6 when a cube map is used) these can be expensive to compute and therefore are usually done offline and loaded into the engine. Because environment maps are rendered from a specific position, they are only accurate for that exact position and therefore can cause artifacts as we get further away from that point. In order to mitigate the issue several environments can be used and the sample can be taken from the closest map or an average between a few maps.

However, the biggest drawback of falling back to environment maps is the fact that those are rendered offline, therefore any dynamic objects including characters or animated objects will not be reflected or will not be reflected correctly. This is where Ray-raced reflections or Hybrid ray-traced reflections, which will be discussed later, can improve on SSR and deliver more accurate reflections.

Blur

As described in the previous step we generate a ray for each pixel by reflecting the ray coming from the camera along the pixel normal vector. This approach will only allow for mirror like reflections and will not work for rough surfaces that produce glossy, blurry reflections. The simplest approach to get glossy reflections with this technique is use another pass that performs a blur filter on the reflection. In order to support both mirror like and glossy reflections and in order to make the glossy reflections look as real as possible the filter kernel (i.e., the amount of blur) will be determined by the pixel roughness value.

As described in [46] in order to support both mirror-like reflections and different levels of surface roughness after calculating the reflection result (stored in a texture) a mip-map chain is generated where each level has a different blur filter therefore producing different levels of blurriness. In the next stage the mip-map chain is sampled and a weighted blend is computed according to the surface roughness value. A low roughness surface will give more weight to samples taken from a higher resolution\less blurry level while a rough surface gives more where to samples taken from a lower resolution\more blurry level. Sharper reflections were not captured well in the gloss mipmaps, therefore, in place, blur using a 12 tap Poisson blur was used in areas with very low glossiness.

Since performing ray tracing in order to compute the reflection buffer can be computationally difficult, a common practice is to render the reflection buffer to a lower resolution (usually half\quarter resolution) and use temporal reprojection, so that the result will be blended with those of the previous frame. In [46] the mentioned approach was to render the reflection results at half resolution by downsampling the image without filtering and instead pick a different pixel of a 2 by 2 quad every frame. This way after 4 frames the algorithm covers the full resolution and the reflection is known for every pixel. However, this results in a jittery and unstable image thus their solution reprojects the final reflection results from the previous frame and blends them with the current reflection results.

The drawbacks of this approach are that it's hard to properly match between surface roughness and the amount of blur needed and the lack of contact hardening, meaning the amount of blur will not be affected by the reflected object's distance from the surface.

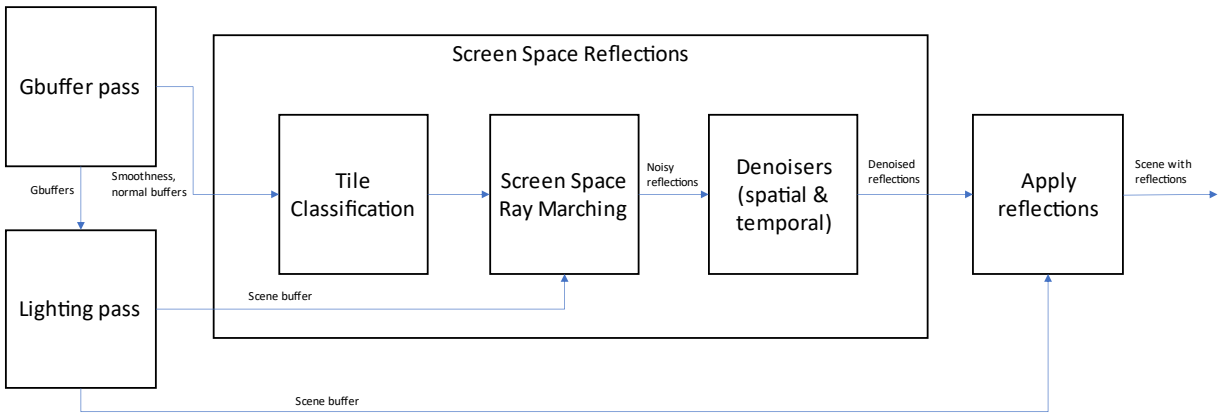

Figure 21 - screen space reflections algorithm flow chart

6.3. Stochastic Screen Space Reflections

Stochastic Screen-Space Reflections presented in [54], [55] is based on the algorithm mentioned above and improves it in order to make glossy reflections more realistic. The main idea is that instead of computing mirror reflections and then blurring the result, importance sampling[4] is used in order to shoot more rays per pixel and doing an average weight according to the BRDF properties. BRDF importance sampling is further explained in [56]. However, since shooting multiple rays per pixel isn't a viable strategy for real time performance the work done by [54], [55] uses a different approach.

Their approach was to start with BRDF importance sampling directions from the surfaces, but only a small number of rays are shot from each pixel, in some settings 1 ray per pixel. Then, instead of returning the reflection color immediately, the intersection point is stored in an intermediate buffer. Then in a 'resolve' pass the previously computed intersection points are reused across neighboring pixels in order to compute the reflected color. The idea is that usually the surface properties of neighboring pixels will be similar and the visibility will be the same. Therefore, they treated neighboring rays as they were shot from the local pixel. While this might not be true in theory, in practice it gave good results. However, care had to be taken, and each ray contribution had to be weighted according to the BRDF.

6.4. Ray Traced Reflections

By using Ray-Tracing for computing reflections, some of the disadvantages of SSR can be avoided.
As we mentioned earlier SSR performs ray tracing in screen-space by using the already rendered scene buffer therefore missing reflection information for objects that are not on screen. In contrast ray traced reflections perform ray tracing in world space, therefore as long as the ray we are casting is long enough, all the objects that are in the scene could be reflected (however we still have to limit the ray to a reasonable length for performance).

At the time of writing there are two common methods for using ray tracing to compute reflections, a fully ray traced reflections pass, or a hybrid approach, that combines ray traced reflections with SSR. Both methods use the same principles, but differ by the fact that fully ray traced reflections cast rays for every pixel on the screen that needs to compute a reflection color whereas the hybrid approach only cast rays when the screen space method fails.

[4] Importance sampling is a Monte Carlo method used to evaluate the properties of a distribution using samples generated from a different distribution then the one we are interested in.[71]

Ray traced reflections build on the same foundations that were presented before when discussing SSR. Since most rendering engines today use deferred rendering the G-buffer can be used to retrieve the ray origins. Tile classification can then be used as a pre-pass to the ray tracing pass in order to mark the pixels where reflection color needs to be computed and calculate the ray direction. As with SSR, in order to support glossy reflection, BRDF importance sampling is used in order to sample in different directions, the number of rays per pixels can be controlled and increased for better quality and more accurate reflections. However, due to the high cost nature of ray tracing, most current implementations rely on 1 ray per pixel and a denoising step in order to mitigate noise.

In the ray tracing pass, positions and ray directions will be retrieved from the previous steps and a ray tracing against the scene in world space will be performed. When an intersection is found the closestHit shader will be executed and the reflection color will be computed. However, unlike SSR when we perform a world space intersection and hit a surface, we will have to compute the color of this surface (in SSR that color was already computed at a previous step). Therefore, this step requires us to have surface parameters and lighting information in order and perform some sort of lighting calculation which can make it more costly.

In some cases, the surface that we hit might also be a reflective surface so multiple bounces will be required. However, for keeping high performance multiple bounces can be avoided and a simplification of the lighting equation can be computed while still maintaining a pleasing visual image.

The area of how to simplify the lighting computation for each hit and the type of data that is passed to those shaders is still an area that is heavily researched and different solutions have been presented, each of which is usually tailored to the product needs and current hardware.

