Chapter 1

Recent developments in ray tracing for video games

1.1 Abstract
Since the emerging of 3D video games, polygon rasterization has become the industry standard for projecting worlds onto the screen. As time progressed, graphics accelerators and other hardware was developed to accelerate the process of rendering polygons as games demanded increasingly complex scenes and lighting. And while there are good reasons to stick to conventional rasterization, a rapid increase of hardware complexity over the years has spurred interest in alternative rendering techniques. Now that modern hardware is becoming considerably faster, ray tracing may be considered as an alternative to rasterization. This paper gives a brief overview of the rasterization and ray tracing rendering algorithms, their characteristics and their latest developments and seeks to find out whether ray tracing could eventually be considered a practical rendering alternative for video games.

1.2 Introduction
In the past decades, video games have evolved from primitive two dimensional arcade to full-fledged three dimensional first person shooters. When computational power of computers and consoles increased, games and other graphical applications were made to utilize that increase to display more realistic graphics. As games become almost real to some, the truth is that virtually all computer graphics algorithms responsible for the real-time graphics in today’s games are plagued by various visual and logical shortcomings. At the heart of each modern game engine lies a decades-old technique called polygon rasterization, accelerated by carefully designed pipelined graphics hardware that is capable of projecting many millions of 2D polygons per second. Through the years GPUs have become faster, but they also gained new features like hardware transforming and lighting, uniform shaders and tessellation. Especially because of uniform shaders, GPUs are now also used for general purpose programming, and are very powerful due to their highly parallel nature. While rasterization is heavily used in combination with different lighting techniques to achieve a high level of realism, an alternative rendering algorithm called ray tracing is becoming more interesting now that hardware is becoming more powerful. In this paper we will look at rasterization and ray tracing, we will discuss their advantages and disadvantages, and the practicality of using them in modern video games.

1.3 Rendering Techniques
Rendering in video games is the process of generating images from geometrical models. Geometry deals with the size, shape and relative positions of n-dimensional figures, and spatial properties. The properties of these models are stored in various types of data structures, depending on the nature of the used rendering algorithm and type of geometry the models consist of.

There are many rendering algorithms that can produce the desired images and each algorithm has its strong and weak points. In the case of video games, there are usually two goals:

1. to produce realistic images.
2. to render images at a high enough frame rate to achieve fluid animations.
Both the chosen rendering algorithm and the type of geometry used play a role in achieving these goals. In this section we will describe the rasterization and ray tracing rendering algorithms, and the advantages and disadvantages of these algorithms.

### 1.3.1 Rasterization algorithm

Rasterization is a rendering algorithm for converting a set of shapes to a raster image made of pixels. It is the industry standard rendering algorithm for rendering 3D graphics in video games and is mainly used in combination with polygons. One of the advantages of using polygons and rasterization is that all of today’s GPUs have hardware support for this algorithm, making it very fast and suitable for real-time 3D graphics demanded by 3D video games. Another advantage is that most stages of the graphics pipeline on modern GPUs can be programmed, in contrast to the fixed graphics pipeline. This gives game developers the opportunity to use different lighting techniques to produce more realistic images.

![Rasterization Pipeline](image.png)

Figure 1.1: A typical rasterization pipeline subdivided into functional stages. Image courtesy of [Antochi, 2007].

#### Polygons

Polygons consist of vertices and each vertex is usually specified by four floating point numbers, one for each axis and a homogenous coordinate. The advantages of using polygonal geometry lie in the fact that they consist of vertices, because vertices can easily be transformed by using matrix multiplications. The most used transformations are translation, scaling and rotation, which can be achieved by multiplying the vertices of a polygon with the desired matrix as shown in figure 1.2. If multiple transformations want to be applied to a vertex, then the needed matrices can be multiplied with each other to obtain a matrix that has all the individual transformations. This new matrix can then be multiplied with the vertices of a polygon, which results in the transformations of the vertices with a single multiplication.

![Matrices for Translation, Scaling, and Rotation](image.png)

Figure 1.2: Matrices for translation, scaling and rotation in the X, Y, and Z axis.

These multiplications can easily be performed by modern GPU’s, since they are massively parallel, which results in fast rendering. The downside is that it is impossible to model certain shapes with polygons, like round objects. In general, only triangles are used in rasterizers. This is because triangles have certain characteristics that make the rasterization algorithm (like clipping) much simpler:

1. Every polygon can be broken into a set of triangles through a process called polygon triangulation.
2. All triangles are convex. This means that every internal angle is less than 180 degrees and that every line segment between two vertices remains either inside or on the boundary of the triangle.
3. All triangles used in computer graphics are either planar or degenerate, i.e. there are no other cases.

