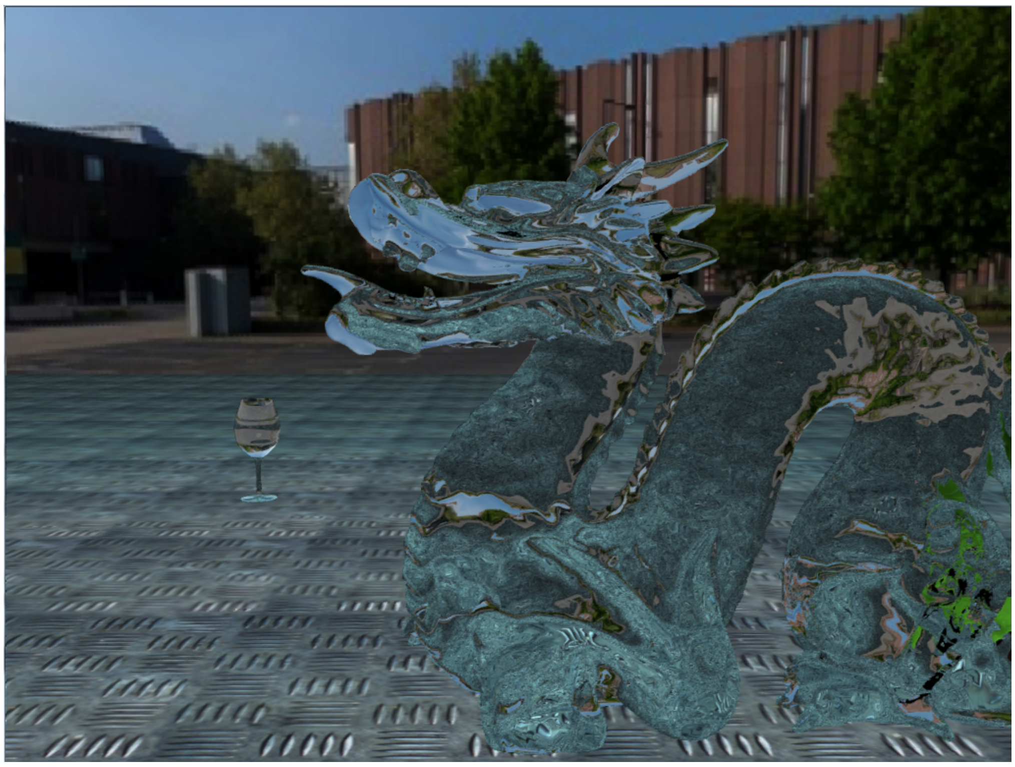
CHALMERS



Improving real time computer graphics quality using hybrid rendering

Master thesis in Computer Science - Algorithms, Languages and Logic

David Sundelius
Department of Computer Science & Engineering
Chalmers University of Technology
Gothenburg, Sweden 2014
Master Thesis

Improving real time computer graphics quality using hybrid rendering

DAVID SUNDELIUS

Examiner: ULF ASSARSSON

CHALMERS UNIVERSITY OF TECHNOLOGY
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden August 2014

Abstract

In computer graphics, there are two general approaches to achieve a picture that resembles reality; rasterization and ray tracing. Rasterization is often used for real time purposes. However, this approach makes it hard and costly to achieve some effects physically correct, for instance reflections and refractions. Ray tracing is a technique that is usually used for offline rendering but makes it much simpler to achieve correct reflections and refractions. Hybrid rendering combines rasterization and ray tracing in a way that tries to benefit from both techniques. This thesis examine hybrid rendering solutions for real time use in the graphics engine Ogre3D using the ray tracing platform Nvidia Optix. The study has resulted in a plugin for Ogre3D that implements some of the usual raytracing techniques and allows the user to mark some of the virtual scene to be ray traced in real time, while the rest of the scene is rasterized.

Keywords: computer graphics, rendering, ray tracing, hybrid rendering

Sammanfattning

Inom datorgrafik finns det två tillvägagångssätt för att skapa en bild som påminner om verkligen; rasterisering och strålföljning. Rasterisering används oftast för att skapa realtidsgrafik men tekniken gör det svårt och kostsamt att åstadkomma vissa effekter fysikaliskt korrekt, till exempel reflektioner och refraktioner. Strålföljning är en teknik som oftast används vid förrenderad grafik och gör det mycket enklare att skapa korrekta reflektioner och refraktioner. Hybridrendering kombinerar rasterisering och strålföljning i ett försök att behålla fördelarna med båda teknikerna. I rapporten undersöks en hybridlösning för realtidsprogram i grafikmotorn Ogre3D som använder strålföljningsplattformen NVidia Optix. Studien har resulterat i en tilläggsprogram för Ogre3D som implementerar den vanliga strålföljningsmodellen och tillåter användare att markera delar av den virituella scenen för rendering med strålföjning i realtid, medan resterande delar av scenen rasteriseras.

Nyckelord: datorgrafik, rendering, strålföljning, hybridrendering

All code created for this thesis are available from https://github.com/davidsundelius.

Contents

Acknowledgements

This thesis work was done in cooperation with EON Reality AB and I would like to thank the development department in Gothenburg for their support and warm welcome to the office. Especially Nils Andersson and my supervisor Vinh-Sang Tri for their help with technical as well as practical issues. I would also like to thank my examiner Ulf Assarsson for the support and help with the thesis.

1 Introduction

This thesis proposes a solution based on combining ray tracing techniques with the use of classic rasterization techniques in real-time computer graphics. This method can generate for instance high quality reflections and refractions (which is a benefit of ray tracing) but still make use of the hardware specialization and the large library of optimized rasterization techniques.

1.1 Purpose

The aim of this study is to develop technology to let programmers and designers render important pieces of an image with ray tracing (in high quality) and the less detailed parts with classic rendering methods in real time. This combination technique could be used in a lot of different real-time software where image quality is an important factor, such as games, development environments, CAD-tools or visualization software.

The work was done in collaboration with EON Reality AB, a company specialized in providing interactive 3d solutions for business and education. The result of the thesis could eventually be implemented in the EON platform for instance to create more realistic images for product advertisement and to improve image quality in other real-time applications.

1.2 Problem definition

The research described in this report evaluates the possibilities of using hybrid rendering with modern computer hardware to create high-quality images in real time. Some issues that will be handled in this thesis are:

- How, and to what cost, can real time rendering using rasterization techniques be improved by using ray tracing on some specific parts of a scene?
- What ray tracing techniques could be implemented in a hybrid renderer to improve quality, while still being usable in real time?
- Is partial ray tracing a feasible and beneficial technique for real time applications on todays hardware?

1.3 Scope

This thesis will focus on the use of Nvidia Optix for ray tracing and Ogre3D for rasterization rendering. This will limit the hardware available for benchmarking to NVidia graphic cards and fairly recent processor/RAM. The use of these systems will also limit the study to not view other alternatives of partial ray tracing that might differ in pros and cons. However, the report studies the theoretical use of hybrid rendering and can be useful for future research on the subject independent of engines and API:s.

1.4 Background and previous work

In the field of computer graphics, the aim usually is to imitate the physical reality as close as possible (although sometimes with some artistic changes). To accomplish this, a simulation of the physical laws of light, as accurate as possible, will be needed. A way of mathematically model this was proposed by James Kajiya in 1986 and is called the rendering equation.

$$L_0(\mathbf{x}, \omega_0, \lambda, t) = L_e(\mathbf{x}, \omega_0, \lambda, t) + \int_\Omega f_r(\mathbf{x}, \omega_i, \omega_0, \lambda, t) L_i(\mathbf{x}, \omega_i \lambda, t)(\omega_i \cdot \mathbf{n}) d\omega_i$$

The equation describes the radiosity of a point in space based on two terms, the light emitted from the point (for instance a light source) and the integral of all incoming light over the hemisphere of the point. In this integral, the incoming light is calculated by taking the incoming light in each direction and multiply the result with the percentage that hits the camera (the dot product of the incoming angle and the normal) and the bidirectional reflection distribution function (BRDF). The BRDF is a function which has unique properties for each specific type of material (reflecting properties for instance).

This equation can however never be completely solved for a point in a computer, since it is eternally recursive. Another problem is that it is infeasible in todays hardware to solve the exact integration problem several times for each pixel. However, we can use approximation models, based on the rendering equation, to generate almost photo-realistic images in ordinary personal computers.

1.4.1 Rasterization

Rasterization is the simplest technique that can be used to display 3d graphics onto a computer screen. The process simply projects all the data points from 3d onto a 2d-plane that is placed in the point where the camera is set. Different matrices is used for this to create an illusion of reality. These are multiplied by all the data points in the scene. The usual matrix model, that are used by most rasterization applications, depends on a a projection matrix, a model matrix and a view matrix (Akenine-Möller et al, 2008).

$$modelViewProjectionMatrix = projectionMatrix * viewMatrix * modelMatrix$$

The projection matrix can be thought of as a model of a camera lens. Changing values in the projection matrix could for instance change the field of view (how wide angle that can be scene from a given point), the screen ratio and at what distance objects are visible. The view matrix models a camera position and orientation in world space coordinates. A model matrix is assigned to each object in the scene (for instance a 3d model) with the task to determine that objects position and orientation. Model matrices can also be hierarchically placed in model objects to simulate objects moving relative to each other. Each joint in an object, when using skeleton based animation, is usually assigned a model matrix that describe the position relative to the model origo. These matrices are multiplied together to create a ModelViewProjection matrix used that is multiplied with each separate vertex to project the data points into screen space coordinates that can be drawn onto the screen.