In the hybrid ray tracing reflections approach an SSR solution is used as described above with a fallback to ray tracing, meaning that for every pixel on a reflective surface where the SSR solution fails we will cast a ray in world space and calculate the reflection color in a similar manner to the full ray tracing technique. Results from SSR and ray tracing will be blended together in the blur

stage, the same as was previously done in SSR when using an environment map fallback.

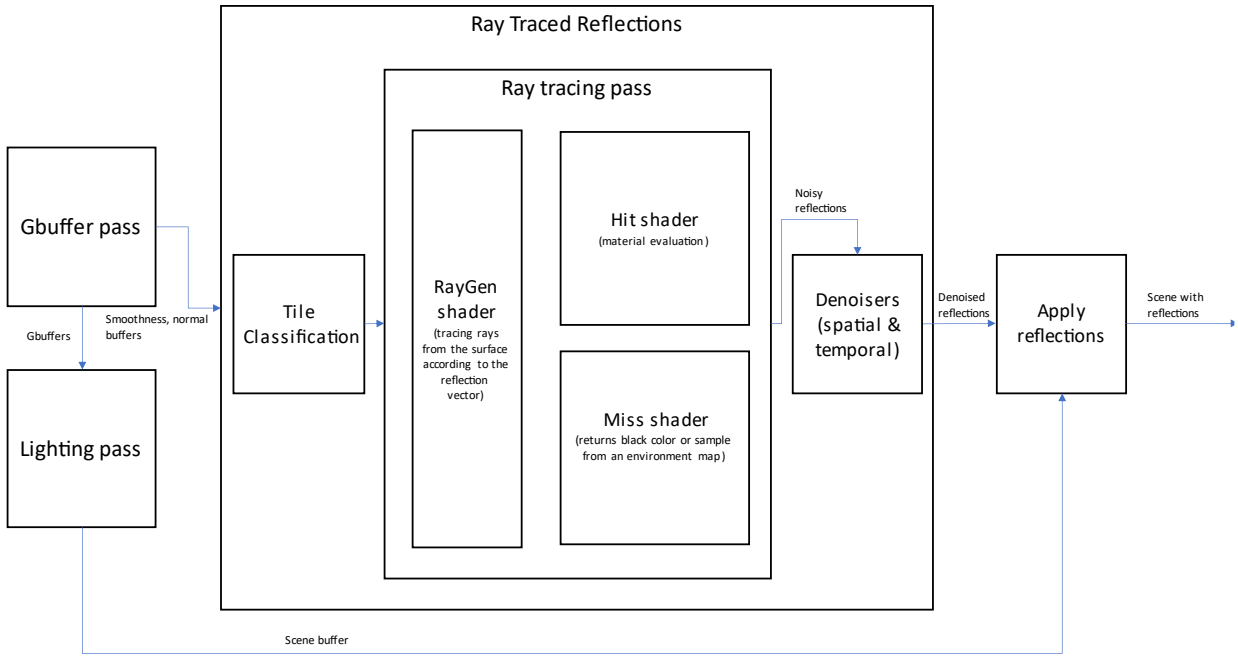

Figure 22 - ray traced reflections algorithm flow chart

6.5. Ray traced reflections in "*Control*"

Control is a video game developed by *Remedy Entertainment* and released in 2019. When it released it was one of the first games to lunch with ray tracing support for multiple rendering techniques. It features a hybrid approach, combining ray tracing with rasterizations (as most games the feature ray tracing currently do). In this section we will describe how *Remedy* implemented ray traced reflections into their game engine.

Remedy's implementation [43] is of a fully ray traced render pass with fallback to precomputed GI data as fallback. The engine uses deferred shading so the ray tracing pass can utilize the G-buffer in order to generate reflection rays based on the view direction and the surface properties. A single ray per pixel is used for computing the reflections (excluding sky pixels). The ray direction for each pixel is determined by importance sampling the GGX distribution of the surface parameters. The ray length is calculated according to the surface roughness where higher roughness leads to a shorter ray length and vice versa. This is done in order to balance visual quality and performance and avoid incoherency (which leads to noise and higher computation time) between neighboring pixels where the roughness is high. On surfaces with a roughness of one they used short rays only 3 meters long as opposed to surfaces with roughness closer to 0 where they set the ray length to 200 meters. They note that these numbers are the result of experimentation so it depends on the content of the game. In order to make reflected objects to appear smoothly into the image when getting closer to the reflecting surface and to hide artifacts cause by the switch between ray hits and misses, they also added a pixel index based random variation to the ray length.

The game engine used in *control* uses a number of different material variations\different shading models for some of its objects, e.g. skin, eyes and hair. That would make the process of computing the reflected color in the hit shader more complex so the developers decided to create a unified material variant that is based on a simpler parameterization of physically based shading and use it for all the materials when performing the ray tracing. This results in a less accurate reflection color but the results where visually pleasing and the performance gained by this simplification was crucial.

As fallback for cases where the rays miss a precomputed GI data was used to approximate the radiance coming from that direction. This approximation is not accurate as it is only based on static objects and light sources. The data's resolution is limited and can potentially cause light leaks, however, the results were good enough for the game.

Control also supports reflection for transparent surfaces using a separate pass. Many levels of the game contain glass windows and other transparent surfaces so being able to render reflections on transparent surfaces was important for the game aesthetics. The transparent reflections algorithm

that was implemented is composed of two parts. In the first part rays are traced against only the transparent objects in the scene. In order to discover visible surfaces only, the length of those rays is limited by the opaque object's depth buffer. If a ray hits a transparent object a secondary ray is generated based on the surface properties in order to compute the reflection color. Transparent objects are a part of the same acceleration structure used for every other object but are marked with a different cull mask so the initial ray tracing doesn't intersect other objects. Only the first transparent object that the ray hits will have reflection, as continuing the primary ray to discover other surfaces would have a high impact on performance.

As with Opaque reflections, only one reflection ray is generated per pixel and is reflected according to the surface normal. Unlike opaque reflections the surface roughness is ignored and therefore, only mirror like reflections are supported. This worked well as most of the transparent objects in the game were mirror reflectors. In the case of transparent reflections, the ray length was limited to 60m (which is shorter than the maximum ray length for opaque reflections but proven good enough). Transparent reflections also use the same environment maps that are used when ray traced reflections are turned off.

Lastly, transparent reflections are not applied immediately to the surface but are applied in separate pass that rasterize and shades transparent objects. In this pass the depth value of the transparent surface is compared with the depth value stored in the reflection texture (the result of the ray tracing pass), when matched the value stored in the texture is used instead of the environment map. Figure 23 shows the result of the algorithm mentioned above in comparison with the standard screen space reflections implementation.

Figure 23 – Screenshots from the video game Control by Remedy Entertainment showing the difference between screen space (on the left) and ray traced reflections (on the right). Image taken from [43] **under the license** [42]

Performance for the RT reflections pass were taken for the same two frames from chapter 5.3.1 and can be observed in Table1 .

7. Ambient Occlusion

Ambient Occlusion (AO) is a basic global illumination effect that fast first developed in the early 2000s by [57] at Industrial Light & Magic in order to improve the quality of the environment lighting that was used for the CG planes in the movie Pearl Harbor. In video games and other real time applications AO is a very common technique used in order to provide better lighting and bringing out object details due to its relatively cheap computational cost. Even though most AO techniques aren't 100% accurate physically and make a fair number of simplifications their result is pleasing and plausible (Figure 24).

The goal of AO techniques is to give an approximation of the amount by which a point on a surface is occluded by the surrounding geometry, which affects the amount of incoming light on that point. Therefore, it's a technique that simulates proximity shadows, e.g., the soft shadows seen in the corner of a room or in narrow spaces between objects. It's a very subtle addition to how lighting is composed into the frame but can drastically improve the realism of the rendered scene. Figure X, shows a scene rendered with AO on and off.