#### Projection

When all the transformations have been applied to the polygons, they have to be projected onto a two-dimensional plane. There are two types of projections, orthogonal projection and perspective projection.
Orthogonal projection (also called \textit{parallel} projection) involves projecting each vertex onto the viewing plane by a ray that is perpendicular to the viewing plane. This is done by simply removing the \(z\) component of the transformed vertices. The result is a projection where two objects of the same size, that are behind each other, appear equally big. It is illustrated in figure 1.3.

Figure 1.3: Orthogonal projection. Image taken from [Woo et al., 1999].

Since one of the goals of most video games is to be realistic, perspective projection is often used. In this type of projection a pyramidal viewing volume is used instead of a rectangular volume, and the addition of the near and far \(z\) clipping planes it becomes a \textit{frustum}, which is a truncated pyramid. This emulates a perspective projection, where objects that are further away appear smaller. This is shown in figure 1.4.

Figure 1.4: Perspective projection. Image taken from [Woo et al., 1999].

To perform perspective projection, one multiplies the transformed vertices with the matrix shown in figure 1.5, where \(N\) is the near \(z\) clipping plane and \(F\) the far \(z\) clipping plane.

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & (N + F)/N & -F \\
0 & 0 & 1/N & 0
\end{bmatrix}
\]

Figure 1.5: Perspective projection matrix

\textbf{Optimization}

The performance of the rasterization algorithm depends on the amount of polygons that need to be processed. Although GPUs become faster, game developers usually use the increase in computational power for visual effects to improve realism instead of processing more polygons [Huddy, 2006]. So it is beneficial to truncate as many polygons as possible without any visual changes, to be able to maintain a high framerate, which is one of the downsides of rasterization. There are a couple of optimizations that do this, and it is the next step in the rasterization process.

\textit{Clipping} is idea of clipping polygons, that are partially visible, to the boundaries of the viewing plane. This means that objects that lie outside of the viewing plane are truncated. For example, objects that are behind the camera, or beyond the far \(z\) clipping plane are not processed at all, as well as the non-visible parts of objects that are partially visible. There are a couple of algorithms that perform clipping and the most well-known is the \textit{Sutherland-Hodgman} algorithm [Sutherland and Hodgman, 1974], which is the oldest and uses a divide-and-conquer strategy. Figure 1.6 illustrates this algorithm.
Backface culling involves removing polygons that face in the opposite direction of the camera, and therefore are not visible. This of course holds true if all the objects are closed. It is performed by checking the winding order of the transformed polygons. For example, if the vertices of polygons facing the camera are defined in clockwise order, then every polygon with a counter-clockwise order is facing away from the camera. This is essentially checking if the surface normals of polygons are facing the camera.

Occlusion culling is another optimization that removes polygons that are behind other polygons, and is mostly used when there are a lot of objects in the viewing volume. There are different approaches to performing occlusion culling, with some being better suited for different types of video games than others. The most well known are Potentially visible set or PVS rendering, and Portal rendering.

Important to notice is that these optimizations assume that there are no (semi-)transparent or reflective surfaces. The same optimizations can still be used, but it must be taken into account that the amount of polygons that can be seen could increase, and that it could require that a tweaked variant of the optimization has to be used.

Scan conversion

The last step of rasterization is filling in the 2D polygons that are projected onto the viewing plane, or in other words, mapping polygons into pixels and texturing those polygons. Most GPU architectures use the scanline rendering algorithm in order to achieve this, although there are exceptions. One example of such an architecture is the POWERVR graphics chips by Imagination Technologies, which utilize a tile-based rendering approach and are used in embedded devices such as mobile phones [Imagination Technologies, 2009]. In tile-based rendering, the mapping of polygons to pixel does not happen per scanline. Instead, the viewing plane is divided in sectors called tiles, which are processed independently [Antochi, 2007] as can be seen in figure 1.7.

Before the filling can be performed, one has to overcome a few problems, like if a pixel should be drawn. This is because the pixel could be occluded, or could not be inside a polygon. A solution to this problem is the z buffer, which is a 2D array that stores a depth value per pixel. Whenever a new pixel is to be filled, the current depth value of that pixel must first be checked so that only the pixel that is nearest to the camera is filled.