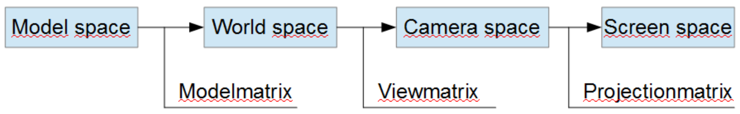


Figure 1.1: The matrix model

The first generated 3d graphics was done by Timothy Johnson in 1963 in a program called Sketchpad III. This program was used to draw simple lines and circles in three dimensions on a computer screen (Johnson, 1963). This was later combined with 4x4-matrixmodel to show and animate 3d images. In the 70's Phong and Gouraud invented the standard shading algorithms to approximate lights, reflections and color in a artificially generated picture (Phong, 1975).

Today, the rasterization (and the matrix-multiplications) are often done on hardware specialized on doing specifically these steps really fast (using parallelization and specialized vector and matrix computation units) since it is fast and still can provide very nice looking graphics by adding different shading techniques. This method is used in almost all real time computer graphics applications today (Akenine-Möller et al, 2008).

1.4.2 Ray tracing

Ray tracing is a rendering technique that was developed in 1968 by Arthur Appel and was based on the idea of treating incoming light as rays that hit the eye or camera. (Appel, 1968) To make this feasible to visualize using a computer Appel calculated the light in reverse, sending rays from the eye and calculating the color of the object that the ray is intersecting. This simple algorithm was called ”Ray casting”. In 1972 Turner Whitter developed the concept into the first ray tracing algorithm by sending out three new rays when a ray hit an object; one reflection ray, one shadow ray and one refraction ray. The results of the rays could then be summarized to compute the resulting color of that intersection. This step could also be repeated using recursion to get a more physically correct result for each beam type (Whitter, 1980). It should be noted that, even though both the Appel and the Whitted approach to ray tracing can produce realistic looking results, it does not fully correspond to the render equation and does not create photo-realistic results. To really simulate all the light in a scene and make it as correct as possible, the ray tracing technique has been further developed into algorithms such as path tracing and global illumination (Kajiya, 1986). These subjects will, however, not be in the focus of this study.

To do efficient ray tracing, an acceleration data structure has to be used to store the environment data to allow fast tracing. These structures are often built in trees to allow the ray tracer to prune branches of the structure and thus eliminating a lot of comparative computations. A simple structure that can be used is called ”Bounding volume hierarchies” (BVH). To build this structure, we divide the scene in rectangular cuboids (since the comparison function of a ray against a plane is a cheap and simple computation). Inside all of the cuboids, the scene is split again into smaller blocks. The more objects there are present within a block, the more the ray tracer gains by splitting them into smaller blocks (Akenine-Möller et al, 2008).

A ray inside a ray tracer is mathematically represented by the equation R = (destination – origin)*k, which is the point-slope form of the common linear equation y = kx+m. Each ray is set equal to the bounding box of a given object (the order and mathematical representation of the object that is used is depending on how the used data structure is representing the scene) and if there is at least one solution where k is a real number, it is determined that the ray hits an object.

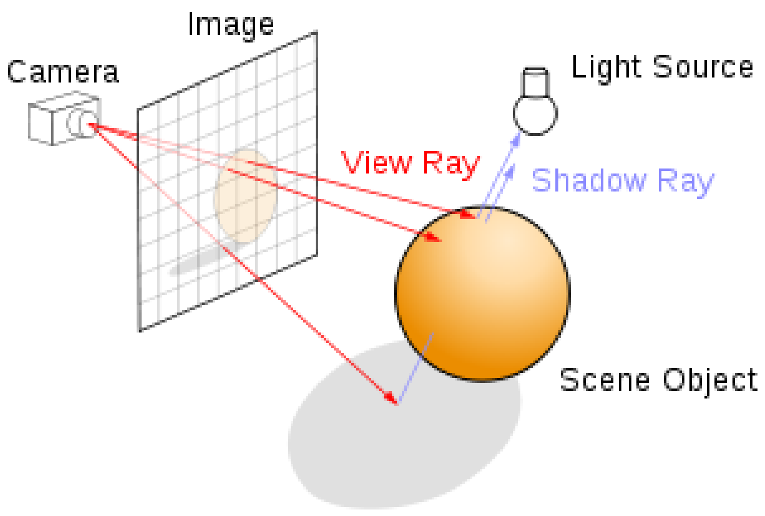


Figure 1.2: Ray tracing

Ray tracing has mostly been used in offline rendering (such as computer-animated movies), since it is a very heavy computational technique but has the possibility to produce a near photo-realistic quality for pictures (Whitter, 1980).

Since the evolution of graphics hardware in the beginning of the 1990's the work to effectively implement efficient ray tracing on graphics cards has made some progress. The pixel shaders enabled the developer to create fairly efficient GPU ray tracers on the programmable pipeline, but it was not until techniques such as CUDA and OpenCL was introduced that the fully flexible pipeline allowed programmers to use the GPU:s potential to create efficient ray tracers on graphics hardware (Steffen et al, 2009).

1.4.3 Hybrid rendering

Hybrid rendering is a concept where the rasterization approach is combined with ray tracing to try to take advantages of both models benefits. For instance, more realistic reflections and refractions can be created using ray tracing than with the usual environment maps normally used for reflections in rasterization (Christansen et al, 2006).

This type of combined rendering technique was not used in industry scale until the Disney Pixar movie ”Cars” in 2006, where the standard Reyes engine was extended to support ray tracing reflections, since this was very important to get a realistic feeling of the cars' metal (Christansen et al, 2006). This engine, however, worked offline to generate images into a video stream.

The technique have been discussed and discredited for some specific purposes in real time rendering (Pohl, 2008). This is due to the fact that hybrid rendering does not provide any performance boost over pure ray tracing in specific cases (for instance, if the camera is zoomed in on a ray traced object). It also requires redundant data storage, in different data structures, to be efficient enough to use for real time purposes.

1.4.4 Shading

Shading is the process of calculating the output color of a specific point of a surface in a 3d environment, dependent on the lights of the scene, properties of the materials, orientation of the object and incoming angle. This process is usually divided into two steps, an interpolation model and an illumination model. The chosen interpolation model determines how the end color of a specific pixel will be calculated depending on the normal of each vertex of a polygon. There are three models normally used for this: flat shading, Gourand shading and Phong shading.

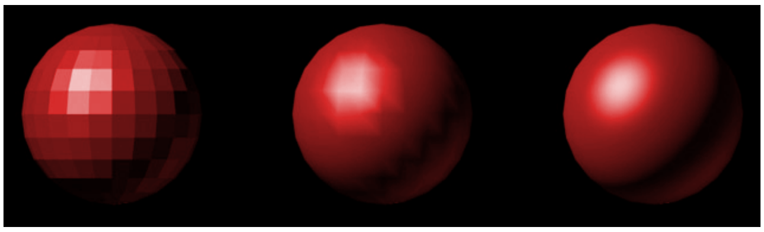


Figure 1.3: The three different interpolation models

Flat shading simply takes the average normalized surface normal based of each vertex normal and use the result to calculate the color intensity for each pixel, resulting in the same color for each pixel on a given polygon. Today this shading model is only used when other models are too costly for the task or for debugging purposes since it does not give a good approximation of reality.

Gourand shading calculates the color for each vertex normal of a polygon and then uses bilinear interpolation of the color intensities of each normal over the pixels of a polygon. This is a fast method to achieve fairly good results. However it can give insufficient results for the specular reflections of a surface.

Phong shading calculates a new normal for each pixel on a polygon using bilinear interpolation between each of the vertex normals. This approach is the most used today and gives good results for most cases. Other models have been presented, but the Phong shading model has a good balance between physically correctness and speed (Akenine-Möller et al, 2008).

1.4.5 CUDA and Optix

CUDA (Compute Unified Device Architecture) is Nvidias platform and programming model for use of the parallell computational power of the GPU. This model allows programmers to directly access the instructions and the memory on the GPU, as well as control over the threading process and use the graphic cards as general processing units and since usable for other purposes than computer graphics.

In the CUDA SDK, a compiler for C/C++ (nvcc) is provided as well as a debugger and several other usable tools for GPU computation. These can be used to program high level applications directly in the same language (often C, or one of the dialects) that is used for the rest of the project and to easily move code from the CPU to the GPU (or the other way around) for conventional optimization of the computational pipeline.

Nvidia Optix is an API (implemented mostly in CUDA) used for general purpose ray tracing using the GPU. It provides a stable, generic and fairly fast way (both performance and implementation wise) to create ray tracing applications that are highly parallellized and make use of the latest features in the GPU. Optix is implemented in two different parts: the host based API to control the ray tracer from within a C/C++ environment and the device based CUDA-style programming language used to write the programs used within the ray tracer (compiled using nvcc) (Parker, 2010).