The theoretical explanation of AO can be derived directly from the reflectance equation and it can be described as the normalized, cosine weighted integral of the visibility function:

$$(7.1)\ K_A(P) = \frac{1}{\pi}\int_{1\in\Omega} v(p,l)(n\cdot l)^{+}dl$$

It represents the cosine weighted percentage of the unoccluded hemisphere. $v(p,l)$ is a visibility function that returns 0 if the ray cast from p to l is blocked and 1 otherwise, $(n\cdot l)^{+}$ = max(0, $(n\cdot l)$) and $(n\cdot l)$ is the dot product between the light and the normal to the surface at p. The values are in the range of 0 – for a fully occluded point to one – a point with no occlusion.

Figure 24 – on the left: a scene rendered with Ambient Occlusion, on the right: the same scene rendered without Ambient Occlusion, notice the darkening of the corners and creases when AO is on. Image taken from [58]

The basic idea of the technique is to compute an occlusion factor for each point on a surface and to factor the result into the lighting model, where more occlusion = less light and less occlusion = more light. There are several techniques that are commonly used today to calculate the occlusion factor and offline renderers that don't require high frame rate usually do via Monte Carlo. A large number of rays are cast in a normal oriented hemisphere in order to look for occluding geometry around the point, since this isn't really practical for interactive applications a few different methods for approximating the occlusion factor have been developed over the years.

While offline renderers usually compute the occlusion factor in object space, the most common techniques for real time ambient occlusion are screen space techniques (similar to how most real time reflections are computed in screen space). It's also worth noting that the AO term can be computed offline and then baked into a texture to be used in runtime, this of course has the disadvantage of not supporting dynamic objects, though it's still useful in some cases. However, this work will focus on real time AO techniques.

7.1. Screen Space Ambient Occlusion (SSAO)

One of the most common AO techniques in recent years is called Screen-Space Ambient Occlusion (SSAO). The technique was introduced by the video game Crysis developed by Crytek [59] and was since used and improved in many other video games and applications. In SSAO the AO term for each pixel is computed in a full screen pass using the z-buffer as the only input. For each pixel in the render buffer the AO factor K_A is estimated by testing a set of points in the pixel's neighborhood that can be found by taking a sphere around the pixel's position, against the z-buffer. The value of K_A is the weighted average of all the samples that are in front of the corresponding values in the z-buffer. Each sample has a weight that decrease according to its distance from the pixel. The idea is that if more samples around the original point are inside geometry, than it's more occluded from light sources, and hence darker (has a larger K_A). (Figure 25)

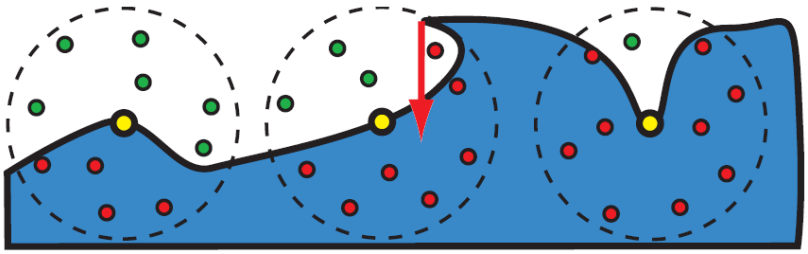

Figure 25 – Crytek's screen space ambient occlusion algorithm applied to 3 different surface points (the yellow circles), shown in two dimensions for clarity. In this example ten samples are distributed over a disk around each surface point (in actuality, they are distributed over a sphere). Samples failing the z test are shown in red while samples passing it are shown in green. The value of the ambient occlusion term is the ratio of the passing samples out of the total samples taken.

Therefore, the AO term in the left most example is 0.6, the one in the middle is 0.3 and the one on the right is 0.1. Image taken from [3]

A disadvantage of this technique is that its quality is proportional to the number of samples taken which needs to be minimized in order to achieve good performance. Reducing the number of samples will produce ugly banding artifacts. Those can be mitigated by randomly rotating the sample kernel for each pixel (and therefore using a different sampling pattern for each pixel). Using a randomly rotated kernel will result in high frequency noise though, but this was easily removed by the developers by blurring the results.
Another drawback of the Crysis algorithm is that the samples are not weighted according to the surface normal and light direction, therefore the resulting ambient occlusion is physically incorrect. Instead of using only samples in the hemisphere that are above the surface, sample points below the surface are counted too, which in the end will result in flat surfaces being darkened with their edges being brighter than their surroundings. (Figure 26)

Figure 26 – The result of the SSAO algorithm developed by Crytek. Image taken from [59]

One way to improve the basic SSAO technique that was introduced by Crytek is instead of sampling from a sphere, sample within a hemisphere that is oriented along the normal at the pixel (Figure 27). This method improves the look of the effect at the cost of requiring per pixel normal data. In modern rendering engine this requirement is trivial as deferred rendering is used in most engines and per pixel normal data is already computed in the g-buffer pass and is available for free when computing AO.

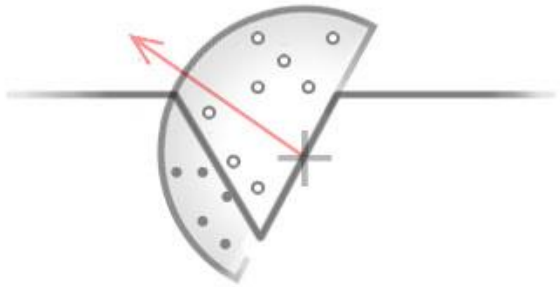

Figure 27 – Shows how a normal oriented hemisphere can be used with SSAO to get better results and improve on some of the drawbacks of the SSAO technique suggested by Crytek. Image taken from [60]

7.2. Horizon Based Ambient Occlusion (HBAO)

Another commonly used AO technique is called Horizon Based Ambient Occlusion or HBAO, it was proposed by [61], [62] and it was drawing inspiration from horizon mapping technique by [63]. HBAO treats the z-buffer data as continuous heightfield. The visibility at a point can be estimated by finding the horizon angles, which is the maximum angles above the tangent plane that are occluded by its neighborhood. In other words, given a direction vector from the point of origin we are looking for the angle of the highest visible object within a certain radius R (Figure 28 - left).

In HBAO we use the following equation to represent the ambient occlusion A at a given surface point P:

$$A = \frac{1}{2\pi}\int_{\Omega} V(\vec{w})W(\vec{w})dw$$

Where V is the visibility function over the normal oriented unit hemisphere Ω, which returns 1 if any ray originating from P in direction $\vec{w}$ intersects an object and 0 otherwise, W is a linear attenuation function.

Using a spherical coordinate system with the zenith axis aligned in the view direction $\vec{V}$, azimuth angle θ and elevation angle α, we split the unit sphere by a horizon line defined by the signed horizon angle $h(\theta)$ and under the assumption that the z-buffer is a continuous heightfield we can re-write the AO occlusion as:

$$AO = 1 - \frac{1}{2\pi}\int_{\theta=-\pi}^{\pi}\int_{\alpha=t(\theta)}^{h(\theta)} W(\vec{w})\cos(\alpha)d\alpha d\theta$$

$t(\theta)$ is the tangent angle between the tangent plane and the view vector.

