Rendering in computer generated movies

Alexander Epple[1]
Seminar: How to Make a PIXAR Movie
alexander.epple@tum.de
Supervisor: Christian Reinbold
[1] Technische Universität München

Figure 1: *Improvements in rendering capabilities over the span of 20 years [CFS*18, p. 2, 15], [CJ16, p. 118]*

Abstract

Rendering is a challenging process in the making of modern computer generated movies. There are many different approaches to render high quality images, such as rasterization, ray and path tracing. The general rendering pipeline will be explained and real time rasterization will be compared to the REYES algorithm used in movie production. Ray tracing, its extensions, problems and current solutions will be discussed in detail. Finally, hybrid rendering systems and the performance of rendering systems over the years will be discussed in this state-of-the-art report.

1. Introduction

In modern computer generated movies, rendering prevails to be one of the most important and difficult steps. With computing power constantly increasing, it becomes possible to achieve almost photo-realistic computer generated images. Actors that have passed away can be digitally recreated, e.g. Peter Cushing in "Rogue One - A Start Wars Story" as seen in the right most picture in figure 1. Rendering, that is the generation of a two-dimensional images based on three-dimensional objects, virtual cameras, lights, materials and so on [AMHH08, p. 11] can be approached in various different ways. In this paper the two most important techniques, rasterization and ray tracing, as well as hybrids will be discussed. The general rasterization process will be explained and compared to the rasterization-based REYES algorithm that was used in Pixar movies. Ray tracing and path tracing, a more modern and sophisticated approach will be discussed in detail. Hybrid variants of the two techniques are going to be explained using examples found in the movies *Cars* and *A Bugs Life*. Finally, the advantages and disadvantages of those approaches are going to be compared regarding performance and production quality.

2. The Rendering Pipeline

In general, pipelines are used to achieve a speed-up by splitting a process into well defined stages. The rendering pipeline of modern computer graphics applications only slightly differs, no matter which rendering technique is used. It is generally split into three conceptual stages: application, geometry and rasterizer [AMHH08, p. 12]. The first two stages are very similar for the described rendering techniques, the last one is, however, quite different and the main focus of this paper.

2.1. Application Stage

In the application stage, the geometry that should be displayed as well as various other necessary data is transported to the graphics processing unit (GPU). It is the connection between the main processor as well as main memory and the GPU. This stage also takes care of calculations that are not performed in any other stage, such as animations with transformation matrices [AMHH08, p. 15] or generating MIP maps of textures to reduce the amount of memory needed to render the scene by a large amount [CFS*18, p. 2].

2.2. Geometry Stage

The geometry stage itself is split into several parts. The first step transforms the geometric objects into camera or eye space. This is done by transforming the points - also called vertices - that each primitive consists of by adjusting their size, rotation and position. In an intermediate step, they are placed in the scene, then the scene itself is rotated and positioned in a way that positions the camera in the scene's origin, looking in the direction of the negative Z-axis [AMHH08, p. 16, 2.3.1]. The influence of lights in the scene on each vertex is calculated - also referred to as shading - next, depending on the vertex's material [AMHH08, p. 17, 2.3.2]. The following steps are usually skipped by ray tracing renderers. For rasterization based renderers the scene is then projected into a unit cube (the canonical view volume). Before the projection transformation, objects look the same no matter the distance to the camera. This defines an orthographic camera and even though this type is used sometimes, perspective cameras are more common. This type is closer to how humans perceive the world. Hence, objects appear smaller the further away they are [AMHH08, p. 18, 2.3.3]. After the projection transformation, the triangles that are not in the canonical view volume (CVV) are clipped against it. Clipping replaces those vertices outside of the CVV with new ones, which are on the intersections of the CVV and the edges of the triangle [AMHH08, p. 19, 2.3.4]. Finally, the remaining vertices are transformed once more to screen space, scaling the unit cube to the final image size. Coordinates after the transformation represent pixel positions, they are now in so-called "screen coordinates" [AMHH08, p. 20, 2.3.5].