1.4.6 Ogre3d

Ogre3d (Object-oriented graphics rendering engine) is a very popular and proven open source graphics rendering engine that is easy to use and makes it simple to produce fast results using classic rasterization techniques. It is also a very customizable system that allows for straightforward manipulation of most of the rendering steps. The Ogre engine abstracts away the rendering API:s (for instance OpenGL or DirectX) using the plugin system, making it possible for most applications to support several different techniques for rendering.

The engine is licensed under the MIT license and are completely open source, crowd developed and free to use in any application. It relies on an active community of developers and companies that supports the project and uses it for their own products.

It supports a highly customizable plugin system to encourage developers to contribute modularized solutions to extend the engine (Ogre, 2014). Several plugins, that span from providing new rendering systems and stereoscopical view to particle systems and special scene loaders, are available today. The engine is built upon the factory pattern and with precompiled plugin libraries it is possible for developers to change almost any factory in the engine to produce specialized objects fitted for the developers purposes. As an example, the Ogre tutorials mentions the BSP Scene loader plugin, which replaces the scene class, with a specialized method for parsing and translating a BSP-file to a format usable by the engine.

1.4.7 Software engineering methodology

The requirements management of a project is the process of elicitation, analyzing, specifying, maintaining and verifying the requirements on a system. The methods for these different stages varies depending on the chosen software engineering method. In the waterfall model, the requirements are identified in the beginning of the project, well specified, never changed during the process and in the end verified against the end product. In an agile development process on the other hand, the requirements are identified along with the development and the specification is compact but updated every predicided time fragment, to more closly mimic the changes in the requirements of a system (Sommerville, 2007).

The evolutionary development process is an iterative process, often used in more research heavy development. The process focus on short, iterative sprints where functionality is added each new continuous delivery. It also involves always having a runnable version of the software after each sprint. The advantages of this is, among others, to always be ready for a demonstration of the progress and make it possible to easily roll back the progress if a new feature does not work. Both abilities make it fit for research development. (Sommerville, 2007).

1.4.8 Open source software

Open source software is a collective term used to describe the type of software where the user has access to the full, or parts of, the source code of a program. The term is often misinterpreted as meaning that a given software is free, as in no cost, to use. Most programs that are open source are however also free to use as well. Some of the most well known open source programs are the GNU/Linux operating system (including the most known distribution Ubuntu), the web browser Firefox and the webserver Apache.

There are many advantages with using open source software. Transparency, to be able to review the code and see what the program does in detail, is an important part of the delivered program. The possibility for anyone to review the code could also lead to increased security, since anyone trying to break the system knows all there is to know about the system and could also fix holes in the software. Another advantage is the possibility to modify the code to make the program more fit for its intended purpose (Stallman, 2013).

Some, however, argue that this type of software development results in obscurity in the code, design and interface of the programs. It also often lacks professional support needed to be used in the industry by some companies and it may lead to bad quality software (Tarver, 2009).

2 Analysis

This section describes the theory behind the work process and the method used to solve the problem at hand. It also discusses and motivates the design choices and different alternatives that was discovered during the work.

2.1 Method

To answer the research questions defined in earlier paragraphs, a proof of concept program was implemented in C++, Ogre, Nvidia Optix, CUDA and related tools and scripting languages. It is also the basis of the rendering part of the EON platform. Nvidia Optix is a flexible ray tracing API which uses CUDA to create a pipeline specialized at ray tracing. It provides a relatively simple interface to implement a ray tracer on the GPU. The implementation is based on the theory specified in Chapter 1.4 of this report and the specification on the API:s of the chosen technology.

2.1.1 Literature study

The thesis work began by doing research on the given topic of ray tracing, rasterization and hybrid solution. This was followed a comparative study of software engineering methods, requirements engineering and open source methodology. Finally, the different technology choices of the project (languages, API:s, etc.) that was possible to use for the proof of concept was examined.

2.1.2 The development process

Since the project was fairly small from a software engineering perspective and only one programmer worked on the task, a simple iterative and evolutionary approach was chosen for the development of the proof of concept program. A requirements list was produced for the project, and from this, a UML diagram was designed to grasp the programs architecture from the beginning. These models was revisited and redesigned at a weekly basis to take into account technical limitations, new ideas and new acquired knowledge of the used technology.

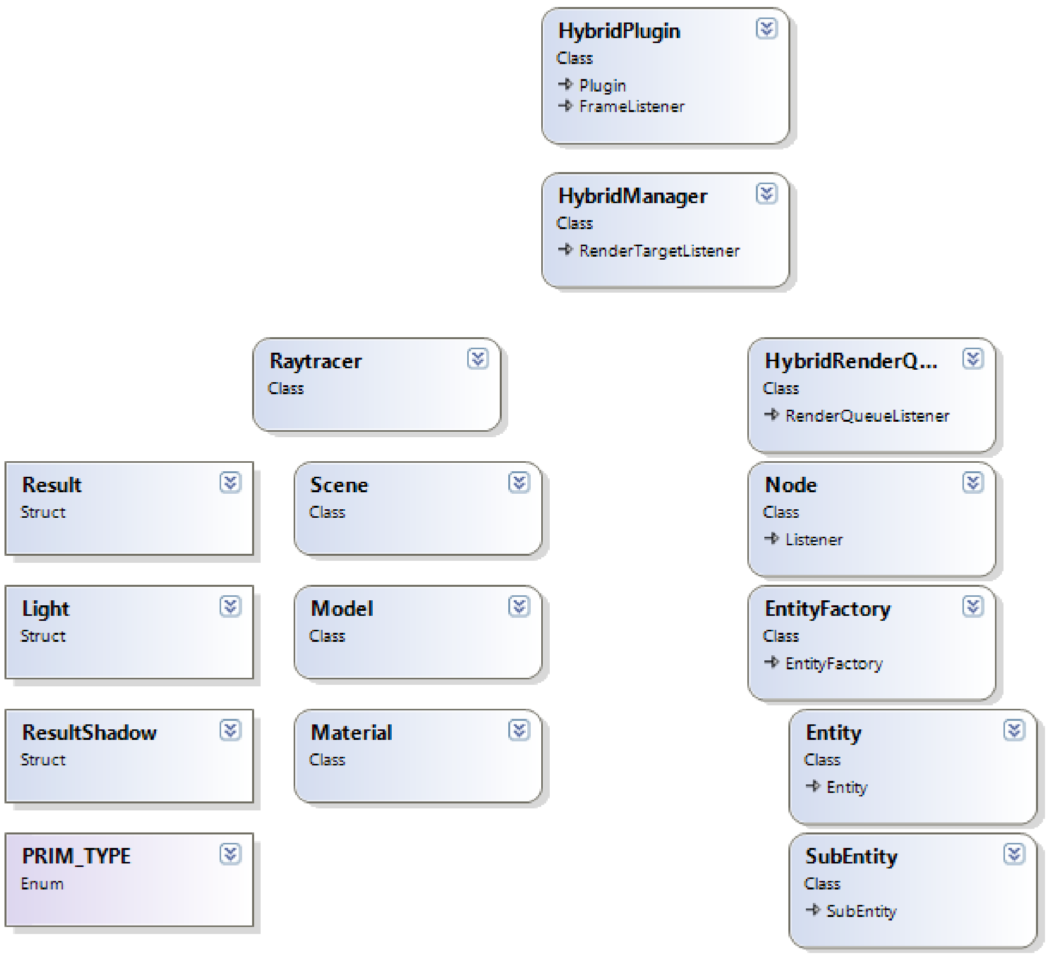


Figure 2.1: The UML diagram of the HybridPlugin project

2.1.3 The final requirement list

- The program should be an implemented solution to show the possibilities and problems associated with hybrid rendering.
- The program should be a plugin for Ogre3d and depend on Optix for hardware ray tracing.
- The program shall depend on OpenGL and rely only on OpenGL, Ogre and Optix libraries (no other third party software should be needed to run the program).
- The program should be able to run at a framerate of at least 15 fps for simple scenes containing both reflective and non-opaque entities.
- The program should be able to be installed in an pre-made engine based on Ogre3D

without corrupting data or otherwise change configurations or data unrelated to the hybrid rendering.

2.2 Implementation

The implementation of the project was divided into four parts that was separated into different processes to increase the modularity of the program and to be easier to understand.

1. Implement a ray tracing engine.
2. Find a way to choose what parts of the screen that is supposed to be ray traced.
3. Construct an interface for Optix to transfer the ray traced data back into Ogre.
4. Create a translator for transferring environment data form Ogre to Optix.

The first part was to implement a modularized ray tracing rendering engine in Optix. This renderer used OpenGL and GLUT to create a window to render a simple scene using the C++ wrapper of Optix to get an object oriented ray tracing component that can be used to interact with a rasterization engine.