Another problem is finding the color of the pixel that needs to be drawn, and it requires color, texture and lighting calculations. When calculating which texel of a texture has to be used to fill in a pixel, some form of interpolation and filtering has to be used. After the needed texel is found, its brightness has to be determined by some sort of lighting calculation. This is one of the most important steps, because certain lighting techniques can produce very realistic results and make
up for the disadvantage of having to build objects from triangles. Among these techniques are bump mapping and parallax mapping [T. et al., 2001], which give the impression that surfaces that are flat look detailed. An improved version of these techniques is relief mapping [Policarpo et al., 2005], which gives a better perception of depth as can be seen in figure 1.8. Another often used lighting technique is environment mapping [Blinn and Newell, 1976], which is basically a texture that is wrapped around an object that is supposed to be reflective. The texture used obtained by placing the camera at the position of the reflective object, and using the result of that viewing plane.

Figure 1.8: Development of displacement techniques. Left to right: bump mapping [Blinn, 1978], parallax mapping [T. et al., 2001] and relief mapping [Policarpo et al., 2005].

While these lighting techniques can produce realistic results, they consume a lot of computational power. With programmable graphics pipelines, better lighting techniques can be used, at the cost of performance, which is one of the goals of most modern video games. Another problem of rasterization with triangles is that the performance scales linearly with the amount of polygons used [Schmittler et al., 2002].

1.3.2 Ray tracing

Figure 1.9: Examples of ray-traced scenes using the freeware POV-Ray ray tracing suite. Images provided by [POV-Ray, 2010].

Over the past decades, there has been a tremendous amount of research on the topic of ray tracing: a popular rendering technique that is frequently used for photorealistic rendering of still images and television or film special effects. First introduced in [Appel, 1968], the fundamentals of ray tracing are built around a technique capable of generating an image by tracing a path (ray) of light that originates from an imaginary eye point through pixels in a virtual image plane and into a scene consisting of virtual objects, as can be seen in figure 1.10. As each ray travels through the scene, it is tested for intersection with the virtual objects in the scene. Once an object has been encountered, the algorithm typically uses the object’s material along with the ray’s properties and other known lighting information to calculate the final color on the screen. The ray may then bounce off the object’s surface and continue in a different direction in the same fashion until it hits another object. Each ray is typically recursively traced for a specific number of bounces. Different algorithms exist for simulating a variety of optical effects, such as reflection, refraction, caustics and diffuse scattering through global illumination (lighting) techniques including radiosity [Goral et al., 1984], photon mapping [Jensen, 2001] and others. Because ray tracing resembles the physical characteristics of light, it is easily extended by physically correct lighting techniques without the need to use ”fake” approximation tricks as is the case with conventional rasterization.
Real-time ray tracing

Because of the inherent complexity of tracing a considerable amount of rays (millions) per frame, high-quality ray tracing has traditionally been limited to off-line dedicated hardware and distributed "render farms" in film studios. Although the first real-time implementations on SGI supercomputers have existed for a while [Muuss, 1995], the rapid increase of CPU performance and the introduction of programmable GPUs (GPGPU) over the recent years have spurred the development of speed critical real-time ray tracing algorithms running on consumer-grade hardware [Wald et al., 2001b]. Research into the practicality of ray tracing for video games has shown that, with the ongoing high-performance hardware trend, ray tracing may make its entry in the video game industry in the near future [Bikker, 2006].

Spatial subdivision

The major bottleneck of the ray tracing technique is the tracing of the rays through the scene and the subsequent hit intersection tests of the rays with the virtual objects in that scene. As the majority of the computation is spent in calculating whether a ray intersects a particular object, one of the first optimizations is to reduce the set of potential hit candidates for each ray. Instead of naively testing each ray with each and every object in the scene, the scene is spatially subdivided by an algorithm so that only a portion of the objects that are close enough to the ray are considered. A number of different algorithms and structures exist, such as the uniform grid [Cleary et al., 1986], quad- and octrees [Glassner, 1988], binary space partition (BSP-) or kd-trees [Russell and Subramanian, 1988] and bounding volume hierarchies [Rubin and Whitted, 1980]. More recent research has shown special interest in the use of kd-trees for real-time use because of their efficiency and low complexity [Wald et al., 2001a]. Although these structures dramatically increase performance, they only capture a single "shot" of all the objects in the scene. Therefore, the scene is static, and if any of its objects are to be moved, the spatial data structure will often have to be completely recalculated. This is one of the current issues that prevents real-time ray tracing from becoming an instant replacement for rasterization in video games.
Chapter 1. Recent developments in ray tracing for video games

Figure 1.12: A number of typical spatial data structures. Left (cube): uniform grid. Middle: binary space partition tree. Right (grey): bounding volume hierarchy. Images courtesy of [Christen, 2005].