By using a linear falloff on the distance to the point that defines the horizon for a given θ , the inner integral can be computed analytically and we can re-write the AO term as:

$$AO = 1 - \frac{1}{2\pi}\int_{\theta=-\pi}^{\pi} (\sin(h(\theta)) - \sin(t(\theta)))W(\theta)d\theta$$

The remaining integral can then be computed numerically by sampling in multiple directions and finding horizon angles. Therefore, the algorithm can be described as follows: for each pixel we compute its eye space position p (the z component can be retrieved from the z-buffer), N_d directions are sampled uniformly by picking an angle θ and finding the horizon angle h(θ) in its direction. The horizon angle can be found by ray marching the depth buffer (in a similar manner to the way ray marching was described earlier for SSR). The ray marching is done by taking a radius of influence R that is projected on the image plane and divided uniformly into N_S steps. We start by computing the tangent angle t(θ) and intersect a view ray with the tangent plane defined by the surface point p and the surface normal $\vec{n}$. We ray march the depth buffer in the direction of θ and compute the direction vector $D = S_i - p$ where S_i is the eye position of a given sample S_i (the previously computed N_S samples). For each of those samples we can compute the elevation angle $\alpha(S_i) = \tan^{-1}(-D.z\ /\|D.xy\|)$ [5] and then the horizon angle h(θ) is $\max(t(\theta), \alpha(S_i), i = 1, \dots, N_S)$. Samples that are out of the influence radius R are ignored, for those samples $\|S_i - P\| > R$. Another small detail that has to be done as part of the algorithm is to snap the texture coordinates of the samples along each direction to the nearest texel centers, this is done so that D.z will have the exact same depth associated with the offset D.xy.

Similar to SSAO this technique also adds random jitter to the step size and randomly rotates the N_d uniform directions per pixel in order to avoid banding artifacts. As with SSAO, that leads to high frequency noise which has to be cleaned with a spatial blur filter. In order to achieve a good balance between quality and performance it is useful to run the algorithm at half resolution and use temporal filtering methods to achieve good results.

[5] D is a 3 dimensional vector D.z is a common convention that means the z component of D, in a similar manner D.xy is the 2 dimensional vector made out of the x and y components of D

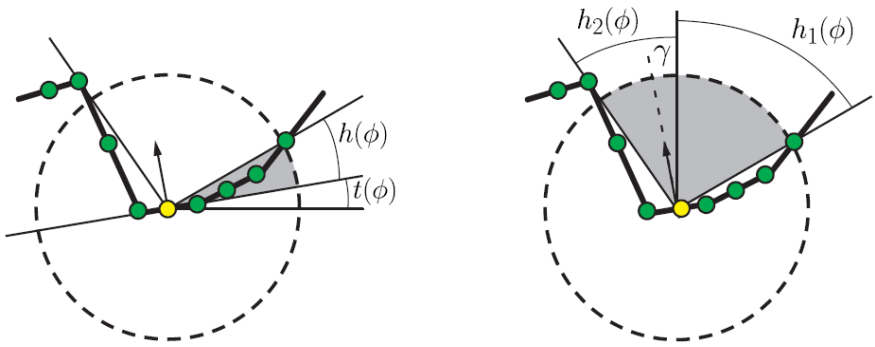

Figure 28 – On the left: Horizon-based ambient occlusion finds the horizon angles h above the tangent plane and integrates the unoccluded angle between them. The angle between the tangent plane and the view vector is denoted as t. On the right: Ground-truth ambient occlusion uses the same horizon angles, h1 and h2, but also uses the angle between the normal and the view vector γ, to incorporate the cosine term in the calculation. In both figures the camera is viewing the scene from above. The figures show cross sections, and horizon angles are functions of ∅, the angle around the view direction. The green dots represent samples read from the depth buffer. The yellow dot is the sample for which the occlusion is being calculated. Image and caption taken from [3]

7.3. Ground Truth Ambient Occlusion (GTAO)

At the time of writing, the state of the art in screen space AO techniques is called Ground Truth Ambient Occlusion (GTAO) and was introduced by Jimenez et al. [64] in 2016. GTAO is based on the same ideas presented in HBAO and aims to achieve ground-truth results that match the results obtained from ray tracing AO techniques. GTAO introduces the cosine factor that was missing in HBAO and also removes the attenuation function, it formulates the occlusion integral around the view vector and the occlusion term is defined as:

$$Ao = \frac{1}{\pi}\int_0^{\pi}\int_{h_1(\emptyset)}^{h_2(\emptyset)} \cos(\theta - \gamma)^+ |\sin(\theta)| d\theta d\emptyset$$

$\emptyset$ is the azimuth angle, θ is a polar angle along the view vector, $h_1(\theta)$ and $h_2(\theta)$ are the right and left horizon angles for a given angle θ and γ is the angle between the view vector and the normal to the surface, and $\cos(\theta - \gamma)^+ = \max(\cos(\theta - \gamma), 0)$. Same with HBAO the inner integral can be solved analytically so we're left with the outer integral to be computed numerically (i.e., by using Monte Carlo integration, therefore computing the value of the integral by averaging samples over the integration domain). The integration is the same as HBAO, we sample a number of directions around a given pixel and ray march the depth buffer along screen space lines to find the two horizon angles (Figure 28 – right).

The inner integral $\vec{a}$ can solved as:

$$\vec{a} = \frac{1}{4}(-\cos 2\theta_1 - \gamma) + \cos(\gamma) + 2\theta_2 \sin(\gamma)) + \frac{1}{4}(-\cos 2\theta_2 - \gamma) + \cos(\gamma) + 2\theta_2 \sin(\gamma))$$

Jimenez et al.[64] points out that the equation above requires that the normal n_x lays in the plane P defined by $t(\emptyset)$ and w_o which in general does not hold. Following Timonen [65] the angle γ is computed as the angle between the normalized projected normal $\frac{\overline{n_x}}{\|n_x\|} \in P$ and w_o as $\gamma = \cos^{-1}(\langle \frac{\overline{n_x}}{\|n_x\|}, w_o \rangle)$ then finally by multiplying the norm of $\overline{n_x}$ we correct the change on the dot product and we are left with the following term for the ambient occlusion factor:

$$Ao = \frac{1}{\pi}\int_0^{\pi} \|n_x\| \vec{a}(\emptyset) d\emptyset$$

The most expensive part of using this technique in both HBAO and GTAO is to sample the depth buffer along a screen space line to find the horizon angle (and in GTAO we have to sample in two directions for each angle) Timonen [65] suggests a method to improve performance for those

computations. He suggests that samples that are used to estimate horizon angles for a given direction can be reused between pixels that lay along straight lines in screen space. The first step is to perform line traces across the z-buffer. At each step of the trace the horizon angles are updated and are written to a buffer. One such buffer is created for each screen space direction used in horizon mapping. The second step is to calculate the occlusion factors based on the horizon information stored in those buffers.

Jimenez et al.[64] suggest that in order to optimize this technique for running at 60 frames percentage, making optimizations are mandatory at the cost of quality. In their paper they performed the AO pass at half resolution which is upsampled later to full resolution. In order to use as much samples as possible while keeping good performance the occlusion integral was distributed over both space and time: the horizon is sampled only one direction per pixel but the information gathered on a 4×4 neighborhood of pixels using a bilateral filter. Temporal filtering is also used aggressively by alternating between 6 different rotations and reprojecting the results, using an exponential accumulation buffer[6]. (This gives a total of 4×4×6 = 96 effective sampled directions per pixel.

The problems with Screen-Space AO techniques

There are two major issues with screen-space AO techniques that are very similar to the limitations represented earlier when discussing screen-space reflections. The first problem is that we are using only the scene information that is available on screen, which means that any object that is outside the camera frustum will not contribute to the AO term, this is especially problematic for areas around the edges of the frame as the only samples that are used will be from within the camera frustum and any sample outside will need to be clamped.