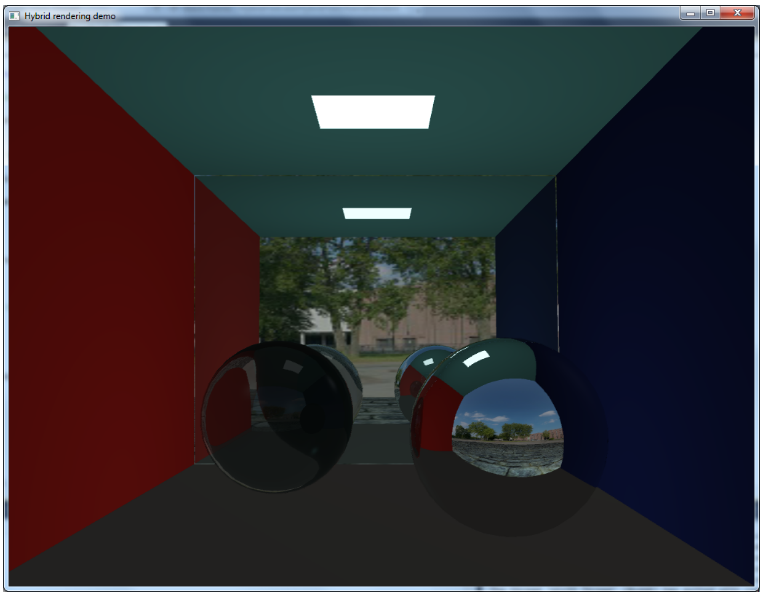


Figure 2.2: This picture shows a simple scene called the Cornellbox, rendered using the Hybrid Renderers ray tracer. The right sphere is a total mirror while the left sphere is semi transparent and semi reflective. The front wall is a mirror that reflects the whole scene, including the skydome outside the box.

The second part was to implement a way for the hybrid renderer to know what part of the screen that is supposed to be ray traced. Thirdly, the Optix part of the program had to present the result in a way that was usable for the hybrid program. Lastly Optix needed to understand the loaded environment including scene geometry, lighting, camera settings and materials.

Then the finished hybrid rendering program was refracted into an Ogre plugin to provide an easy interface for programmers to use in their own graphic engines. When the implementation phase was over the remaining time of the thesis project was used to benchmark and optimize the implementation to get the data presented later in this thesis.

2.2.1 System overview

The implementation is called OgreHybrid and is an Ogre plugin that allows the users to mark objects in a scene (imported in any way, for instance manually created or loaded using another plugin) to be ray traced using a real time ray tracer. The aim with the plugin was to make as little change as possible to the standard state of the program to allow the user to activate the plugin to work with Ogre applications designed for other purposes to enhance their appearance.

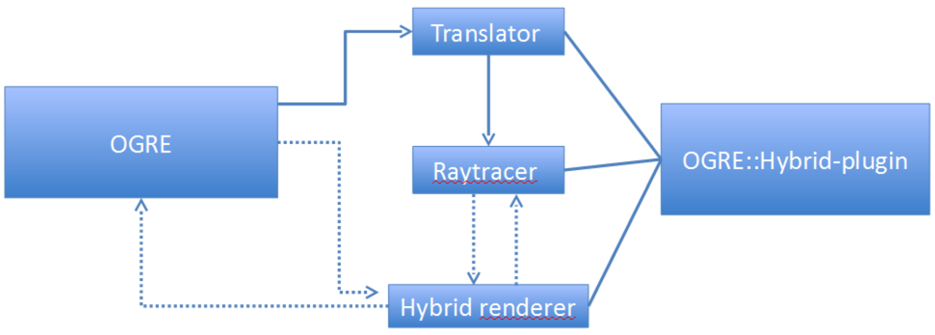


Figure 2.3: The components of the OgreHybrid plugin. A solid arrow means that a component sends/receives data at startup and then observes the original data for a change. A dotted arrow means that the component communicates each frame. The solid lines interprets as ”is a part of”.

OgreHybrid is built in three separate components; the ray tracer engine, the translator and the hybrid renderer. The three components are designed to be used as stand alone components, e.g. the ray tracer could be connected to another component that just views the output to the screen. The translator component synchronizes the data between Ogre and Optix. The ray tracer component is a fully workable ray tracer, with the addition that it takes a mask (with the same resolution of the rendertarget) as input that determines if a specific pixel should be ray traced. The hybrid renderer component is responsible for rendering the mask texture and controlling the Ogre objects. The communication of data between the components are controlled by the HybridPlugin master class (see figure 2.1 for UML diagram).

2.2.2 Hybrid renderer component

The approach chosen to accomplish hybrid rendering was to let the user mark the parts of the models in a scene that they want to ray trace. This allows for a lot of flexibility and makes it easy to customize the renderer for specific scenes.

Other approaches discussed to solve this problem was to let some materials be ray traced. This, however, would not be as simple to use, since that would force the user to write separate materials for ray-traced objects and would not allow for users to just install the plugin in their engines and use it without any settings or customizations of the application. Another solution proposed was to write an algorithm to choose what parts of a scene that would benefit the most from being ray traced and activate hybrid rendering for these areas. This approach would be interesting, it is however quite complicated to implement and still would have artists change the proposed settings since it does not exactly fit the given scene.

When the OgreHybrid plugin is initialized, it adds a new rendertarget and sets it to render a specific hybrid material scheme. It also adds a black solid material to all entities in the scene and connects that material to the hybrid rendering scheme. When an object is marked as ray traced, the hybrid material is changed to a solid white color. This makes the rendertarget create a black and white stencil each frame that is sent to the ray tracer as an input buffer.

When the ray tracer has generated a result, this buffer somehow has to be transformed into a texture that is usable to show onto the screen. Several different approaches was tested to achieve this; the simplest being saving Optix-output to a buffer on the CPU and then loading this as a texture in Ogre by looping through the output buffer and insert the result into a texture. This is a very slow approach and slowed the program down several frames per second,. The next approach is to use PBO:s and simply copy the data from the CPU into GPU memory in one go (using memcpy). This is a lot faster but still very slow. It is, however, the solution in the final product, since it is possible to do on all computers independent of hardware and since there are several limitation to using PBO streaming in Ogre and it is not possible to write directly to the texture memory from Optix. Several different approaches was implemented and tested. However, none of them worked properly, since the previous mentioned limitations of Ogre and Optix. This will be fixed as soon as Optix is updated to make use of CUDA 5.5, where there is a feature called surfaces that can can be used to write directly to texture memory. This feature would allow OgreHybrid to skip a lot of data transfers. However, surfaces require hardware based on the Nvidia Kepler core, which is not present in most personal computers at the printing of this report.

Another approach, which was tested, is the use of pixel buffers generated by Optix and then streamed to an Ogre texture. This was implemented very easily using pure OpenGL using the algorithm:

```
glBindTexture(GL_TEXTURE_2D, target);
glBindBuffer(GL_PIXEL_UNPACK_BUFFER, vboId);
glPixelStorei(GL_UNPACK_ALIGNMENT, 4);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, width, height, 0,
    GL_BGRA, GL_UNSIGNED_BYTE, 0);
glBindBuffer(GL_PIXEL_PACK_BUFFER, 0);
glBindTexture(GL_TEXTURE_2D, 0);
```

Pixelbufferobjects (PBO:s) are however not supported in the current version of Ogre to create textures, which makes it impossible to use for most purposes without patching Ogre manually. More on this issue, its consequences and solutions are discussed in the results chapter.

2.2.3 Ray-tracer engine component

The ray-tracer component is a ray-tracing engine based on Optix, which takes a scene at startup from the translator component and a mask texture, from the hybrid component, identifying the regions of the screen that is supposed to be ray traced and then outputs a buffer that is filled with the color values of the ray-traced pixels.

This component is also capable of doing full real-time ray tracing of a given scene, if connected to other components. For instance, the simple OpenGL renderer used in the testing of this component works for this purpose.

Ray tracing is done using the Optix model for ray tracing. Thus, the device-part of the ray tracer (the code running on the GPU) is divided into 7 different types of programs; the ray generator, miss, exception, closest hit, any hit, intersection, bounding. There are also selector-programs available. These are, however, not used in OgreHybrid.

The ray generation program handles the input and output buffers and generates rays for each pixel on the screen that is marked by the input buffer to be ray traced. It also implements dynamic super sampling to get less aliasing in the result buffer. This feature can be configured during runtime from the OgreHybrid-plugin.

```
RT_PROGRAM void generate() {
  float2 sc = make_float2(index)/make_float2(outBuffer.size());
  if(tex2D(mask, sc.x, sc.y) != 0.0f || skipMask) {
    float2 d = sc * 2.0f - 1.0f;
        float3 orgin = camPos;
        float3 direction = w + d.x*u - d.y*v;
        Result res;
        float3 color = make_float3(0.0f);
        float3 tmpDir;
        for(int i=0;i<numSamples;i++) {
          if(i==0)
            tmpDir = direction;
          else if(i==1)
            tmpDir = direction+epsilon*u+epsilon*v;
            else if(i==2)
              tmpDir = direction-epsilon*u+epsilon*v;
            else if(i==3)
              tmpDir = direction-epsilon*u-epsilon*v;
            else
              tmpDir = direction+epsilon*u-epsilon*v;
            Ray r = make_Ray(orgin, tmpDir, rayTypeRadience,
                             0.005f, RT_DEFAULT_MAX);
            res.depth = 0;
            res.importance = 1.0f;
            rtTrace(scene, r, res);
            color+=res.color;
          }
          color/=numSamples;
          outBuffer[index] = make_float4(color,1.0f);
  } else {
    outBuffer[index] = make_float4(0.0f, 0.0f, 0.0f, 0.0f);
  }
}
```

The miss program is called when a ray misses all objects in the scene. In OgreHybrid this calculates a color from the environment map passed from the translator to imitate the skydome or skybox from the Ogrescene.