Geometry

The intersection routine is a critical part of the ray tracing technique. The typical choice for the intersection routine is a highly optimized triangle intersection test, allowing for objects to be made out of tessellated polygon meshes, the industry standard for video games. However, ray tracing allows for virtually any type of geometric primitive to be tested against a ray. It is therefore possible to define mathematically perfect primitives and quadrics with a single intersection test: planes, spheres, cylinders, cones, tori and even fractal sets [Crane, 2005]. One can even implement support for boolean operations on geometric primitives, also known as constructive solid geometry [Hijazi et al., 2008], as can be seen in 1.13.

Figure 1.13: Left: various geometrical primitives and constructive solid geometry. Right: quaternion julia sets.

Hardware platforms

Modern CPUs provide an extensive high-performance platform for vectorized implementations with some degree of parallelism using Intel's SSE SIMD extensions, which can be used to significantly speed up the traversal and intersection routines in the ray tracing algorithm. For example, as neighbouring rays shot from the camera are inherently coherent (e.g. they follow similar paths through the scene), rendering can be sped up by tracing packets of multiple rays at once as explained in [Wald et al., 2001a].

In the past decade, the introduction of programmable GPUs from NVIDIA and ATI has caused a shift from pure CPU software implementations towards GPGPU (general-purpose GPU) implementations. One of the major benefits of the GPU hardware architecture is that it functions as a massively parallel platform with a large computational power that can be exploited when using frameworks such as NVIDIA CUDA [NVIDIA, 2010]. As can be seen in figure 1.14, the theoretical GPU peak performance has long surpassed that of state-of-the-art Intel CPUs. Because ray tracing is a process that can easily be parallelized (e.g. every ray can be processed independently and in parallel), a massively parallel implementation can dramatically benefit overall performance. This has resulted in increased academical focus on real-time GPU ray tracing [Günter et al., 2007] and the introduction of commercial interactive ray tracing engines such as NVIDIA OptiX [Parker, 2008] and AMD Fusion Render Cloud [Urbach, 2008] with the goal of improving future video games.

As an alternative to the software CPU and GPGPU implementations, there are also ongoing efforts to develop dedicated ray-tracing hardware. One such example is the Ray Processing Unit (RPU), a PCI-card containing one or more dedicated FPGAs accessible through an API called OpenRT and capable of ray tracing scenes at real-time rates of several frames per second [Woop and Schmittler, 2005]. Another promising example is the CausticRT graphics platform by Caustic...
Graphics that consists of a PCI-express real-time ray tracing accelerator card [Caustic Graphics, 2009]. Although these cards are still in the very early stages of development, the concept shows that real-time ray tracing has the potential to be implemented very efficiently by means of a dedicated hardware architecture.

1.4 Conclusion

We have discussed two rendering algorithms, that are very different from each other. With the introduction of the programmable graphics pipeline, the quality of rasterization has increased dramatically, by the use of shaders capable of creating illusions of depth and detail without requiring a high polygon count. Polygonal objects are also very easy to transform by using simple matrix multiplications, making it very fast and dynamic, which is ideal for interactive video games. The downside is that the computational complexity of rasterization increases linearly with the amount of polygons used, whereas that of ray tracing, if done correctly, increases logarithmically. Still the minimum amount of computational power needed for interactive ray tracing is above the practical limit, even though the core of the ray tracing algorithm is inherently simple. The emergence of massively parallel GPGPU architectures and dedicated ray tracing hardware has increased the potential of the industry acceptance of real-time ray tracing in video games. We hope to see that recent developments will be incorporated in graphics engines so that we may witness the rising of real-time ray traced games within the next decade.
Chapter 1. Recent developments in ray tracing for video games

Figure 1.16: Ray-traced bottles of Jack Daniels. We drank the left one, cheers.

1.5 References


Bikker, J. Real-time Ray Tracing through the Eyes of a Game Developer. 2006.


Christen, M. Ray Tracing on GPU. 2005.


URL http://www.cs.caltech.edu/~keenan/project_qjulia.html


Parker, S. Interactive ray tracing with the NVIDIA OptiX engine. 2008.