2.3. Rasterizer Stage

The last stage before the final image output is the rasterizer stage. The transformed vertices are now tested for visibility and pixels are drawn according to the scene setup. This stage varies heavily based on the used technique and will be discussed in greater detail later on in sections 3, 4 and 5.

3. Rasterization

The process of splitting the geometry of the scene into pixels and giving them an appropriate color is called rasterization. This process occurs after all transformations are completed and the final image can be computed. There are several ways to resolve the scene into pixels using rasterization, two of which will now be explained.

3.1. General Approach

The traditional version used in most real time rendering applications is done in four stages: triangle setup, triangle traversal, pixel shading and merging [AMHH08, p. 22]. In the first step, the triangle setup, the necessary data for shading is computed [AMHH08, 2.4.1]. In the second step, triangle traversal (or scan conversion), each primitive (triangles, lines, etc.) is then tested against each pixel for coverage. If the pixel is covered by the primitive, a fragment is generated. The data from the previous step is interpolated for each fragment based on the primitive type [AMHH08, 2.4.2]. This process is shown in figure 2. Each fragment is now processed by a so-called shader, that computes the corresponding color. The calculation varies heavily since shaders are programmable. A very common way is to use an image containing surface information to determine the color. This procedure is called texturing [AMHH08, 2.4.3]. In the final stage the processed fragment is tested for visibility. This is done using several buffers, most importantly one that contains depth information of previously processed fragments. If the buffer already contains a fragment, the fragment that is closer to the camera is kept. It must be noted that this algorithm needs to process opaque fragments before processing transparent ones to work. If this is not the case, transparent fragment colors can not be combined (blended) with opaque fragment colors properly, which results in a wrong final color [AMHH08, 2.4.4]. After all fragments have been processed, the image can be stored or displayed.

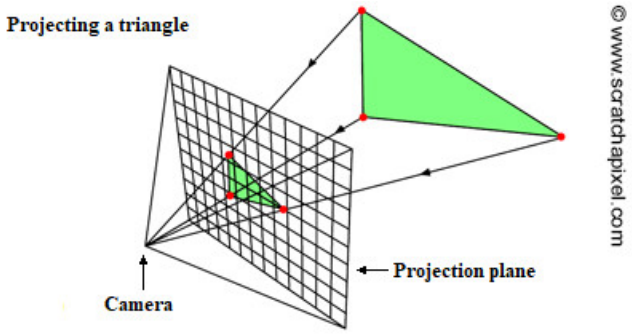

Figure 2: *Rasterization visually explained (adapted from [scr])*

3.2. REYES (*Renders Everything You Ever Saw*)

While the previous approach is fast and the quality is good enough for real time rendering, it has its drawbacks. Traditionally, this version of rasterization has major problems with edges, causing aliasing artifacts. This collided with the image quality requirements of Pixar [CCC87, p. 96, 2.2]. The developers also tried to avoid using traditional solutions or environments in order to achieve the best result possible [CCC87, p. 95, 1.]. Since the goal was to be able to render images more complex and in a better quality than possible at the time, [CCC87, p. 95, 1.] a new rendering technique was invented: The Reyes algorithm. It was used to build Pixar's in-house renderer **Renderman** and produced many animated movies as well as special effects.

3.2.1. The REYES Algorithm

The algorithm has three distinct methods; **bound, dice and split**. **Bound** computes the bounds (with displacement) of each primitive that fully contains it. They do not have to be tight. **Dice** tessellates the primitives into a grid of quadrilaterals ("micro-polygons"), their use will be explained later on. **Split** cuts a primitive into several smaller primitives of either the same or a different type [CCC87, p. 99, 3.]. The implementation used for Renderman is optimized by sorting objects into image tiles ("buckets"). The screen space is divided into an matrix of buckets, which are processed one at a time in a given order [CCC87, p.100, 5.]. The algorithm determines for all primitives in a bucket if they can be diced or if they need to be split first. A primitive may only be considered for dicing when it

does not produce a large grid of micro-polygons or a wide range of micro-polygon sizes. Once a primitive can be diced, it is converted into a grid of micro-polygons. Those micro-polygons are roughly half the size of an image pixel [CCC87, p.97, 2.3]. Figure 3 shows an example of this process. Afterwards, each vertex of the micro-polygon grid is shaded. Finally, pixel colors are computed by stochastically sampling the grid at various points and testing against a depth buffer [CFS*18, p. 2, 2.1].