Exception programs are called if, at some point during execution of a thread in Optix, an exception is thrown. These exceptions could for instance be a stack overflow or an out-of-memory-exception. In these cases, the specified pixel is given a distinct pink color to be easily identifiable during debugging.

Intersection and bounding-box programs are used to describe specific geometry in a scene of Optix. Bounding-box programs are called to check for a larger intersection and are often simpler, while the intersection program is run if the bounding-box program for a given object is marked as hit. OgreHybrid includes two different geometries: planes and spheres. However, since most rasterization engines work with triangle meshes, all geometry directly converted through the Translator component use the paged triangle mesh loader included in Optix utilities package.

The closest hit program can be thought of as a pixel shader in classic rendering. It determines the color returned by a specific ray depending on the closest hit objects properties. OgreHybrid contains an implementation of Phong shading that is dependent on the material properties fetched from the Translator as well as values of reflectiveness, opaqueness and refractive index of an object given as input to the object by the user. It also generates shadow rays to calculate hard shadows for the hit point, reflection rays for reflective objects and refraction rays for non-opaque objects.

```
RT_PROGRAM void closestHit() {
  //Calculate normals and ray hitpoint
  //Calculate each light components contribution
  //to the pixels end color:
  //Ambient
  //Direct (diffuse, shadow, specular) for each light source
  //Reflection, Refraction, Emissive
  res.color = tA + tD + tRefr + tRefl + emissive;
}
```

Any hit programs determines if a ray has hit an object. It does not, however, get any information on where along the ray the object is hit, which makes it perfect to use for calculation of shadows (where we only need to know if a point is occluded from a light source or not). In OgreHybrid, AnyHit programs are used to calculate shadows (opaque and semi-transparant created by non-opaque objects) and an approximation of simple caustic effects. The formula used to create the caustics was developed using an approximation of the natural laws of optics.

```
float ri = clamp(refractiveIndex-1.0f, 0.0f, 1.0f);
if(ri>0.01f) {
  float exp = powf(2.0f,lerp(0.0f,9.0f,ri));
  sRes.attenuation *=powf(theta, exp)*refractiveIndex
                    *lerp(1.0f,5.0f,ri);
}
```

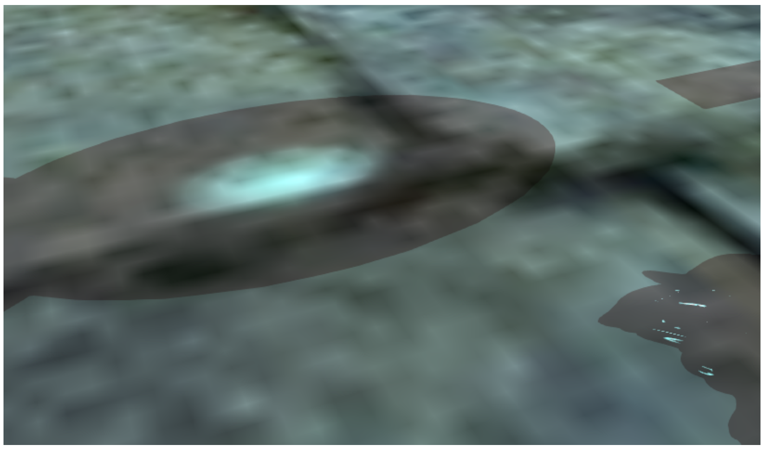


Figure 2.4: The approximated caustics

2.2.4 Translator component

To be able to render both by ray tracing and rasterization at the same time, both of the systems need to be able to operate on the same environment data, such as geometry, camera, textures, camera position and orientation and light sources. This is done by parsing the Ogre environment and, with the result, try to recreate the environment in Optix. Afterwards, the Translator component sets up a system to keep the Optix environment updated when data is changed in the Ogre setup. Using this method the data can be stored in the most efficient way possible for the specific rendering system but still be rendered at the same time. It also allows the user to concentrate on working with the Ogre environment and does not concern the user to worry with the Optix renderer at all. The drawback of this approach is that all data has to be stored redundantly and for big scenes. This can create a limitation caused by memory shortage.

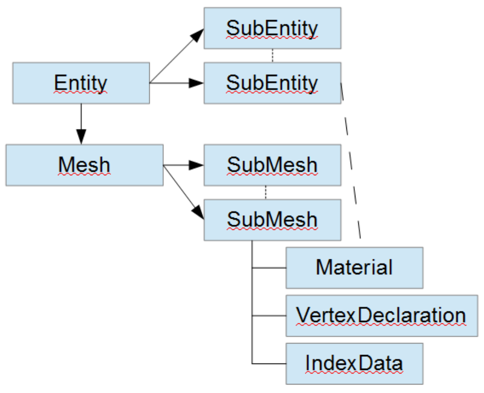
The structure of the translator system tries to mimic the system of Ogre as near as possible to fit into the different design patterns used in Ogre to keep components updated with the same data.

The first component that needs to be synchronized is the camera. To do this, the model-, view- and perspective matrices need to perform the same transformations in the Optix environment as it does in Ogre. However, there are some difference in how Optix and OpenGL handle the matrix stack. For instance, the Z-axis is reversed and the Optix system is built upon a coordinate system set up by three unit vectors (u, w, w). This demands some additional modification of the matrices at the right time in the pipeline. The following pseudo code demonstrates how the camera translation is performed in OgreHybrid.

```
function(input)
  fovY = input.fovY * 0.5
  fovX = tan^-1 (tan(inputFovY) * aspectRatio)

  gu = firstRowOfMatrix(input.modelViewPerspectiveMatrix)
  gv = secondRowOfMatrix(input.modelViewPerspectiveMatrix)
  gw = thirdRowOfMatrix(input.modelViewPerspectiveMatrix)

  ulen = input.focal * tan(fovX)
  vlen = input.focal * tan(fovY)

  u = ulen * gu
  v = vlen * gv
  w = focal * gw
end
```

To translate the geometry into the ray-tracing system, each Ogre node under the root node of the type “Entity” is used as base to create a HybridEntity and a HybridModel object. These objects are based on the position, scaling, orientation and materials of the Ogre Entity and the geometry of the Ogre Mesh assigned to the entity. The mesh data is parsed into an intermediate data structure similar to the Wavefront OBJ mesh format (Murray, 1996), since the Ogre Mesh data structure is a bit more advanced than the very simple system of Optix. This intermediate format also allows the translator to make use of other model loaders by just implementing a new parsing method for the new format. Due to different math systems used by Ogre and Optix all data points (vertices, normals and texture coordinates) have to be translated into the correct system to work correctly.

The Ogre mesh system is built on a concept that all models has a top mesh with an optional amount of shared vertex data and a specified number of “SubMeshes”. These submeshes contain a “VertexDeclaration” object that specifies the structure of that specific submesh, for instance if it is using shared vertices and a map to the pointers of the different data points. Each submesh may also contain a pointer to a material. The material assignments can also be done via the “SubEntity” without affecting the SubMeshs material pointer. Finally submeshes also have an “IndexData” object to keep track of what vertices, normals and texture coordinates that together form polygons.



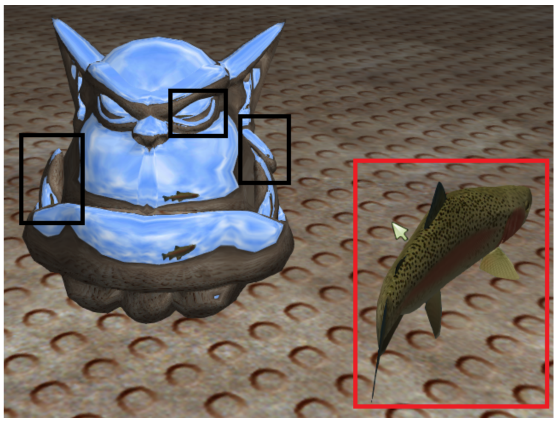


Figure 2.5: The structure of the Ogre entity and mesh system. In the image to the right, the black areas mark some of the submeshes while the red area marks a whole mesh.

The intermediate format consists of five simple lists: vertices (vec3), normals(vec3), texture coordinates(vec2), materials(Material) and polygons, where polygons are represented by a struct containing three indices to vertices, normals, texture coordinates and materials (integers that references an index in the lists). The intermediate format uses the glm library for math, since it is a generic and well-structured math library that is easy to cast to both the other systems (Riccio, 2014).

From the intermediate format, the program creates seven Optix buffer objects containing the vertex data, normal data, texture coordinate data, vertex index data, normal index data, texture coordinate index data and material index data. These buffers are then used to create a paged Optix Geometry object (assigned a Bounding box program and an Intersection program).