The second issue is that the z-buffer is not a perfect representation of the scene since it only holds depth information for the closest object, thus missing required information about the thickness of the objects in the scene. This issue can be somewhat mitigated with various techniques that try to heuristically infer information about the thickness of objects. While there are techniques that use layers of depth to mitigate the problems (i.e., depth peeling [66]) and can achieve good results, those methods are usually very expensive in terms of performance and are not really viable for real time.

[6] When accumulating results over time using a formula such as:
result = (blendWeight * currFrame) + ((1 - blendWeight) * prevFrame), previous samples are being exponentially weighted down over time, this is sometimes referred to as an exponential moving average.

7.4. Ray Traced Ambient Occlusion (RTAO)

Ray Traced Ambient Occlusion (RTAO) solves the main issues that exist in the previously mentioned screen space techniques by operating in world space. As we discussed earlier in this chapter, our goal is to compute an occlusion factor for each point on the surface by casting a number of rays in a normal oriented hemisphere and look for occluding geometry. We also mentioned that offline renderers use Monte Carlo integration and a high number of samples to compute the occlusion factor. When performing RTAO Monte Carlo integration is used (although with a much smaller sample set) and the AO term equation (7.1) can be estimated as:

$$K_A(P) = \mathrm{E}[\frac{1}{N}\sum_{i=1}^{N}\frac{1}{\pi}V(l_i)(n \cdot l_i)]$$

Where N is the number of samples, l_i is the direction of the i-th ray sampled randomly over the hemisphere and E is the expected value. This estimator however is taking N **uniformly** sampled rays over the hemisphere Ω and cosine-weighting the contribution, which means that some samples will contribute very little to the final result whereas some samples will have more significant contribution. i.e., for a sample that is very close to the horizon the dot product with the surface normal $(n \cdot l_i)$ will result in a very small value.

Figure 29 shows that even a small number of uniformly distributed samples can achieve results that resembles the ground truth AO term. On the left the AO term is computed using only 4 random rays per pixel with uniform distribution, on the right the - the ground truth AO terms computed using 2048 rays per pixel.

Figure 29 – shows how even a small number of rays per pixel is enough to get the general look of the AO term. On the left: the result of an AO pass on a model of a crown running with 4 rays per pixel, on the right: the ground truth AO results on the same model running with 2048 rays per pixel. Image taken from [14] **under the license** [42]

As our goal is to have our AO algorithm run at real time, we want to minimize the number of samples (rays) per pixel to a very small number (usually as small as 1-2 rays per pixel) and therefore choose a very small number of meaningful samples. This can be done by using importance sampling [14].

By using importance sampling the distribution from which our samples are randomly chosen are cosine weighted and therefore there's a higher probability that a ray direction that is closer to the normal is picked. This scheme will allow us to have roughly the same quality while keeping the number of needed rays much smaller.

Since we're using a cosine distribution, we need to divide the estimator by the probability of choosing a given direction l, the probability can be expressed as $p(l) = (n \cdot l_i) / \pi$, which gives us the following equation for the AO term:

$$K_A(P) = \mathrm{E}\left[\frac{1}{N}\sum_{i=1}^{N}\frac{1}{\pi}\frac{V_d(w_i)(n \cdot l_i)}{(n \cdot l_i)\frac{1}{\pi}}\right] = \mathrm{E}\left[\frac{1}{N}\sum_{i=1}^{N}V_d(w_i)\right]$$

This make sure that each ray we shoot has the same contribution (one or zero) while is also more likely to be closer to the surface normal. Figure 30 shows the benefits of using importance sampling, the image on left is using 8 rays per pixel with uniform distribution while the picture on the right shows the results of using 4 rays per pixel with an importance sampled distribution, both with roughly the same variance (same quality).

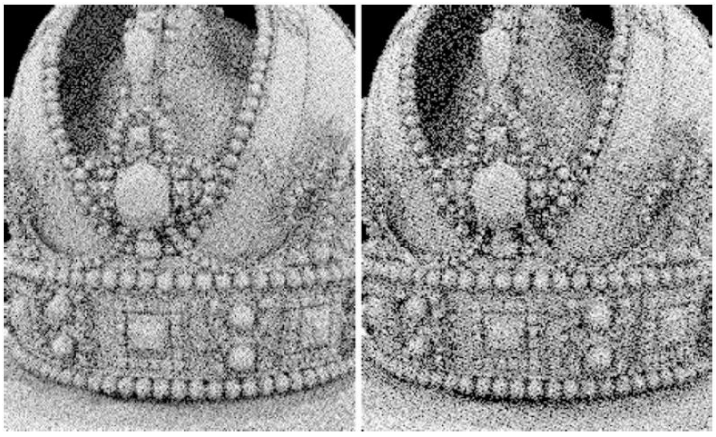

Figure 30 – On the left: a crown model is rendered with eight uniformly distributed samples, on the right: the same crown model is rendered with only four cosine distributed samples. Both results have the same variance showing the benefits of using cosine weighted importance sampling over uniformly distributed samples. Image taken from [14] **under the license** [42]

Although using a small number of samples per pixel means the result will be very noisy, even a very small number of importance sampled rays combined with good denoisers can produce very

good results that looks much better than results achieved with some of the simpler screen space based techniques.

Just like with ray traced reflections and ray traced shadows the rays shot in RTAO can be generated by using an origin point sampled from the gbuffer and depth buffer, this approach is much better suited for a hybrid approach as the gbuffer is already given and we can save on using camera rays. This approach was used in [67][68]. Both implementations mentioned they use 1 ray per pixel combined with an SVGF based denoiser in order to achieve high quality results while maintaining real time performance. [68] measures an average of 0.9ms for the ambient occlusion pass and 0.4ms for the AO denoising pass for a total of 1.3ms average for computing AO in a 1920X1080 resolution on an RTX 3070 GPU. Figure 31 shows the result of the AO pass before and after the denoiser, implemented in [68]

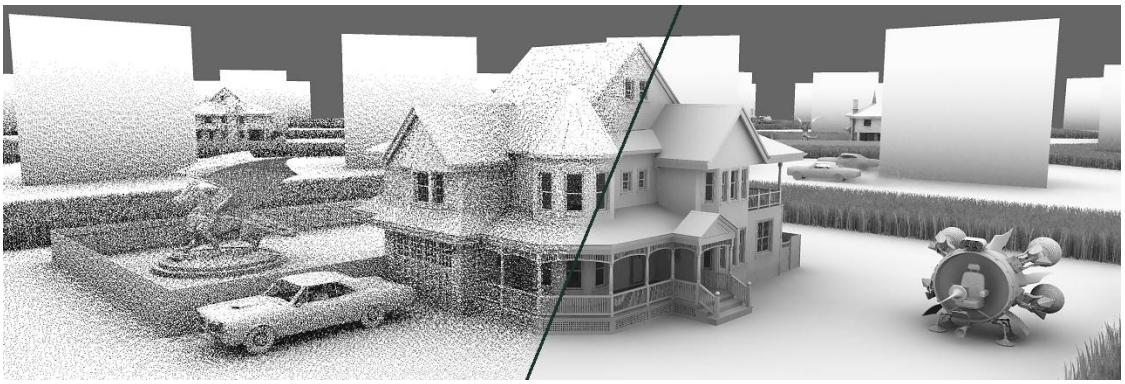

Figure 31 – Shows the result of an RTAO pass before and after denoising. Image taken from [68]

8. Summary

In this work, we've seen the differences between the rasterization pipeline and the relatively new ray tracing pipeline. We discussed how some rendering techniques that were previously only in offline renderers used for films can now be used for run time applications such as simulations and video games.