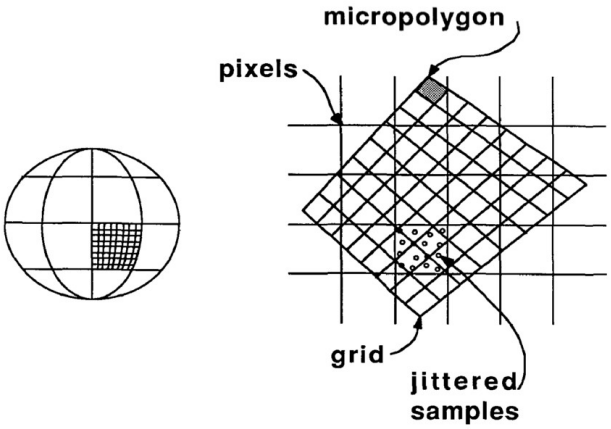

Figure 3: *A sphere made of primitives being split and diced into micro-polygons using the REYES algorithm [CCC87, p. 99]*

3.2.2. REYES versus Standard Rasterization

The Reyes algorithm has several advantages over the technique used in real time rendering in regard to the requirements of Pixar. Firstly, it allows for very complex scenes. Since one image tile typically consists of either 16x16 or 32x32 pixels the computation usually only has to access a small amount of the geometry and textures. The micro-polygons are smaller than pixels, which means it is easy to determine and load only the ones needed for rendering an image tile. Since those micro-polygons are usually part of a small amount of objects, the accessed textures are often the same and already in the cache. This is ideal to keep memory consumption low and avoid texture trashing (loading and discarding the same texture from the cache many times). All data can also be removed from memory after the tile has been processed [CFLB06, p. 2, 4.]. Another advantage lies in Reyes avoiding clipping calculations, since micro-polygons that aren't in the viewing frustum of the camera are culled immediately [CCC87, p. 100, 3.]. Finally, it removes the need for expensive texture filtering, because the micro-polygons are diced in UV-space (texture coordinates). This leads to one polygon in the grid covering almost exactly one texture pixel (texel). Due to this, colors can be directly read from the texture, using the polygon coordinates, without filtering them first. [CCC87, p. 98, 2.4.].

4. Ray tracing

The principle of utilizing rays to generate images has been known for a long time, in fact the concept was already used in the middle ages by Albrecht Dürer to paint pictures with correct perspective projection [JC07, p. 14, 2.1]. This section will describe different ray tracing methods and their advantages.

4.1. General Approach

The easiest algorithm for ray tracing can be split into two calculations per pixel: Step one is to find the closest surface, step two calculates the color of the hit point. Finding the closest surface is also rather simple, since every object in the scene can be intersection tested against the current ray [JC07, p.14, 2.2]. There exist many calculation methods for intersecting rays with primitives, they will however not be discussed. To extend the capabilities, shadow rays can be used to calculate light contribution and, as the name implies, shadows. This is done by tracing rays from each hit point to each light source in the scene or parallel rays for direction lights. If a ray does not hit an opaque object on the way to a light source, that light contributes to the final color, otherwise the hit point is in shadow [JC07, p.24, 2.5]. This results in more realistic lighting and shadows compared to rasterization. It is further possible to render realistic reflections and refractions, which are important for specular and transparent materials. Reflection is the concept of specular materials like metals reflecting rays in the mirror direction. Refraction is important for transparent materials such as glass and water. It deflects a ray, giving it a slightly altered direction through the material. Those new rays influence the color of the original hit point and may lead to more reflection or refraction rays [JC07, p. 25-28, 2.6].