During the process of going through all Entities and converting their meshes into Optix format, each material used by any Entity in the loaded scene is converted and saved into a list used later to create the full scene. Ogre materials are a part of a very complex system with scriptable materials, custom shaders, with several possible passes and techniques. To translate the whole system would be a massive work and did not seem to be inside the scope of this thesis. However, to get a good approximation of most simple Ogre materials the translator takes the first pass of the first technique of the material and converts the Phong variables assigned to each material (ambient, diffuse, emissive, specular and shininess) and the most significant texture (the first in the texture list) from each material and uses this to create an Optix Material object. A material object contains a ClosestHit program and an AnyHit program. These are the same for all materials but have different values assigned to their variables depending on the base Ogre material.

To convert the texture some OgreHybrid had to access the underlying render system of Ogre to get the low level index of the texture. This is the reason why OgreHybrid is only usable with OpenGL. To port these parts to DirectX would not be a hard task and is a possible next step for OgreHybrid.

From all the HybridEntities and Materials created in the part of the scene parsing, the program creates a Transform object and GeometryInstance object. These types are specified by Optix and are used to create scenes to be fast to ray trace. The Transform specifies its children nodes position and orientation (much like a Model matrix, see Section 1.4.1 for more information). A GeometryInstance is a connection between a Geometry (a mesh) and a list of Materials – a fully defined object in a ray tracing scene.

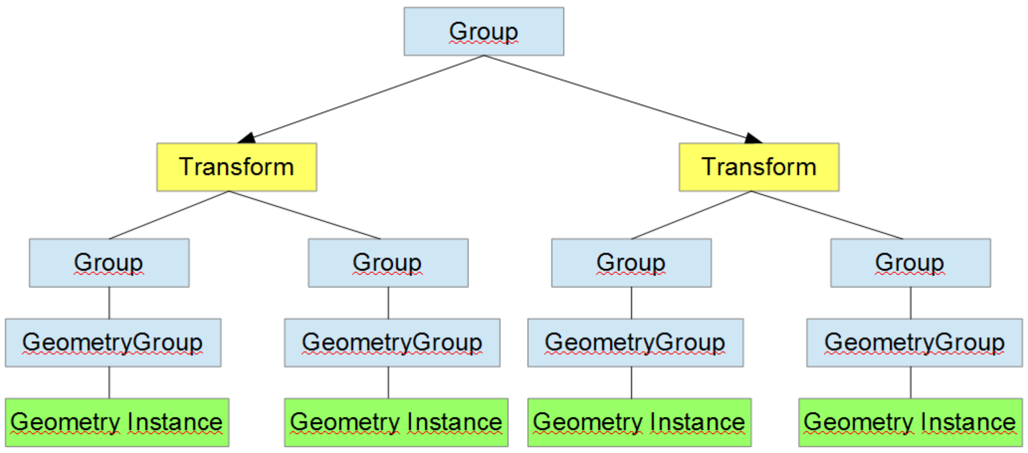


Figure 2.6: An Optix scene graph

To mimic an Ogre scene, the light sources also have to be imported from the rasterization engine. This is done simply by looping through the light sources in the Ogre scene and creating a sample struct object that is passed to Optix by a bufferobject.

Ogre uses a design pattern called “Factory pattern” to create some its objects, for instance entities. This means that whenever a new entity is supposed to be created, the engine calls the initialized entity factory and asks for a new object. This allows for a very flexible way to create objects with specific properties. OgreHybrid replaces the usual Entityfactory with the specially designed HybridEntityFactory. This creates HybridEntities that has some specific properties, such as a HybridMaterial associated with each SubEntity and SubMesh as well as a flag to mark the SubEntity as ray traced. The HybridMaterial keeps track of reflectivity, opaqueness and refractive index of the material. This means that all entities in an Ogre application that have OgreHybrid initialized has these properties. However, they are only used if the SubEntity is marked as ray traced.

To keep the environments synchronized between the ray tracer and Ogre, OgreHybrid utilizes the observerpattern implemented in the Ogre architecture. When a new Hybridobject is created (such as a scene node), the Hybridobject adds itself as a listener to the Ogreobject. When this object is manipulated in some way (for instance rotated, moved or removed), the Hybridobject gets this signal and tries to mimic its modification. Using this pattern, no polling of objects are needed to keep the environments synchronized. The only exception to this are the lights, where a simple command is needed to update all the lights. This is due to the way the lights are transferred to the GPU (as a buffer of structs). All the lights have to be reparsed each time one of the lighting changes. Therefore, the design choice was made to let the user update these each time the lighting changes.

3 Results

The results of the study shows that this approach of hybrid rendering is applicable and can be used within the boundaries set for this study.

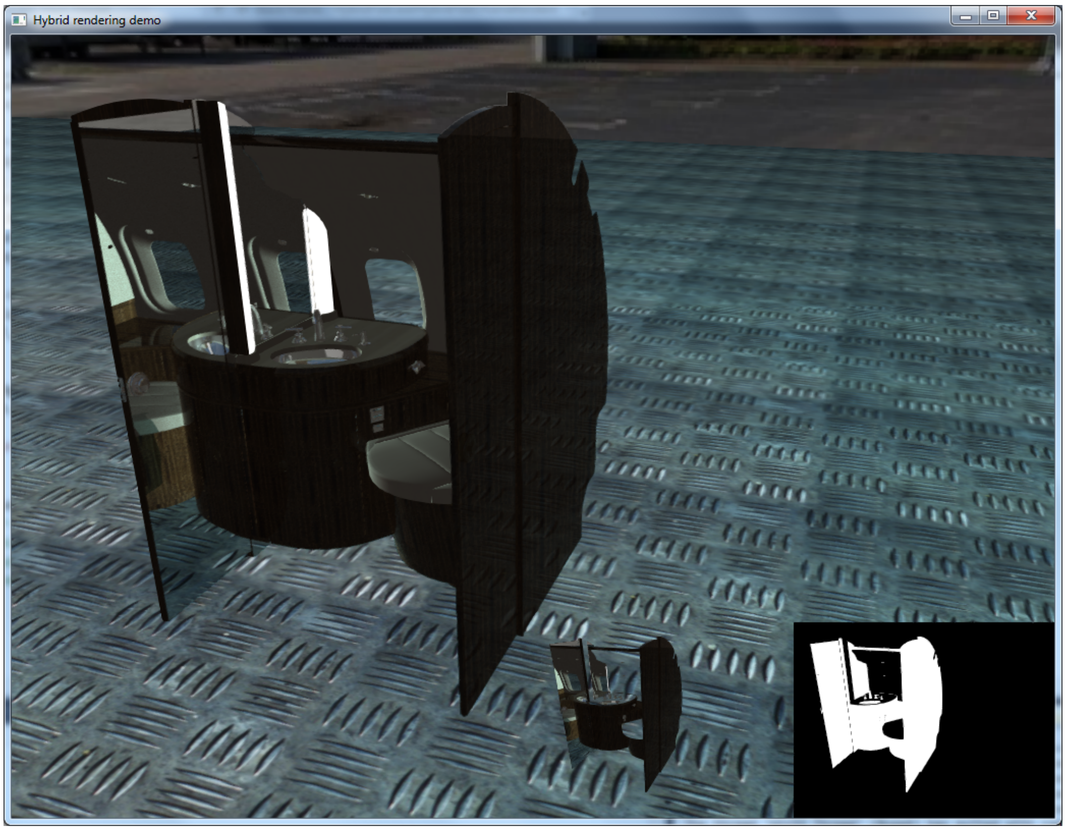


Figure 3.1: Results

This picture shows a hybrid rendered scene where the two miniscreens to the lower right on the screen shows, to the right, the mask layer that in is inserted into the ray tracer and, to the left, the buffer output from the ray tracer. The parts rendered black in the mask are rasterized in Ogre.

Scene	Rasterization	Hybrid rendering	Full ray tracing
Simple scene	1012	27 (20)	7 (2)
Intermediate scene	513	18 (8)	7 (2)
Complex scene	773	15 (6)	6 (1)

The numbers are representing frames displayed per second (fps) and the numbers inside the parenthesis is with supersampled ray tracing enabled. All numbers have FFXA enabled, since disabling it only had a minor effect on the result. The resolution was 1024x768 and operated in windowed mode.

These results were measured on a computer with Intel Dual Core 6600 2.4 Ghz with 8GB of RAM and an Nvidia Geforce 760 GTX. The tests were performed with the Nvidia Geforce driver 327.02 on Windows 7 Ultimate 64bit.

The first measured scene is a simple textured quad as floor and a non-textured wine glass with no reflection or refraction enabled. Only the glass is marked as ray traceable. The scene contained 19 625 triangles and had one moving and one static light source.

The second scene had two spinning, textured spheres, one reflexive and the other semi transparent and semi reflexive. It contained a solid dragon model and a spinning Ogre head. It also contained a transparent wine glass and a floor squad. Every object except the floor was ray traced. This scene contained 128 232 triangles and had the same lighting as the first scene.