We have discussed 3 different areas in rendering where ray tracing can be used to improve the realism and quality of the rendered frame: shadows, reflections and ambient occlusion.

In shadow rendering, we've seen how ray tracing can improve shadow quality by adding high quality details to the rendered shadows and tackling some of the problems that exists with shadow maps because of its discrete nature. Ray traced shadows can also be used to better simulate soft shadows coming from area light sources. Without ray tracing area light shadows are hardly used and when used it's usually an inaccurate estimation.

Before ray tracing, reflections used to rely on screen space techniques which usually provide good results but fail in cases when the reflected object is located outside the screen boundaries. Using ray traced reflections or a hybrid screen space – ray tracing solution fixes this problem and provides a much more accurate reflection which can really add to realism.

Even though the recent improvements and innovation in graphics cards and the introduction of specialized ray tracing hardware and APIs can now open the door to using ray tracing in real time, those techniques still take longer to render compared to their raster counterparts so special care must be taken in order to ensure high performance. Since ray tracing support is only available in recent graphics cards there are still many people who can't use it thus it's still not possible to rely on those techniques exclusively and fallbacks to the raster methods needs to stay in place. Usually ray tracing is an option that can be controlled through user settings in order to allow the user to configure the game to his current hardware and desired performance.

As older hardware becomes obsolete and ray tracing support will be standardized, we could see more games and applications that rely on ray tracing by default.

9. Future work

Ray tracing is not new, but developers only recently started using it for real time as hardware accelerated ray tracing features were added to graphics cards. This opened the door for developers to start experimenting with ray tracing and adapting well known techniques to real time while also allowing new innovations in this field.

Real time ray tracing also opens the door to speed offline workflows such as light and GI baking or even rendering animation movies frames. We mentioned earlier that animation studios usually rely on the combined computation power of a rendering farm with a large number of computers. Speeding up ray tracing techniques to real time can reduce the number of computers needed to render such films and can potentially save some of the production costs.

This work touched some of the rendering techniques commonly being enhanced recently with ray tracing. One technique we did not discuss is indirect diffuse lighting (Global Illumination). Global Illumination techniques are often done offline (baked) so that the computed data can be used in real time. Baking the GI data has a significant disadvantage which is the fact that it does not support dynamic objects. Although there are some ways to augment those techniques in order to add some super for dynamic objects it is often costly and complicated. Therefore, real time dynamic GI is considered the holy grail of real time ray tracing and is already a highly sought after and researched topic.

Besides graphics, ray tracing can also be used in other areas in video games and simulations such as physics (e.g., more accurate collision detection and hair simulation), 3D audio [69] and more.

There's still a lot of research to be done on adapting some of the offline ray tracing techniques to real time. As newer hardware adds more powerful ray tracing support and graphics APIs mature and improve, ray tracing techniques will play a larger role in real time rendering. At the time of writing this ray tracing is mainly used to augment and improve on some of the traditional rendering techniques (Since ray tracing for real time is relatively new, but mainly because developers still need to support older and cheaper hardware). As time goes by and those techniques and APIs mature, they will become standard and be used by everyone. It's still unclear if or when we will see applications that fully use ray tracing without using rasterization techniques at all.

As we mentioned earlier, the basic paradigm in using ray tracing in real time is to try to achieve more information with less sample and utilize more sophisticated and finely tuned denoising techniques to get rid of noise caused by the low number of rays per pixel. Denoising and Super Sampling techniques are a key area of research in graphics in recent years and machine learning based denoisers and super samplers in particular.

10. References

[1] T. Whitted, “An Improved Illumination Model for Shaded Display,” *Commun ACM*, vol. 23, no. 6, 1980, doi: 10.1145/358876.358882.

[2] R. L. Cook, T. Porter, and L. Carpenter, “DISTRIBUTED RAY TRACING.,” *Computer Graphics (ACM)*, vol. 18, no. 3, 1984, doi: 10.1145/964965.808590.

[3] T. Akenine-Möller, E. Haines, N. Hoffman, A. Pesce Michaa, and I. Sébastien Hillaire, “Real-Time Rendering Fourth Edition,” 2018. [Online]. Available: http://realtimerendering.com

[4] P. Shirley, *Ray Tracing in One Weekend*. 2016. Accessed: Apr. 29, 2022. [Online]. Available: https://www.amazon.ca/Ray-Tracing-Weekend-Minibooks-Book-ebook/dp/B01B5AODD8/ref=sr_1_1?crid=Q0Y3PILK2QOU&keywords=ray+tracing+in+one+weekend&qid=1651345878&sprefix=ray+tracing+in+one+weekend%2Caps%2C106&sr=8-1

[5] A. Marrs, P. Shirley, and I. Wald, Eds., *Ray Tracing Gems II*. Apress Open, 2021.

[6] E. Haines and T. Akenine-Möller, *Ray tracing gems: High-quality and real-time rendering with DXR and other APIs*. Apress Media LLC, 2019. doi: 10.1007/978-1-4842-4427-2.

[7] T. Akenine-Möller, E. Haines, N. Hoffman, A. Pesce Michaa, and I. Sébastien Hillaire, “The Graphics Rendering Pipeline,” in *Real-Time Rendering Fourth Edition*, 2018, pp. 11–27.

[8] “D3D12_RASTERIZER_DESC (d3d12.h) - Win32 apps | Microsoft Docs.” https://docs.microsoft.com/en-us/windows/win32/api/d3d12/ns-d3d12-d3d12_rasterizer_desc (accessed Jul. 15, 2022).

[9] Tomas Akenine-Moller, Eric Haines, Naty Hoffman, Angelo Pesce, Michal Iwanicki, and Sebastien Hillaire, “Online chapter: Real-Time Ray Tracing,” in *Real-Time Rendering*, Fourth Edition.,

[10] “DirectX Raytracing (DXR) Functional Spec | DirectX-Specs.” https://microsoft.github.io/DirectX-Specs/d3d/Raytracing.html (accessed Apr. 24, 2022).

[11] “Introduction to NVIDIA RTX and DirectX Ray Tracing | NVIDIA Technical Blog.” https://developer.nvidia.com/blog/introduction-nvidia-rtx-directx-ray-tracing/ (accessed Apr. 25, 2022).

[12] D. Meister, J. Boksansky, M. Guthe, and J. Bittner, “On Ray Reordering Techniques for Faster GPU Ray Tracing,” in *Proceedings - I3D 2020: ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, 2020. doi: 10.1145/3384382.3384534.

[13] “AMD FidelityFX - Denoiser - GPUOpen.” https://gpuopen.com/fidelityfx-denoiser/ (accessed Apr. 29, 2022).

[14] M. Pharr, “On the importance of sampling,” in *Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs*, 2019. doi: 10.1007/978-1-4842-4427-2_15.

[15] M. Drobot, “Hybrid Reconstruction Anti Aliasing,” Aug. 2014.

[16] C. Schied *et al.*, “Spatiotemporal variance-guided filtering: Real-time reconstruction for path-traced global illumination,” in *Proceedings of High Performance Graphics, HPG 2017*, 2017. doi: 10.1145/3105762.3105770.

[17] C. Schied, C. Peters, and C. Dachsbacher, “Gradient Estimation for Real-time Adaptive Temporal Filtering,” *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, vol. 1, no. 2, 2018, doi: 10.1145/3233301.

[18] “NVIDIA DLSS 2.0: A Big Leap In AI Rendering | GeForce News | NVIDIA.” https://www.nvidia.com/en-us/geforce/news/nvidia-dlss-2-0-a-big-leap-in-ai-rendering/ (accessed Apr. 25, 2022).