4.2. Recursive Ray Tracing and Path Tracing

Those effects can be computed using *recursive ray tracing* or *path tracing*. The **recursive model** is capable of specular reflections, refractions and shadows. Diffuse reflections and indirect illumination is not considered. For each hit point, a reflection and/or refraction ray is spawned and the illumination contribution of each light source is calculated using shadow rays. This continues until a diffuse and opaque material, or no material is hit. In the **path tracing model**, every ray can cause either a reflection or a refraction ray. The reflection here can be either diffuse or specular. This enables indirect and global illumination. The direction for diffuse reflections, as well as which type of ray is spawned is stochastically chosen. The recursion continues until random termination (also referred to as "Russian Roulette"), when reaching a given recursion depth or if the ray does not hit a surface [CJ16, p. 111, 3.2.]. The recursive concept is illustrated in figure 4, showing a ray scattering into several reflection and refraction rays. Figure 5 is using the path tracing model.

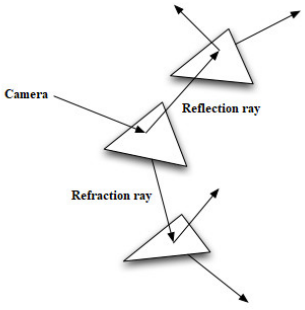

Figure 4: *Recursive ray tracing example [adapted, [JC07, p. 25]]*

4.2.1. Cornell Box Example

To illustrate the various techniques, figure 5 shows how two rays are path traced from a camera through the image plane - or screen - into a scene. The first ray is reflected by the chrome teapot and walls before hitting a light source. The second one is reflected by a wall, refracted by the glass teapot and then reflected again. Each hit point is tested for illumination using shadow rays towards random points in the light sources. Since the used method is path tracing, the diffuse reflection direction as well as the choice of refraction over reflection on the glass teapot is stochastically determined.

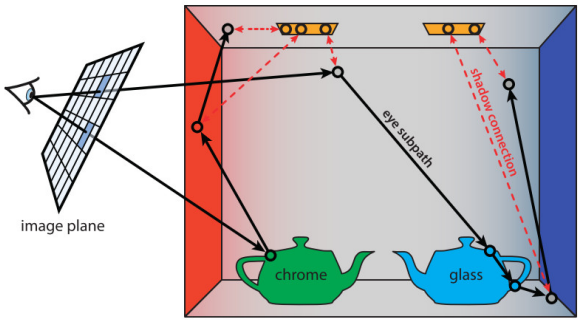

Figure 5: *Several optical effects (reflections, refractions and shadows) in a cornell box using path tracing [CJ16, p. 111, 3.2]*

4.3. Monte Carlo

Monte Carlo simulations are in general used to solve a problem that is analytically difficult or impossible to solve. This is done by repeating the same experiment with random variables. The concept is used in ray tracing primarily to render images with less noise. This is because even shooting many rays per pixel cannot guarantee a noise-free image if there are transparent or reflective materials in the scene. In Monte Carlo ray tracing, ray origins and directions, sampling patterns and more are stochastically determined. This is why Monte Carlo is also called stochastic ray tracing. There are two main categories: **path tracing** as explained in 4.2 and **distribution ray tracing** [JC07, p. 28, 2.7]. The difference to path tracing is that instead of one ray being traced all the way to a light source, multiple rays are traced at once. This leads to a lot of additional rays after just a few reflections. However, a good distribution of ray directions is also ensured [JC07, p.29, 2.7.1]. The advantages of path tracing outweigh the worse distribution and noise in comparison to distribution ray tracing, which is why most modern rendering softwares use path tracing. It is also rather trivial to simulate camera effects such as motion blur and depth of field, both of which is important for movie production. Motion blur can be added by shooting rays at different times within a frame. Depth of field can be added by slightly varying ray origins and directions for each pixel [JC07, p.32-33, 2.7.7/8].

4.3.1. Sampling Patterns

For ray tracing it is beneficial to use sampling patterns instead of random sample points for each pixel, because it can improve convergence and reduce noise. If the number of samples per pixel is known in advance, a sample set can be used. This set contains the randomized sample points. Sample sets are however not used in modern renderers, since it is important that the partially computed image already has low noise. Therefore, ordered sets - so called sample sequences - are used instead [CJ16, p. 130, 5.1.2]. Figure 6 shows different sample patterns, the left most being uniform random samples while the rest are quasi-random samples.