The third scene was on an airplane bathroom (an industry used model for marketing private jet planes), where most of the scene was marked as ray traced. The model can be seen in Figure 3.3 (but another, more zoomed camera angle was used). The scene contained 82 778 triangles and had two moving light sources. The measurements were also performed with the transfer of the output buffer from the CPU to the GPU turned off (the output buffer is thus not transferred from the GPU to the CPU and back to the GPU as a texture every frame). This aimed to simulate a scenario where Optix is able to write directly to a texture or a solution in Ogre is implemented to reflag data on the GPU as texture memory (which is very easy to do in plain OpenGL). This resulted in the following table:

Scene	Rasterization	Hybrid rendering	Full ray tracing
Simple scene	1012	91	9
Intermediate scene	513	36	10
Complex scene	773	25	7

The first column is unaffected since the ray tracer is turned off and no ray tracing data is transferred between the CPU and the GPU. In the other columns however, it can be observed that the fps is up to 300% higher than the ordinary case.

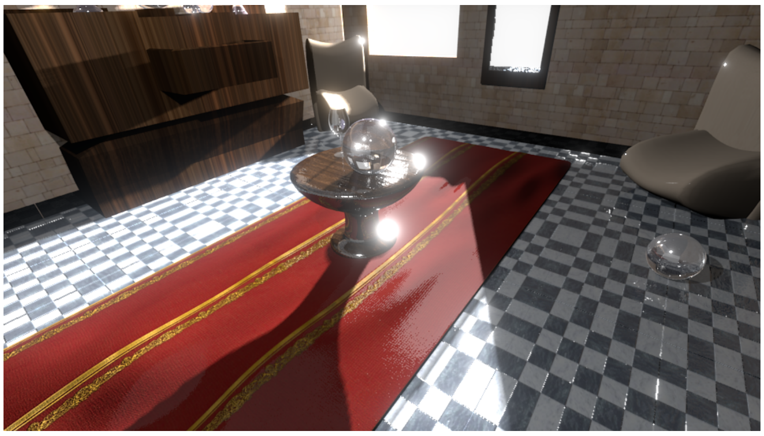


Figure 3.2: An inside picture

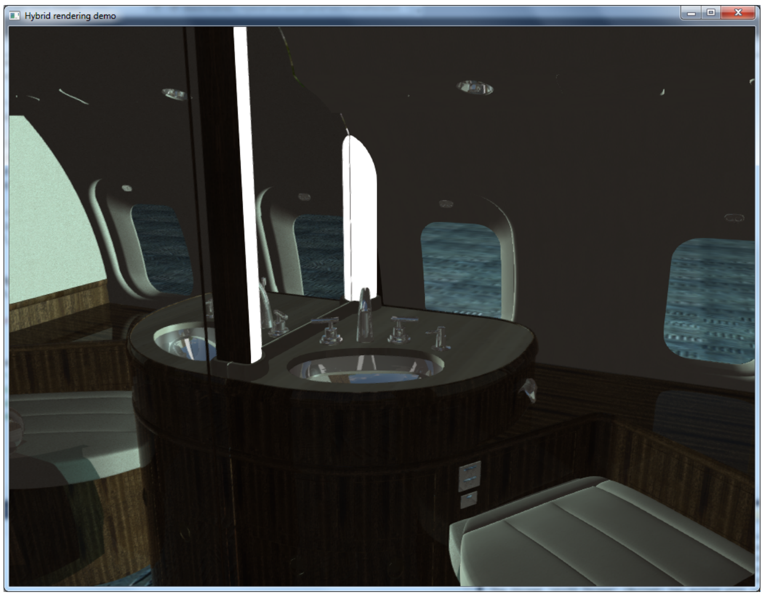


Figure 3.3: A bathroom scene from a private yet

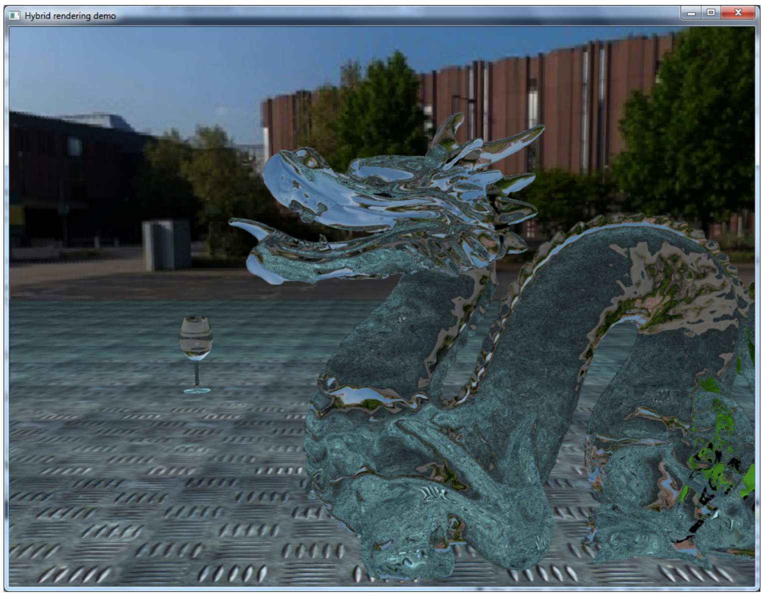
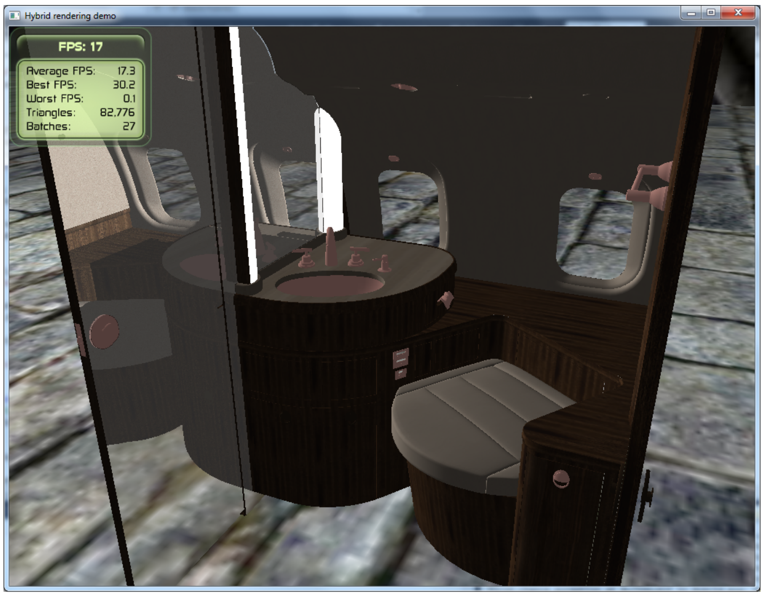


Figure 3.4: A simple scene with a Stanford dragon



```
FPS: 17
Average FPS:   17.3
Best FPS:      30.2
Worst FPS:      0.1
Triangles:   82,776
Batches:         27
```

Figure 3.5: A scene with moving light sources

4 Discussion and conclusions

Hybrid rendering as a technique is, as shown in this study, usable in some applications and can produce nice looking results. However, it is not applicable in all kinds of implementations and fits especially well for some topics. The technology contains in itself, by definition, some problems that has to be handled when using it.

4.1 The hybrid dilemma

The great drawback of hybrid rendering is the memory dilemma of double storage or ineffective rendering. To perform efficient ray tracing, the data has to be stored in specific data structures (spatial structures). These structures will not be the most efficient in most cases (if it is even possible) for storing the data for a rasterization rendering. This is especially problematic when applying a ray tracing solution to a graphics engine for rasterization, since this already includes data handling for the nodes in the scene.

This results in the dilemma of either redundantly storing all scene data in two different data structures (as proposed in this report) and taking the toll of using a lot of unnecessary memory or by using the same data structure but instead not getting the benefits of optimized data structures (and the ability to use an established engine which make a lot of code reusable).

Another problem with hybrid rendering approach (as mentioned in Section 1.4.3), is that the engines performance is heavily dependent on how much screen space that is currently occupied by ray traced objects. If the camera is zoomed onto a ray traced object so the whole screen is ray traced, the hybrid solution is not gaining any performance over a normal ray tracer, but will still be using double the memory (because of the hybrid dilemma). Using the hybrid approach therefore requires some assessment of the application that is using it, as well as the content that will be used by the application. The application will need to assert that the camera will not zoom in onto some object (or for instance accept framedrop, lower the resolution temporarily or stop ray tracing if the camera is too close a ray traced object).

It is theoretacally possible to share the actual data of geometry, shaders and textures between the ray tracer and rasterizer and only save pointers to the data inside the different optimized datastructures. This is however not possible using the ordinary OpenGL pipeline (for instance using the Ogre engine) and CUDA, since both CUDA and OpenGL allocates their own memory and does not yet support shared resources. An own rasterzer engine that use the hardware in an optimized way could be implemented in CUDA. Combined with the Optix framework this solution could be an interesting topic for further research.

4.2 Applications

Using the approach described in this report, it is possible to use hybrid rendering in interactive real-time systems in a professional environment at a usable speed for some applications. The solution works best in situations where the ray-traced objects take up a reasonably fixed part of the screen, since it is heavily dependent on how many pixels that need to be ray traced. The software's performance is also dependent on the resolution, so it is best fitted for applications which is not dependent on high resolution, or use hardware scaled to the resolution. The technology could be used in simulation technology where it is important that some parts of the scene are physically correct, for instance marketing and educational software for flight simulation or medical training software.