[19] E. Liu, “DLSS 2.0 – IMAGE RECONSTRUCTION FOR REAL-TIME RENDERING WITH DEEP LEARNING,” 2020. Accessed: Apr. 28, 2022. [Online]. Available: http://behindthepixels.io/assets/files/DLSS2.0.pdf

[20] A. Watson, “Deep Learning Techniques for Super-Resolution in Video Games.” [Online]. Available: https://www.nvidia.com/en-

[21] “Radeon™ Software | FidelityFX | AMD.” https://www.amd.com/en/technologies/radeon-software-fidelityfx (accessed Apr. 25, 2022).

[22] “Intel® Arc™- Xe Super Sampling.” https://www.intel.com/content/www/us/en/products/docs/arc-discrete-graphics/xess.html (accessed Apr. 28, 2022).

[23] J. Hasselgren, J. Munkberg, M. Salvi, A. Patney, and A. Lefohn, “Neural Temporal Adaptive Sampling and Denoising,” *Computer Graphics Forum*, vol. 39, no. 2, 2020, doi: 10.1111/cgf.13919.

[24] L. Williams, “Casting curved shadows on curved surfaces,” in *Proceedings of the 5th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1978*, 1978. doi: 10.1145/800248.807402.

[25] “Dealing with Shadow Map Artifacts - Roar11.” http://roar11.com/2015/05/dealing-with-shadow-map-artifacts/ (accessed Apr. 26, 2022).

[26] “Common Techniques to Improve Shadow Depth Maps - Win32 apps | Microsoft Docs.” https://docs.microsoft.com/en-us/windows/win32/dxtecharts/common-techniques-to-improve-shadow-depth-maps (accessed Apr. 26, 2022).

[27] J. Birn, *Digital Lighting and Rendering*, 2nd ed. 2006.

[28] W. Donnelly and A. Lauritzen, “Variance shadow maps,” in *Proceedings of the Symposium on Interactive 3D Graphics*, 2006, vol. 2006. doi: 10.1145/1111411.1111440.

[29] T. Annen, T. Mertens, P. Bekaert, H.-P. H.-P. Seidel, and J. Kautz, “Convolution Shadow Maps,” *2007 Eurographics Symposium on Rendering*, 2007.

[30] T. Annen, T. Mertens, H. P. Seidel, E. Flerackers, and J. Kautz, “Exponential shadow maps,” in *Proceedings - Graphics Interface*, 2008.

[31] M. Salvi, “Rendering filtered shadows with exponential shadow maps.,” in *ShaderX6: Advanced Rendering Techniques* , Charles River Media, 2008, pp. 257–274.

[32] L. Atty, N. Holzschuch, M. Lapierre, J. M. Hasenfratz, C. Hansen, and F. X. Sillion, “Soft shadow maps: Efficient sampling of light source visibility,” in *Computer Graphics Forum*, 2006, vol. 25, no. 4. doi: 10.1111/j.1467-8659.2006.00995.x.

[33] R. Fernando, “Percentage-closer soft shadows,” in *ACM SIGGRAPH 2005 Sketches, SIGGRAPH 2005*, Jul. 2005, p. 35. doi: 10.1145/1187112.1187153.

[34] M. Tamas and V. Heisenberger, “Screen Space Soft Shadows.,” in *GPU Pro 6*, 2015.

[35] S. Seibert and S. Radicke, “Spatial Multisampling and Multipass Occlusion Testing for Screen Space Shadows,” in *SIGGRAPH Asia 2017 Posters, SA 2017*, 2017. doi: 10.1145/3145690.3145692.

[36] J. Boksansky, M. Wimmer, and J. Bittner, “Ray Traced Shadows: Maintaining Real-Time Frame Rates,” in *Ray Tracing Gems*, 2019.

[37] I. Wald and P. Slusallek, “State of the art in interactive ray tracing,” *State of the Art Reports, EUROGRAPHICS*, 2001.

[38] “Hybrid Ray Traced Shadows | NVIDIA Developer.” https://developer.nvidia.com/content/hybrid-ray-traced-shadows (accessed Apr. 25, 2022).

[39] “Hybrid raytraced shadows and reflections – Interplay of Light.” https://interplayoflight.wordpress.com/2018/07/04/hybrid-raytraced-shadows-and-reflections/ (accessed Apr. 25, 2022).

[40] “Ray Tracing | NVIDIA Developer.” https://developer.nvidia.com/discover/ray-tracing (accessed Apr. 25, 2022).

[41] J. T. Kajiya, “RENDERING EQUATION.,” *Computer Graphics (ACM)*, vol. 20, no. 4, 1986, doi: 10.1145/15886.15902.

[42] “Creative Commons — Attribution-NonCommercial-NoDerivatives 4.0 International — CC BY-NC-ND 4.0.” https://creativecommons.org/licenses/by-nc-nd/4.0/ (accessed Apr. 26, 2022).

[43] J. Sjöholm, P. Jukarainen, and T. Aalto, “Ray Tracing In Control,” in *Ray Tracing Gems 2*, A. Marrs, P. Shirley, and I. Wald, Eds. Apress, 2021, pp. 739–764.

[44] T. Sousa, N. Kasyan, and N. Schulz, “CryENGINE 3: Three Years of Work in Review,” in *GPU Pro 3*, W. Engel, Ed. 2012, pp. 133–168.

[45] S. Lagarde and A. Zanuttini, “Practical Planar Reflections Using Cubemaps and Image Proxies,” in *GPU Pro 4*, W. Engel, Ed. 2013, pp. 51–68.

[46] M. Valient, “Reflections And Volumetrics of Killzone Shadow Fall,” *SIGGRAPH*, 2014.

[47] M. Valient, “Killzone Shadow Fall,” in *Devcon*, 2013.

[48] B. Wronski, D. Programmer, and U. Montreal, “Assassin’s Creed 4: Black Flag Road to next-gen graphics.”

[49] “The future of screenspace reflections | Bart Wronski.” https://bartwronski.com/2014/01/25/the-future-of-screenspace-reflections/ (accessed Apr. 27, 2022).

[50] M. Giacalone, “Screen Space Reflections in The Surge.” Accessed: Apr. 27, 2022. [Online]. Available: https://www.slideshare.net/MicheleGiacalone1/screen-space-reflections-in-the-surge

[51] “Indirect Drawing - Win32 apps | Microsoft Docs.” https://docs.microsoft.com/en-us/windows/win32/direct3d12/indirect-drawing (accessed Apr. 27, 2022).

[52] M. Mcguire and M. Mara, “Efficient GPU Screen-Space Ray Tracing,” *Journal of Computer Graphics Techniques*, vol. 3, no. 4, 2014.

[53] L. Sébastien and A. Zanuttini, “Local image-based lighting with parallax-corrected cubemaps,” in *ACM SIGGRAPH 2012 Talks,SIGGRAPH’12*, 2012. doi: 10.1145/2343045.2343094.

[54] T. Stachowiak and Y. Uludag, “Stochastic Screen-Space Reflections,” *SIGGRAPH*, 2015.

[55] T. Stachowiak, “Stochastic all the things: Raytracing in hybrid real-time rendering”.

[56] J. Lawrence, S. Rusinkiewicz, and R. Ramamoorthi, “Efficient BRDF importance sampling using a factored representation,” in *ACM Transactions on Graphics*, 2004, vol. 23, no. 3. doi: 10.1145/1015706.1015751.

[57] H. Landis, “Production-ready global illumination,” *Siggraph course notes*, 2002.

[58] “What Is Ambient Occlusion? (SSAO, HBAO, HDAO And VXAO).” https://www.gpumag.com/what-is-ambient-occlusion/ (accessed Apr. 29, 2022).