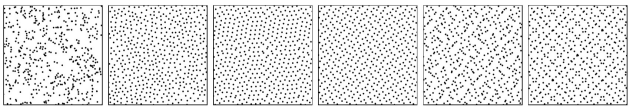

Figure 6: *Various pixel sample patterns [CJ16, p. 130]*

To illustrate the advantage that a sample sequence has over a sample set, figure 7 shows the shadow of a teapot rendered with the same amount of samples per pixel. The image on the left used a sample set, the image on the right a sample sequence.

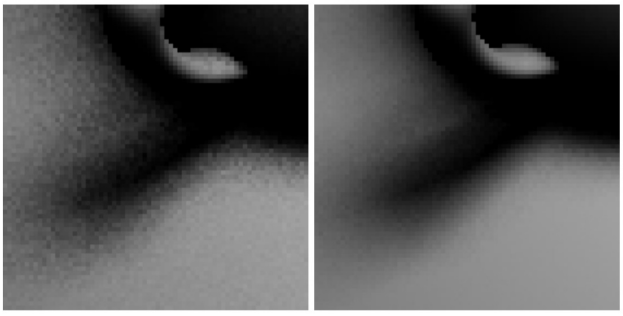

Figure 7: *Results of ray tracing a penumbra region with a sample set (left) and a sample sequence (right) (100 samples per pixel each) [CFS*18, p. 9]*

4.4. Improvements and Speed-Ups

Ray tracing, while unparalleled at rendering high quality images, is inherently slower than traditional rasterization. The amount of computations required for each frame is one big problem, incoherent and unpredictable data access the other one. Back when the first renderers where implemented, it seemed impossible to use ray tracing efficiently. With improvements in computational power as well as algorithmic advances however, it is now the state of the art. Therefore a few of those improvements will now be highlighted.

4.4.1. SIMD and Multi-Core Processors

The first major improvement is using single instruction, multiple data (SIMD) instructions and multi-core processors. Most SIMD processors allow for at least four operations in parallel. Therefore it is, for example, possible to either test four rays against one triangle, or one ray against four triangles at the same time. For coherent rays, speed-ups of 350% are possible [JC07, p. 44, 3.4.1]. While it seems like ray tracing is easy to implement in parallel, this only holds true if the entire scene fits into memory. Using threads, a 120-150% speed-up is achievable, given that the software has been optimized for parallel execution [CFS*18, p. 8, 5.4].

4.4.2. Path Differentials

Another improvement consists of using path differentials. Here a ray also has the information about how a ray with a slightly different origin or direction would have traveled. This prevents incoherent rays, like reflection rays from curved or diffuse surfaces, from trashing geometry and texture caches. Using the "radius" of the ray at impact, it is possible to choose the best tessellation level and texture filter size [CJ16, p. 134, 5.1.3]. A selection of coherent and incoherent rays with ray differentials is shown in figure 8.

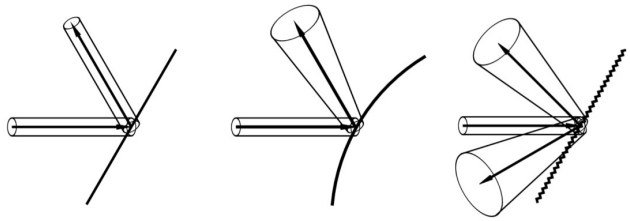

Figure 8: *Specular reflections on smooth / curved surfaces and a diffuse reflection using path differentials [CJ16, p. 134, 5.1.3]*

4.4.3. Denoising

Reducing noise with more samples alone has diminishing returns, therefore denoising filters are necessary. While simple Gaussian blurring can improve image quality, it also blurs edges. Renderman for example therefore uses an advanced filter, it adjusts weights based on color similarities and also considers the following and previous frames for optimal noise removal. Since most noise originates from direct and indirect illumination, the filter can remove textures to reduce noise more accurately. It is also possible to split illumination into finer subdivisions. Experiments with machine learning have already been done as well, denoising filters are however still an area of ongoing research. To illustrate the impact that denoising can have, figure 9 shows close-ups of a frame from *Finding Dori*. The image on the bottom is almost noise free, while preserving all the edges and features of the original image.