As of now, this technology is not good for development of highly interactive software that need to work on several different hardware, as well as letting the users change the camera angle and zoom in onto ray traced objects. These types of behavior are often wanted in most games. To make hybrid rendering appropriate for use in games, some other types of optimizations need to be implemented. Some approximations could be done by for instance only perform the ray tracing step when the camera is moving, precompute some angles and blending between ray traced frames to make a new approximation. Another approach that could be used is to only activate the ray tracing when the environment is static, for instance by using an environment map to approximate the appearance of the semitransparent objects until the user fixates the camera, and then slowly blend over to the ray tracing solution, making the non-opaque objects rendered more physically correct.

Hybrid rendering techniques such as this one is usable as a way to use ray tracing in real applications now. In the future, when the graphics hardware is faster and more customizable (or optimized for ray tracing purposes), ray tracing have the possiblity to be the standard way to produce computer graphics. If this is the case, the hybrid theories with all of its cons, such as redundant data structures for storing the scene, will no longer be needed. Then, all computational power and memory can be used to optimize the ray-tracing calculations.

4.3 Alternative approaches

The combination of ray tracing and rasterization could have been done in several other ways than the approach of this thesis. These where discussed in the pre study of this report, but was discarded since they all had some problems that made it unfeasible to prove that they could improve the visual quality within the limits set up in the purpose of the study.

One of the first discussed solutions was to implement blended ray tracing. To render a full ray traced image each frame as well as a rasterized one and then blend the pictures together and calculate the balance between the two images uniquely for each pixel, submesh or mesh. This solution would probably produce better looking visual results than the solution that is proposed in this thesis. However, it would be a lot more demanding on the hardware as well as more depending on the resolution of rendered image.

Another proposed solution was to write a new ray tracer in GLSL and simply apply the hybrid technology in a post-process shader onto the Ogre rendering. This would however result in a lot of work being put into creating the ray-tracer basics, as well as doing optimizations and work-arounds to access the geometry data. To be efficient, the ray tracer need to save data in the GPU memory and access it through optimized spatial data structures. Due to limitations in GLSL, it is not possible to access the position of vertices in post-processing pixel shader (where a ray tracer would be implemented). Therefore, to implement a GLSL ray tracer all mesh data have to be sent from the CPU each frame. There is also a limit to how much data that can be handled in arrays in GLSL, making it hard to support larger scenes.

As an addition to the proposed solution in this thesis, a technique for identifying the areas of the screen that benefited the most from the ray tracing was discussed. This, however, was scoped out of the thesis after some domain research. In all scenes used, it was easy by hand to identify what materials that would be better with ray tracing (reflective and semitransparent materials was clearly the two properties of a material that benefited most of the technique). Since this feature was not interesting from the perspective of the task manager, it was not prioritized in the planning work.

4.4 Method choices

The choice of using Optix, OpenGL, OGRE3D and C++ where prerequisites in this study to make it feasible to perform. However, it also, naturally, limits the possibilities of testing the extent of hybrid technologies. These tools (with the exception of Optix) are commonly used in the industry, and all of them have a lot of supporters, custom tools and extensions that are freely available to be used in research.

Optix is still a relatively new product but has a good modular structure and is a good model to create future rendering API:s from. The limitations to Nvidia hardware and CUDA as a language will potentially harm the spreading of a standardization in ray tracing API:s, but an open standard, potentially based on OpenCL, will probably be agreed upon before this becomes a problem for ray tracing to become market standard for heavy real time computer graphics.

4.5 Hardware dependencies

Since Optix is a fairly new technology and ray tracing as a concept demands a lot of computational power, OgreHybrid was bound from the beginning to require quite a lot from the host computer. However, a goal of this thesis was to make it runnable on today's hardware. This was defined to be a graphics card based on Nvidias Fermi (as Geforce 760 GTX) and a quite modern dual core processor. However, this limited what was possible to do with Optix, since many of the more advanced features of CUDA 5.5 requires a Kepler-based GPU. For a proof of concept, however, this was never a requirement. For future research, the recommendation is to use more bleeding edge hardware, since this technology with high probability will be available to normal users within short. This thesis also showed that this technology is not recommended to use for programs where the user has full control of the camera angles, zooming and/or moving freely around the virtual world. With this in mind and the fact that most of the graphics heavy applications run on normal personal computers are these types of software, there is really no reason to design the technology for hardware that has been around the market for a couple of years.

4.6 Ethical complications

Since the proposed technology itself does not introduce new functionality or a new physical product, the model of ecological footprint at first sight is not affected nor the economy of a company. However since the technology improves the presentation layer of applications, could improve selling of commercial products that use it, that may result in more produced and shipped hardware products. The technology can potentionally demand an upgrade of present hardware and marginally increase the power used by processors, GPU:s and fans.

The technology can be used to further improve the visual quality to have a more realistic appearance of applications. This could be used to improve medical training software or educational software as well as military simulators or software used in for instance the oil industry.

The choice of an MIT license of the source code for OgreHybrid was made since the technology and knowledge is supposed to be free to use, improve as well as the possibility to make use of it in commercial products. This could be essential for some small developers or new companies to be established regardless of the product or service that is sold. These consequences and the concept of open knowledge was redeemed to be more important than the risk of the technology allowing actions that might hurt mankind. Especially since the hybrid technology in itself does not create any new incitement to develop new such products or services.

5 Future work

The OgreHybrid plugin is fully functional for basic tasks. However it could be improved in several ways to accomplish more advanced tasks and make it more applicable for specific purposes. First of all, the most significant bottleneck in the program is the transfer of data from Optix back into a texture in OpenGL (Ogre). This step will probably be easy to solve in future versions of Optix when it is possible to write to specific texture memory from inside CUDA. There is a feature available in CUDA 5.5 called surfaces that aim to accomplish this. These features are, however, not available in Optix yet. This optimization would grant an up to 300% performance boost to OgreHybrid.

With more advanced hardware, effects like Monte Carlo-raytracing, soft shadows and physically correct caustics would be interesting to investigate further.

The OgreHybrid plugin would also benefit from a more flexible structure when it comes to compiled programs. One such possibility to be explored is to allow for developers to dynamically change specific entities to use certain ClosestHit and AnyHit programs. This could possibly speed up some environments as well as allowing for new graphical effects to be explored.

There are also several possibilities to improve the concept of hybrid rendering further, by for instance making it possible to use classic shader effects in the raytracing engine. This could be done by baking the environment textures in Optix at startup with the normal shaders for static shaders or to run an extra pass in Ogre to rewrite dynamic surfaces each frame (this would be very costly). This would make reflections even more accurate when working with advanced shaded environments, as well as provide an easy and similar way of creating shaders in a whole program.

References

James D. Murray, W. v. R. (1996), *Encyclopedia of Graphics File Formats, 2nd Edition*, O'Reilly Media, USA, pp. 874–876.

Johnson, T. E. (1963), 'Sketchpad iii: a computer program for drawing in three dimensions', *Proceedings of the AFIPS Spring Joint Computer Conference* pp. 347–353.

Kajiya, J. T. (1986), 'The rendering equation', *In SIGGRAPH '86* pp. 143–150.

Michael Steffen, J. Z. (2009), 'Design and evaluation of a hardware accelerated ray tracing data structure', *Theory and Practice of Computer Graphics* pp. 101–108.

Ogre (2014), Ogre3d api - ogre::plugin class reference.
URL: *http://www.ogre3d.org/docs/api/1.9/class_ogre_1_1_plugin.html#details (2014-07-16)*

Phong, B. T. (1975), 'Illumination for computer generated pictures', *Communications of the ACM* pp. 311–317.

Pohl, D. (2008), Hybrid rendering: Combining ray tracing and rasterization.
URL: *http://www.pcper.com/reviews/Processors/Ray-Tracing-and-Gaming-One-Year-Later/Hybrid-Rendering-Combining-Ray-tracing-and- (2014-07-16)*

Riccio, C. (2014), Opengl mathematics (glm).
URL: *http://www.g-truc.net/project-0016.html (2014-07-16)*

Sommerville, I. (2007), *Software Engineering, 8th Edition*, Pearson Education Limited, England, pp. 498–505.

Stallman, R. (2013), Free software is even more important now.
URL: *https://www.gnu.org/philosophy/free-software-even-more-important.html (2014-07-17)*

Steven G. Parker, James Bigler, A. D. H. F. J. H. D. L. D. M. M. M. K. M. A. R. M. S. (2010), 'Optix: a general purpose ray tracing engine', *In SIGGRAPH '10* pp. 93–101.

Tarver, M. (2009), The problems of open source.
URL: *http://www.lambdassociates.org/blog/the_problems_of_open_source.htm (2014-07-17)*

Tomas Akenine-Möller, E. H. & Hoffman, N. (2008), *Real-Time Rendering 3rd Edition*, A. K Peters, Ltd., Natick, MA, USA, pp. 26–27, 53–71, 110–116, 647–654.

Whitted, T. (1980), 'An improved illumination model for shaded display', *Magazine Communications of the ACM, Volume 23 Issue 6* pp. 343–349.