[59] M. Mittring, “Finding next gen - CryEngine 2,” in *ACM SIGGRAPH 2007 Papers - International Conference on Computer Graphics and Interactive Techniques*, 2007.

[60] “john-chapman-graphics: SSAO Tutorial.” http://john-chapman-graphics.blogspot.com/2013/01/ssao-tutorial.html (accessed Apr. 29, 2022).

[61] L. Bavoil, M. Sainz, and R. Dimitrov, “Image-space horizon-based ambient occlusion,” in *SIGGRAPH’08: ACM SIGGRAPH Talks 2008*, 2008. doi: 10.1145/1401032.1401061.

[62] R. Dimitrov, L. Bavoil, and M. Sainz, “Horizon-split ambient occlusion,” *Proceedings of the 2008 symposium on Interactive 3D graphics and games*, 2008.

[63] N. L. Max, “Horizon mapping: shadows for bump-mapped surfaces,” *Vis Comput*, vol. 4, no. 2, 1988, doi: 10.1007/BF01905562.

[64] J. Jimenez, “Practical Realtime Strategies for Accurate Indirect Occlusion,” *Siggraph 2016*, 2016.

[65] V. Timonen, “Line-sweep ambient obscurance,” *Computer Graphics Forum*, vol. 32, no. 4, 2013, doi: 10.1111/cgf.12155.

[66] F. Liu, M. C. Huang, X. H. Liu, and E. H. Wu, “Efficient depth peeling via bucket sort,” in *Proceedings of the HPG 2009: Conference on High-Performance Graphics 2009*, 2009. doi: 10.1145/1572769.1572779.

[67] C. Barre-Brisebois, “Full Rays Ahead! From Raster to Real-Time Raytracing.”

[68] “DirectX-Graphics-Samples/readme.md at master · microsoft/DirectX-Graphics-Samples · GitHub.” https://github.com/microsoft/DirectX-Graphics-Samples/blob/master/Samples/Desktop/D3D12Raytracing/src/D3D12RaytracingRealTimeDenoisedAmbientOcclusion/readme.md (accessed Apr. 28, 2022).

[69] W. Mueller and F. Ullmann, “Scalable system for 3D audio ray tracing,” *International Conference on Multimedia Computing and Systems -Proceedings*, vol. 2, 1999, doi: 10.1109/mmcs.1999.778592.

[70] M. Kurt and D. Edwards, “A survey of BRDF models for computer graphics,” *Computer Graphics (ACM)*, vol. 43, no. 2, 2009, doi: 10.1145/1629216.1629222.

[71] S. T. Tokdar and R. E. Kass, “Importance sampling: A review,” *Wiley Interdisciplinary Reviews: Computational Statistics*, vol. 2, no. 1. 2010. doi: 10.1002/wics.56.

תקציר

עד לא מזמן טכניקות רינדור מבוססות על עקיבת קרניים היו בשימוש רק בתעשיית הסרטים והאנימציה, בהן הביצועים פחות חשובים. בפרויקטים כאלו, כל פריים יכול לקחת זמן רב לרנדר ומספר פריימים שונים יכולים להיות מרונדרים על גבי מספר מחשבים שונים בחוות רינדור ומחוברים בסוף לסרט באורך מלא.

במשחקי וידאו, יש משמעות רבה לביצועים וזמני ריצה מכיוון שמחשב אחד צריך לרנדר 30-60 פריימים בשניה (או אפילו יותר במקרה של מציאות מדומה) על מנת לתת למשתמש חווית משחק טובה. על כן, משחקי וידאו וסימולציות צריכים להסתמך על טכניקות רינדור מהירות יותר אשר מבצעות אומדן של חוקי הפיסיקה, כדי לתת תמונה איכותית ומציאותית ועדיין להישאר במסגרת תקציב זמן ריצה מוגבל של מספר מילי שניות עבור כל פריים. הטכניקות הללו מבוססות על תהליך הרסטריזציה בGPU אותו נציג גם כן בעבודה זו.

בשנים האחרונות חלה התפתחות משמעותית בכרטיסי מסך כאשר חומרה ייעודית להאצת עקיבת קרניים התווספה והפכה לחלק סטנדרטי מכרטיסי המסך האחרונים שיצאו לשוק. ההוספה של החומרה הנ״ל פתחה פתח למפתחים להביא טכניקות שעד עכשיו השתמשו בהן רק לסרטים ואנימציה, ולהתאים אותן למשחקי וידאו ואפליקציות גרפיות אחרות, שעובדות בזמן אמת, ולשלב אותם במנועים גרפיים ביחד עם הטכניקות מבוססות רסטריזציה. כיום חברות משחקים וחברות חומרה רבות (כגון AMD, Intel, Nvidia) משקיעות הרבה משאבים במטרה לחקור איך אפשר לנצל את החומרה החדשה על מנת להטמיע טכניקות עקיבת קרניים בצורה האופטימלית בדור הבא של משחקי וידאו וסימולציות.

בעבודה הזו אנו סוקרים ודנים במספר טכניקות רינדור שונות ומסבירים כיצד ניתן להשתמש בעקיבת קרניים כדי לשפר את איכות הטכניקות הללו. הטכניקות שנדונו בעבודה זו הינן: צל, השתקפויות ו- Ambient Occlusion-.

בעבודה נידונים בקצרה סוגים של מסנני רעשים (Denoisers) שניתן להשתמש בהם בזמן אמת, שכן אלו חשובים מאד כאשר מנסים להשתמש בעקיבת קרניים בזמן אמת ותחת מגבלות זמני ריצה.

תוכן עניינים

רשימת איורים 4
1. תקציר 5
2. הקדמה 6
3. The graphics pipeline 8
3.1. Rasterization and the standard graphics pipeline 8
3.2. עקיבת קרניים 12
3.3. סוגי שיידרים 13
3.4. מבני האצה 15
3.5. Coherency 16
4. סינון רעשים 18
4.1. מסננים מרחביים 19
4.2. מסננים טמפורלים (מבוססי זמן) 20
4.3. DLSS & AI based denoisers 20
5. טכניקות הצללה 22
5.1. Shadow Maps 22
5.1.1 Disadvantages of shadow mapping 23
5.1.2 Soft Shadow Techniques 26
5.1.3 Fixed sized penumbra filtering techniques 27
5.1.4 Variable sized penumbra filtering techniques 28
5.2. Screen Space Shadows 29
5.3. Ray-Traced Shadows 30
5.3.1 Ray traced contact shadows in “Control” 38
6. טכניקות השתקפות 41
6.1. Planar reflections 41
6.2. Screen space reflections (SSR) 42
6.3. Stochastic Screen Space Reflections 47
6.4. Ray Traced Reflections 47
6.5. Ray traced reflections in “*Control*” 50
7. Ambient Occlusion 53
7.1. (SSAO) Screen Space Ambient Occlusion 54
7.2. Horizon Based Ambient Occlusion (HBAO) 56
7.3. Ground Truth Ambient Occlusion (GTAO) 59

7.4. Ray Traced Ambient Occlusion (RTAO) 61
8. סיכום 64
9. עבודה נוספת 65
10. ביבליוגרפיה 66

האוניברסיטה הפתוחה
המחלקה למתמטיקה ולמדעי המחשב

שיטות רינדור מבוססות עקיבת קרניים בזמן אמת

העבודה המסכמת מוגשת כחלק מהדרישות לקבלת תואר

"מוסמך למדעים" .M.Sc במדעי המחשב

האוניברסיטה הפתוחה

המחלקה למתמטיקה ולמדעי החטיבה למדעי המחשב

על ידי
עמית עפר

העבודה הוכנה בהדרכתה של ד"ר מיריי אביגל

מרץ 2023