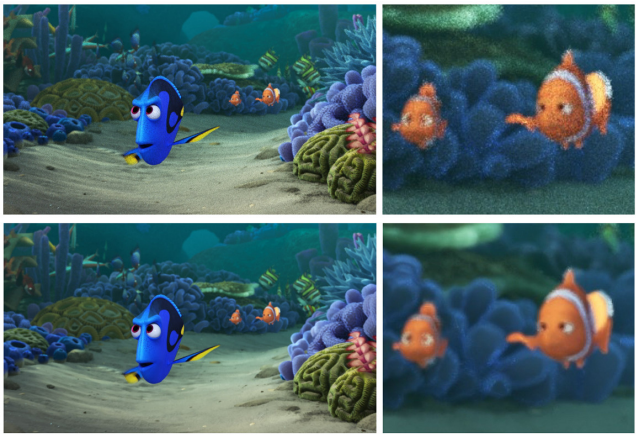

Figure 9: *A frame from **Finding Dori** before and after denoising, as well as close-ups [CFS*18, p. 18, 13]*

4.4.4. Ray reordering

To achieve coherent memory access, it is also possible to reorder rays and ray hits. This can for example be done by collecting rays in bins based on the cardinal directions. The rays can then be sorted further based on the ray origins and directions. This guarantees coherent geometry access. Ray hits can be sorted by which texture they need to access as well. This reduces the amount of times a texture file has to be transferred into memory. Disney's Hyperion renderer for example uses this version of ray reordering [CJ16, p. 136, 5.1.5].

4.5. Bidirectional Path Tracing

To conclude ray tracing, bidirectional path tracing is presented as an outlook on areas of future research. As the name implies, paths are here traced from both the camera and light sources at the same time and then connected. Bidirectional path tracing is especially advantageous in scenes that are dominated by indirect light, because unidirectional path tracers can have a difficult time finding the light source [CJ16, p. 154, 8.1]. While this type of path tracing has its advantages, there are also problems that are not yet solved. Incoherent paths of light sources can cause texture trashing and path differentials are not as useful for bidirectional path tracing as for unidirectional path tracing. Optimal texture filter sizes for light paths can currently only be determined after the path has been fully traced, which does not solve the problem of not knowing the optimal size *during* the tracing process [CFS*18, p. 16, 12.1]. The advantages are however worth exploring the problem further in the upcoming years, as figure 10 shows.

Figure 10: *Unidirectional and bidirectional path tracing results after the same amount of render time [CFS*18, p. 16, 12.1]*

5. Hybrid Rendering Techniques

In the years between fully path traced movies and rasterization-based rendering, several hybrid rendering methods were developed. Those hybrids enabled certain elements or effects in the scene to be rendered with ray tracing instead of rasterization. This section will focus on developments of Pixar's Renderman. One of the first uses of ray tracing was to render reflections and refractions of a glass bottle in the movie *A Bugs Life* (1998), as seen in figure 11. Afterwards, ray tracing gained traction as a means to calculate ambient occlusion, first in the movie *The Incredibles* (2004). Even though

Figure 11: *Examples of ray tracing in combination with REYES rasterization used in the movies* ***A Bugs Life*** *and* ***Cars*** *[CJ16, p. 118, 4.3], [CFLB06, p. 5, 9]*

ambient occlusion requires a lot of rays to be traced, it would have been far more time consuming to evaluate the shaders at ray hit points for e.g. reflections [CJ16, p. 118, 4.3]. The first use of ray tracing for more complex tasks was in the movie *Cars* (2006). Here, REYES was used to render everything directly visible to the camera. The shading points were then able to initiate ray tracing for reflections, ambient occlusion and shadows [CFLB06, p. 5, 7]. While most of the scenes used one level of reflection with a maximum ray distance of 12 meters, there were some shots that needed two levels of reflection [CFLB06, p. 5, 9]. An example of a close up of reflective chrome car parts can be seen in figure 11. Those hybrid use cases were later replaced by pure path tracing. The latest Pixar movies, such as *Coco* (2017), are fully path traced.

6. Performance

Rendering nowadays takes a lot of time, memory and processing power. To illustrate the resources necessary to render even just a single frame, examples from *Cars* (2006) and *Coco* (2017) are going to be presented. This will show that even though the visual and geometric complexity has grown, the render time remains very long and challenging.

Figure 12: *Several scenes from* ***Cars*** *(left) and* ***Coco*** *(right) that are heavy on rendering performance [CFLB06, p. 5, 8], [CFS*18, p. 7, 5.1, 5.3]*

6.1. *Cars*

The scene in figure 12 shows a partially ray traced test scene with 15 non-instanced cars. The shadows, reflections and ambient occlusion are ray traced, everything else is rendered using REYES. This scene was rendered on a 2GHz PowerPC with 2 GB of memory. At full tessellation, the scene features 383 million vertices and 678 million triangles. The required storage for the fully tessellated scene would be 4.6 GB. The total amount of rays needed to render this scene was 174 million, causing 1.2 billion intersection tests. The render time for this single frame was 106 minutes. The rendering occurred with several improvements in place, without those it would have taken almost 9 times as long [CFLB06, p. 5, 8].

6.2. *Coco*

In one of Pixar's latest movies, *Coco*, the average scene consumed 35 GB of memory each frame, with a few requiring up to 120 GB, with several hundred million ray hits per frame [CFS*18, p. 6, 5]. The upper image in figure 12 shows a scene with 20 million objects, some of which are instanced. The average complexity of a scene was in the range of 10-100 million objects. Objects are however tessellated on demand [CFS*18, p. 6, 5.1]. The lower image in figure 12 illustrates the amount of lights that were used in some frames. In this scene, 8 million unique lights had to be considered for shading. For efficient illumination, only a subset of lights were path traced for each hit point. With several improvements the average render time for one frame was cut down from 1000 to just 50 hours [Sey]. The render time is only that long if a single core is used. Most modern movie productions however use so called render-farms, big clusters of many computers, to render faster.

7. Conclusion

In conclusion, rendering remains to be an interesting problem that is far from solved. Even though rasterization such as the REYES algorithm is not commonly used in movies anymore, ray tracing and its challenges remain to be a difficult problem. While improvements such as denoising and bidirectional path tracing have been made, there is still a lot of room for future research. The visual fidelity that can be achieved with modern path tracing may be astonishing, but the complexity and diversity of upcoming films is not going to stop increasing, as has been proven time and time again.

References

[AMHH08] AKENINE-MÖLLER T., HAINES E., HOFFMAN N.: *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA, 2008. 1, 2

[CCC87] COOK R. L., CARPENTER L., CATMULL E.: The reyes image rendering architecture. *SIGGRAPH Comput. Graph. 21*, 4 (Aug. 1987), 95–102. 2, 3

[CFLB06] CHRISTENSEN P., FONG J., LAUR D. M., BATALI D.: Ray tracing for the movie 'cars'. *2006 IEEE Symposium on Interactive Ray Tracing* (2006), 1–6. 3, 6

[CFS*18] CHRISTENSEN P., FONG J., SHADE J., WOOTEN W., SCHUBERT B., KENSLER A., FRIEDMAN S., KILPATRICK C., RAMSHAW C., BANNISTER M., RAYNER B., BROUILLAT J., LIANI M.: Renderman: An advanced path-tracing architecture for movie rendering. *ACM Trans. Graph. 37*, 3 (Aug. 2018), 30:1–30:21. 1, 3, 4, 5, 6

[CJ16] CHRISTENSEN P. H., JAROSZ W.: The path to path-traced movies. *Found. Trends. Comput. Graph. Vis. 10*, 2 (Oct. 2016), 103–175. 1, 3, 4, 5, 6

[JC07] JENSEN H. W., CHRISTENSEN P.: High quality rendering using ray tracing and photon mapping. 3, 4

[scr] SCRATCHAPIXEL.COM: Rasterization: a practical implementation. Accessed: 2018-11-10. 2

[Sey] SEYMOUR M.: Renderman's visuals for coco. Accessed: 2018-12-09. 6